Incomplete Recovered Document with document format translation !! issues !!

IMAGINE 2

The IMAGe engINE

Documentation & User Manual

May 1997

VERSION 0.70

Chapter 3

THE PROGRAMMING MODEL

THE **IMAGINE 2** CORE CONTROL REGISTERS

CONTROL REGISTER	UNIT	NR			
cr0	REG	1	Register file control register	cr0	REG_Control
cr1	REG	1	Extended instruction control	cr1	REG_Monitor
cr2	REG	1	Vector index control register	cr2	REG_Vector
cr3	REG	4	Vector indices entry / write delay line entry	cr3	REG_Fifo
cr4	REG	1	Vector indices port A	cr3.0	REG_A_Indices
cr5	REG	1	Vector indices port B	cr3.1	REG_B_Indices
cr6	REG	1	Vector indices port C	cr3.2	REG_C_Indices
cr7	REG	1	Vector indices write enable & status	cr3.3	REG_C_Flags
cr8	BSH	1	Q BUS register	Q-bus	BSH_Qbus
cr12	ALU	1	F BUS register	F-bus	ALU_Fbus
cr13	ALU	1	Three operand logic function	cr6	ALU_Logic
cr15	ALU/RNG	1	Status register	cr5	ALU_RC_Status
cr16	MAC	1	M BUS register	M-bus	MAC_Mbus
cr17	MAC	1	Multiplier control register 1	cr12	MAC_Control1
cr18	MAC	1	Multiplier control register 2	cr16	MAC_Control2
cr19	MAC	1	Vector & Coefficient pointers	cr13	MAC_RamPtrs
cr20	MAC	16	Coefficient registers entry	new	MAC_Coef
cr21	MAC	1	MAC pipeline output	cr15	MAC_Pipe
cr22	MAC	1	Lower limit compare register [31:00]	new	MAC_LoLimit0
cr23	MAC	1	Lower limit compare register [63:32]	new	MAC_LoLimit1
cr24	MAC	1	Higher limit compare register [31:00]	new	MAC_HiLimit0
cr25	MAC	1	Higher limit compare register [63:32]	new	MAC_HiLimit1
cr26	MAC	1	Low limit register (32 bit)	cr17	MAC_LoLimit32
cr27	MAC	1	High limit register (32 bit)	cr18	MAC_HiLimit32
cr28	MAC	1	Accumulator register [31:00]	cr14.0	MAC_Accu0
cr29	MAC	1	Accumulator register [63:32]	cr14.1	MAC_Accu1
cr30	MAC	1	State save & restore entry	cr19	MAC_Save
cr32	UFU	1	U BUS register	U-bus	UFU_Ubus
cr33	UFU	1	IEEE 754 float conversion register	cr8	UFU_IEEE
cr36	DIO	1	D BUS register	D-bus	DIO_Dbus
cr37	DIO	1	Data I/O control register	cr20	DIO_Control
cr38	DIO	1	Data I/O address register	cr21	DIO_Address
cr39	DIO	1	Data I/O linear offset register	new	DIO_offset
cr40	I3D	1	3D graphics Look up table indices	new	I3D_ClutIndices
cr41	I3D	256	3D graphics Look up table entry	new	I3D_ClutData
cr42	I3D	1	3D graphics Color Key Low values	new	I3D_ColorKeyLo
cr43	I3D	1	3D graphics Color Key High values	new	I3D_ColorKeyHi
cr44	VIO	1	V BUS register	V-bus	VIO_Vbus
cr45	VIO	1	Vector I/O control register 1	cr24	VIO_Control1
cr46	VIO	1	Vector I/O control register 2	new	VIO_Control2
cr47	VIO	1	Vector I/O alpha test & generation	new	VIO_Alpha
cr48	VIO	640	Vector I/O translation table entry	new	VIO_TableData
cr49	VIO	1	Vector I/O Transparent Output Color	new	VIO_Transparent
cr50	VIO	1	Vector I/O Color Key Low values	new	VIO_ColorKeyLo
cr51	VIO	1	Vector I/O Color Key High values	new	VIO_ColorKeyHi
cr52	SEQ	1	Sequencer status control register	cr32	SEQ_Status
cr53	SEQ	1	Program counter	cr33	SEQ_PrCounter
cr54	SEQ	1	Address register	cr34	SEQ_Address
cr55	SEQ	1	Interrupt table register	cr35	SEQ_Interrupt
cr56	SEQ	1	Repeat count register	cr36	SEQ_Repeat
cr57	SEQ	1	Maximum repeat count	cr37	SEQ_MaxRepeat
cr58	SEQ	1	Sequencer test register	cr39	SEQ_Test
cr60	SEQ	1	Instruction cache store register 0	cr30	ICA_Low
cr61	SEQ	1	Instruction cache store register 1	cr31	ICA_High

THE **IMAGINE 2** CORE CONTROL REGISTERS (continued)

CONTROL REGISTER	UNIT	NR			
cr64	I3D	1	Linear interpolator control register	new	I3D_Control
cr65	I3D	1	Texture mapping control register	new	I3D_Texture ?
cr66	I3D	1	Depth buffer control register	new	I3D_ ?
cr67	I3D	1	Low level control register	new	I3D_LowLevel ?
cr68	I3D	8	Lighting Alpha component parameter entry	new	I3D_Alpha ?
cr69	I3D	8	Lighting Red component parameter entry	new	I3D_Red ?
cr70	I3D	8	Lighting Green component parameter entry	new	I3D_Green ?
cr71	I3D	8	Lighting Blue component parameter entry	new	I3D_Blue?
cr72	I3D	8	Texture Q co-ordinate parameter entry	new	I3D_TextureQ ?
cr73	I3D	8	Texture R co-ordinate parameter entry	new	I3D_TextureR ?
cr74	I3D	8	Texture S co-ordinate parameter entry	new	I3D_TextureS ?
cr75	I3D	8	Texture T co-ordinate parameter entry	new	I3D_TextureT ?
cr76	I3D	12	Depth 1/Z co-ordinate parameter entry	new	I3D_Depth ?
cr77	I3D	10	Lighting Fog Attenuation factor entry	new	I3D_Fog ?
cr78	I3D	1	Border color register	new	I3D_Border ?
cr79	I3D	16	Texture MIP map address offset table entry	new	I3D_MipMap ?
cr80	MES	1	Motion Estimator control register	new	MES_Control
cr81	MES	1	Sum of Differences	new	MES_SumOfDiff
cr82	MES	1	Minimum value found	new	MES_Minimum
cr83	MES	1	Position of the minimum value	new	MES_Position
cr88	MSK	1	Image mask control register 1	cr40	MSK_Control1
cr89	MSK	1	Image mask control register 2	cr41	MSK_Control2
cr90	MSK	1	Window X minimum / maximum	cr42	MSK_Window_X
cr91	MSK	1	Window Y minimum / maximum	cr43	MSK_Window_Y
cr93	MSK	1	Polygon Y minimum / maximum	new	MSK_Polygon_Y ?
cr94	MSK	1	Polygon start coordinate entry	cr44	MSK_PolyStart
cr95	MSK	1	Polygon end coordinate entry	cr45	MSK_PolyEnd
cr96	MSK	1	Polygon start/end coor.register	cr46	MSK_PolyCoord
cr97	MSK	4	Spanline start coordinate register [3:0]	new	MSK_SpanStart
cr98	MSK	4	Spanline end coordinate register [3:0]	new	MSK_SpanEnd
cr99	MSK	4	Spanline start/end coord. Register [3:0]	cr47	MSK_SpanLines
cr100	MSK	1	Spanline start edge delta value	new	MSK_DeltaStart ?
cr101	MSK	1	Spanline end edge delta value	new	MSK_DeltaEnd ?
cr102	MSK	1	Span Line Length	new	MSK_SpanLength ?
cr103	MSK	1	Span Line Address	new	MSK_SpanAddr ?
cr104	MSK	8	Complex alpha mask register [1:0][3:0]	cr48	MSK_CplxAlpha
cr105	MSK	8	Range clip mask register [1:0][3:0]	cr49	MSK_RangeClip
cr106	MSK	8	Transparent mask register [1:0][3:0]	cr50	MSK_Transp
cr107	MSK	8	Opaque mask register [1:0][3:0]	cr51	MSK_Opaque
cr112	VAU	1	Vector access control register	cr52	IMM_Control
cr113	VAU	1	Bit plane mask register	cr53	IMM_PlaneMask
cr114	VAU	1	Foreground color register	cr54	IMM_FG_color
cr115	VAU	1	Background color register	cr55	IMM_BG_color
cr116	VAU	1	Image 1 XY pointer (mask ref.)	cr57	IMM_Image1
cr117	VAU	1	Image 2 XY pointer	cr58	IMM_Image2
cr118	VAU	1	Image 3 XY pointer	cr59	IMM_Image3
cr119	VAU	1	Display XY size register	new	IMM_DispSize
cr120	VAU	1	Image 1 offset address	new	IMM_Offset1
cr121	VAU	1	Image 2 offset address	new	IMM_Offset 2
cr122	VAU	1	Image 3 offset address	new	IMM_Offset 3
cr123	VAU	1	Display offset address	new	IMM_DispOffset
cr126	EMI	?	External Memory Interface Address	new	EMI_Address ?
cr127	EMI	?	External Memory Interface Data	new	EMI_Data ?

THE **IMAGINE** 2 CORE REGISTERS

The IMAGINE has about 900 core registers. They play a central role in the programmers model.

The control registers
Almost all functional units in the IMAGINE contain so called control registers. The data processing units can refer to these registers during so called extended instructions. Some of these control register entries give access to multiple similar registers (normally 4). These registers are accessible in an auto increment way with preset-able 2 bit pointers (a 64 bit version of the IMAGINE will have 8 instead of 4 registers on these entries). The programmer sees the control registers as 128 extra registers in the three port register file. The B read port has a choice of 120 normal and 128 control registers for reading while the write port can write to 120 normal and 128 control registers. Two control registers can be accessed each cycle (one read access and one write access).

The bus register/drivers
All data processing and I/O units contain a 'bus-register' which contents can be used by other units.

A bus:	three port register file read port
B bus:	three port register file read port
Q bus:	the barrel shifter result.
F bus:	the ALU result.
M bus:	the multiplier/accumulator result.
U bus:	the unary function unit result.
D bus:	the data memory I/O register.
V bus:	the image memory I/O register.

These bus registers are visible in the native instruction language:
AB = rd(r43,cr36) -> **F** = add(**A,B**) -> wr(cr36, **F**);

The three port register file registers
The three port register file contains 120 general purpose registers and 256 vector registers. Two ports can read data and one port can write data each cycle. Some of these registers have a predefined function for the C compiler. A direct access to register 63 with any of the 3 ports is a no op for this port. The vector registers are accessible by the vector index generator. One read port and one write port are used to access the control registers.

The multiplier/accumulator register file
The MAC contains 64 internal 128 bit registers which can be used as 128 registers of 64 bit (the typical vector length is 64). These registers can be used for vector accumulation operations as well as parameters for differential engine type operations in combination with the accumulator. These registers can further be split up in the typical HISC way. A 64 bit register can also be a double 32 bit and a quadruple 16 bit register.
A 128 bit register can be a double 64 bit and a quadruple 32 bit register. An example of vector accumulation is image filtering: A 3x3 convolution adds three vector while a 4x4 convolution adds four vectors. Examples off differential engine functions are: Gouraud shading interpolation and 2D or 3D coordinate calculation (linear, Bezier spline etc..).

The multiplier pipeline registers
These registers are used in those 8 bit multiplications which make the most efficient use of the HISC multiplier. The 16 multiplications (4x4) per cycle performed by the matrix times vector multiplication and the quadruple inproduct multiplication require up to 32 bytes as input operands per cycle. The pipeline register provides these operands to the multiplier. The important graphics and image processing algorithms like interpolated rotation and scaling, discrete cosine transformation, color space conversion, convolution and correlation are directly supported.

The sequencer on chip micro stack
This little on chip stack can be used to speed up
- Library function
- Assembly code
- Non recursive C-functions.
- Interrupts.

examples: The on chip micro stack can be used to speed up a number of elementary library functions such as division, emulated floating point functions etc. These functions can use the micro stack and scratch pad registers to implement a call/return mechanism with only 2 or 3 cycles overhead, much less as the normal C compiler. The inner loops of assembly code might be implemented as a call rather than a loop in functions with lots of options. The call address is determined in the initialisation phase of the function and used in the inner loop. The C compiler can use the stack for return address saving in non-recursive functions. At least all functions who do not call other functions can use this mechanism. The situations is more complex for functions which do contain calls. Horizontal sync interrupts: a simple horizontal sync interrupt routine or DRAM refresh interrupt is as short as 2 instructions, one for the required function and one return instruction. No state saving is needed because of the micro stack. The total time needed for the interrupt including the branch delays of the jump to the interrupt vector table and the return is only 5 cycles: 100 ns at 50 MHz.

THE **IMAGINE** INSTRUCTION SETS

THE HIERARCHY OF INSTRUCTION LEVELS

The IMAGINE has an hierarchical instruction set to provide compatibility with existing software, compilers and operating systems at one end, and highly efficient parallel vector processing at the other end. The various instruction levels can be intermixed freely in the assembler code. All instruction levels can be written as "In Line" code in C and C++ programs.

We differentiate between the following levels:

(♦) Compiler based C code and C++ code
(compiler generated)

♦ RISC/CISC level assembly code
(macro function set)

♦ Free pipeline assembly code
(native machine language: graphs of pipeline sequences)

♦ Vector processing level
(set of macro functions for vector operations and user defined vector operations)

♦ Specific use of control registers and special purpose graphics hardware.

The RISC/CISC instruction set (which is hierarchically the highest of the four levels of assembly code) contains typical register based instructions like:

mnemonic:	function:
mul_(p1, c10, v0);	p1 = c10 * v0
add_(v1, v2, v3);	v1 = v2 + v3
abs_(vx, vy);	vx = abs(vy)
st_byte_(data, val);	*(ptr) = val

This level is the 'interface layer' to compilers, existing graphics and image processing software written in high level language, operating systems etc. The RISC/CISC level is a macro function level within the IMAGINE assembler. The functions above are expanded to small instruction graphs defining the sequence of individual pipeline stages; typically: read registers, execute function, write back register. This notation level is the native instruction language of the IMAGINE.

```
AB = rd( c10, v0)       ->  M = mult(A, B, iss) ----->  wr(p1,M);
AB = rd( v2,v3)                  ->  F = add(A,B)         ->  wr(v1,F);
A   = rd( vy)           ->  U = abs(A)            ->  wr( vx, U);
AB = rd( ptr, val)               ->  DA = wrAd(A), D = byte(B);
```

The RISC/CISC level includes all functionality needed to interface to the code generation part of modern optimising compilers. In this sense it provides already more functionality than most RISC processors do, which generally lack explicit functions for byte and 16 bit operations and conversions between the various
formats (compilers need to include up to 3 or 4 barrel shift operations for a single 8 bit instruction). Almost all IMAGINE instructions at this level can be orthogonally used on all word lengths.

We present version 1.0 of the RISC/CISC level instructions in this document. The instruction set can be expanded since it is a macro function set. Some instructions are added which are more CISC-like, such as the multiple stack push and pop operations. The bit operations are included in the Intel processors since the 386 are implemented as a CISC example. The instructions take 2 to 3 pipeline slots in some contrast with the 486 which needs 3 to 103 cycles. The programmer will use the RISC/CISC type operations in the areas outside the inner loops of the graphics and image processing where the lowest programming levels are used. RISC/CISC type assemble code will normally be used in the preparation stage. This stage can be predominant in some graphics functions such as Gouraud shaded triangles. Some of this code can then be replaced during an optimisation stage with native IMAGINE instructions which can perform up to 3 or 4 instructions per cycle.

So the lowest level functions are found in the innermost loops of the code where Gouraud interpolated pixels are drawn. Textures mapped from a source to a destination area, images filtered etc. The RISC/CISC level is useful in the preparation stage of these function where parameters are handled, prepared and transformed to the format needed in the inner loops

The capability to go smoothly and instantaneously from hardware like pixel oriented processing to higher level parameter processing up to C or C++ code and back is one of the great advances of the IMAGINE processor. The lack of this facility cripples other attempts found in the market place which at one end of the spectrum use multiple on chip 8, 12 or 16 bit ALU's for pixel processing but break down in terms of speed when higher level operations are required. As well as RISC-like processors which have included some very specific hardware for certain graphics operations. These special purpose circuits have to communicate through or along side autonomous bus handling units which have to schedule cache line reads and writes, bus snooping operations for cache coherency etc. These 'non deterministic' operations do hinder both the hardware and programmer to reach the very high sustained I/O speeds needed for high performance graphics.

THE RISC/CISC LEVEL INSTRUCTIONS (1)

LOAD and STORE accesses to DATA MEMORY

mnem.	operands		cycl.	function
ld_byte_	(dst, base, offset)	1	yes	Load sign extended byte from *(base+off) into dst
ld_short_	(dst, base, offset)	1	yes	Load sign extended short from *(base+off) into dst
ld_word_	(dst, base, offset)	1	yes	Load word from *(base+off) into dst
ldu_byte_	(dst, base, offset)	1	yes	Load zero extended byte from *(base+off) into dst
ldu_short_	(dst, base, offset)	1	yes	Load zero extended short from *(base+off) into dst
ldu_word_	(dst, base, offset)	1	yes	Load word from *(base+off) into dst
st_byte_	(src, base, offset)	2	yes	Store byte from src to *(base+off)
st_short_	(src, base, offset)	2	yes	Store short from src to *(base+off)
st_word_	(src, base, offset)	2	yes	Store word from src to *(base+off)
std_byte_	(src, addr)	1	yes	Store byte from src to *(address)
std_short_	(src, addr)	1	yes	Store short from src to *(address)
std_word_	(src, addr)	1	yes	Store word from src to *(address)
push_	(stack, src)	2	no	Push (control-) register to stack
push2_	(stack, src1, src2)	3	no	Push 2 (control-) registers to stack
push3_	(stack, src1 .. src3)	4	no	Push 3 (control-) registers to stack
push4_	(stack, src1 .. src4)	5	no	Push 4 (control-) registers to stack
push5_	(stack, src1 .. src5)	6	no	Push 5 (control-) registers to stack
push6_	(stack, src1 .. src6)	7	no	Push 6 (control-) registers to stack
push7_	(stack, src1 .. src7)	8	no	Push 7 (control-) registers to stack
push8_	(stack, src1 .. src8)	9	no	Push 8 (control-) registers to stack
pop_	(stack, dst)	2	no	Pop (control-) registers from stack
pop2_	(stack, dst1, dst2)	3	no	Pop 2 (control-) registers from stack
pop3_	(stack, dst1 .. dst3)	4	no	Pop 3 (control-) registers from stack
pop4_	(stack, dst1 .. dst4)	5	no	Pop 4 (control-) registers from stack
pop5_	(stack, dst1 .. dst5)	6	no	Pop 5 (control-) registers from stack
pop6_	(stack, dst1 .. dst6)	7	no	Pop 6 (control-) registers from stack
pop7_	(stack, dst1 .. dst7)	8	no	Pop 7 (control-) registers from stack
pop8_	(stack, dst1 .. dst8)	9	no	Pop 8 (control-) registers from stack

dst:	DESTINATION	register or control register
src:	SOURCE	register or control register
base:	ADDRESS-BASE	register or control register
offset:	ADDRESS-OFFSET	register or immediate
address:	ADDRESS	register or control register
stack:	STACK-POINTER	register (or control register)

Sizeable instructions can be used to access an allocated 2 dimensional or 3 dimensional area in data memory:

2D example: **ld_word_16** (dst, base, offset)
3D example: **std_byte_8** (src, address)

base:	2D/3D BASE	register or control register
offset:	2D/3D OFFSET	register
address:	2D/3D ADDRESS	register or control register

	X-component	Y-component	Z-component
2D address:	bits 16..31	bits 0..15	---
3D address:	bits 16..23	bits 8..15	bits 0..7

THE RISC/CISC LEVEL INSTRUCTIONS (2)

REGISTER MOVE INSTRUCTIONS

mnem.	operands	cycl.	size.	function
move_	(dst, src)	1	yes	Move (control-) register to (control-) register
movi_	(dst, imm)	1	yes	Move immediate value to (control-) register
swap_	(reg, opB)	2	yes	Swap register with (control-) register

ARITHMETICAL ALU INSTRUCTIONS

mnem.	operands	cycl.	size.	function
add_	(dst, opA, opB)	1	yes	dst = opA + opB
sub_	(dst, opA, opB)	1	yes	dst = opA - opB
rsub_	(dst, opA, opB)	1	yes	dst = opB - opA
incr_	(dst, opB)	1	yes	dst = opB + 1
decr_	(dst, opB)	1	yes	dst = opB - 1
addincr_	(dst, opA, opB)	1	yes	dst = opA + opB + 1
subdecr_	(dst, opA, opB)	1	yes	dst = opA - opB - 1
rsubdecr_	(dst, opA, opB)	1	yes	dst = opB - opA - 1

LOGICAL ALU INSTRUCTIONS

mnem.	operands	cycl.	size.	function
clear_	(dst)	1	yes	dst = '0000'
set_	(dst)	1	yes	dst = 'FFFF'
invert_	(dst, opB)	1	yes	dst = !opB
and_	(dst, opA, opB)	1	yes	dst = opA & opB
nand_	(dst, opA, opB)	1	yes	dst = !(opA & opB)
or_	(dst, opA, opB)	1	yes	dst = opA | opB
nor_	(dst, opA, opB)	1	yes	dst = !(opA | opB)
xor_	(dst, opA, opB)	1	yes	dst = opA ^ opB
equiv_	(dst, opA, opB)	1	yes	dst = !(opA ^ opB)
andrev_	(dst, opA, opB)	1	yes	dst = !opA & opB
andinv_	(dst, opA, opB)	1	yes	dst = opA & !opB
orrev_	(dst, opA, opB)	1	yes	dst = !opA | opB)
orinv_	(dst, opA, opB)	1	yes	dst = opA | !opB

dst:	DESTINATION	register or control register
opA:	OPERAND A	register or immediate
opB:	OPERAND B	register or control register
reg:	REGISTER	register
imm:	IMMEDIATE	immediate value

Sizeable instructions operate on double 16 bit and quadruple 8 bit data:

double 16 bit example: **add_16** (dst, opA, opB)
quadruple 8 bit example: **shl_8** (dst, imm, opB)

THE RISC/CISC LEVEL INSTRUCTIONS (3)

BARREL SHIFT/ROTATE INSTRUCTIONS

mnem.	operands	cycl.	size.	function
shflog_	(dst, opA, opB)	1	yes	dst = shift opB over opA places logical, (bi-directional)
shfarh_	(dst, opA, opB)	1	yes	dst = shift opB over opB places arithmetic, (bidirect.)
rotate_	(dst, opA, opB)	1	yes	dst = rotate opB over opA places (bi-directional)
shr_	(dst, imm, opB)	1	yes	dst = shift logical right opB over imm places.
shl_	(dst, imm, opB)	1	yes	dst = shift logical left opB over imm places.
sar_	(dst, imm, opB)	1	yes	dst = shift arithmetical right opB over imm places.
sal_	(dst, imm, opB)	1	yes	dst = shift arithmetical left opB over imm places.
ror_	(dst, imm, opB)	1	yes	dst = rotate right opB over imm places.
rol_	(dst, imm, opB)	1	yes	dst = rotate left opB over imm places.

UNARY FUNCTION UNIT INSTRUCTIONS

mnem.	operands	cycl.	size.	function
abs_	(dst, opA)	1	yes	dst = absolute value of opA
sign_	(dst, opA)	1	yes	dst = sign of opA (1,0,-1)
notzero_	(dst, opA)	1	yes	dst = if A=0: '0000' else 'FFFF'
swap_	(dst, opA)	1	yes	dst = swap bits of opA
unary_	(dst, opA)	1	yes	dst = exp2(opA) - 1 (opA = 0 -> 0)
binary_	(dst, opA)	1	yes	dst = log2(opA) + 1 (opA = 0 -> 0)

ZERO / SIGN EXTENSION INSTRUCTIONS

mnem.	operands	cycl.	size.	function
zextbyte_	(dst, opB)	1	no	zero extend byte from opB to 32 bit
sextbyte_	(dst, opB)	1	no	sign extend byte from opB to 32 bit
zextshort_	(dst, opB)	1	no	zero extend short from opB to 32 bit
sextshort_	(dst, opB)	1	no	sign extend short from opB to 32 bit

MULTIPLE UNIT FUNCTIONS: BIT TEST FUNCTIONS (ix86 TYPE)

mnem.	operands	cycl.	size.	function
bt_	(opA, opB)	2	yes	dst = test bit opA of opB (set minus flag if '1')
btc_	(dst, opA, opB)	3	yes	dst = test bit opA of opB, complement bit -> dst
btr_	(dst, opA, opB)	3	yes	dst = test bit opA of opB, reset bit -> dst
bts_	(dst, opA, opB)	3	yes	dst = test bit opA of opB, set bit -> dst
bsf_	(dst, src)	2	yes	dst = 'bit scan forwards' (lowest order '1' in src)
bsr_	(dst, src)	3	yes	dst = 'bit scan reverse' (highest order '1' in src)

dst:	DESTINATION	register or control register
opA:	OPERAND A	register or immediate
opB:	OPERAND B	register or control register
reg:	REGISTER	register
imm:	IMMEDIATE	immediate value

Sizeable instructions operate on double 16 bit and quadruple 8 bit data:
double 16 bit example: **abs_16** (2x16, dst, opA)
quadruple 8 bit example: **mulx_8** (4x8, dst, opA, opB, mtype)

THE RISC/CISC LEVEL INSTRUCTIONS (4)

MULTIPLIER INSTRUCTIONS

mnem.	operands	cycl.	size.	function
mul_	(dst, opA, opB)	1	yes	dst = opA * opB (signed)
umul_	(dst, opA, opB)	1	yes	dst = opA * opB (unsigned)
mulx_	(dst, opA, opB,mtyp)	1	yes	dst = opA * opB (any of 16 types of multiplication)

DIVIDE INSTRUCTIONS

mnem.	operands	cycl.	size.	function
div_	(dst, opA, opB)	-	yes	dst = opA / opB (signed)
udiv_	(dst, opA, opB)	-	yes	dst = opA / opB (unsigned)
mod_	(dst, opA, opB)	-	yes	dst = opA / opB (signed)
umod_	(dst, opA, opB)	-	yes	dst = opA / opB (unsigned)
divmod_	(dst, opA, opB)	-	yes	dst = opA / opB (signed)
udivmod_	(dst, opA, opB)	-	yes	dst = opA / opB (unsigned)

MISCELLANEOUS INSTRUCTIONS

mnem.	operands	cycl.	size.	function
swapbyte_	(dst,src,s0,s1,s2,s3)	2	no	swap bytes: dst(0)=src(s0), dst(1)=src(s1),...dst(3)=src(s3)
logic_	(dst,reg,opA, opB)		yes	
logic3_	(dst,reg,opA,opB,op3)		yes	

IEEE 754 32 BIT FLOATING POINT FUNCTIONS (non-pipelined)

mnem.	operands	cycl.	function	types
int_sf	(dst, opB)	1	dst = int (opB)	float -> integer
float_sf	(dst, opB)	1	dst = float (opB)	integer -> float
neg_sf	(dst, opB)	2	dst = -opB	float = neg (float)
abs_sf	(dst, opB)	2	dst = abs(B)	float = abs (float)
add_sf	(dst, reg, opB)	9	dst = reg + opB	float = float + float
addint_sf	(dst, reg, opB)	10	dst = reg + opB	float = float + integer
sub_sf	(dst, reg, opB)	9	dst = reg - opB	float = float - float
subint_sf	(dst, reg, opB)	10	dst = reg - opB	float = float - integer
rsubint_sf	(dst, reg, opB)	10	dst = reg - opB	float = integer - float
add3_sf	(dst, reg, op2, op3)	11	dst = reg + op2 + op3	float = float + float + float
mul_sf	(dst, reg, opB)	12	dst = reg * opB	float = float x float
mulint_sf	(dst, reg, opB)	13	dst = reg * opB	float = float x integer
mul3_sf	(dst, reg, op2, op3)	17	dst = reg * opB	float = float x float x float
div_sf	(dst, reg, opB)	26	dst = reg / opB	float = float / float
divint_sf	(dst, reg, opB)	27	dst = reg / opB	float = float / integer
rdivint_sf	(dst, reg, opB)	26	dst = reg / opB	float = integer / float

IEEE 754 32 BIT FLOATING POINT FUNCTIONS (pipelined)

matxvec4x4_sf	(dstptr, srcptr)	34	homogeneous coordinate transformation
trans4x4_sf	(dstptr, srcptr)	60	homogeneous transformation + perspective division

THE RISC/CISC LEVEL INSTRUCTIONS (5)

CONTROL FLOW INSTRUCTIONS

General format of the control flow instructions

function condition type ([label] , [opA, OpB])

function:

branch	jump pc relative
jump	jump absolute
subr	call pc relative
call	call absolute
return	return

type:

''	32 bit test
32	32 bit test
16	16 bit test
8	8 bit test
sf	32 float test

condition:

''	no condition
eq	if opA equal opB
ne	if opA not equal opB
gt	if opA greater than opB
ge	if opA greater or equal opB
lt	if opA less than opB
le	if opA less or equal opB
ugt	if unsigned opA greater than opB
uge	if unsigned opA greater or equal opB
ult	if unsigned opA less than opB
ule	if unsigned opA less equal opB
set	if bit opA of opB is '1'
res	if bit opA of opB is '0'

examples:

branch_lt_(label, opA, opB)
branch_set_(label, 3, opB)
subr_gt_16(label, opA, opB)
return_eq_(-1, opB)

THE FREE PIPELINE LEVEL INSTRUCTIONS (1)

THREE PORT REGISTER FILE INSTRUCTIONS

Basic & extended accesses:

AB = rd*sz*(opA, opB),	**wr(dst, bus)**	opA	= register or immediate value
AB = rx*sz*(opA, opB),	**wx(dst, bus)**	opB, dst	= register or control register

Indexed basic & indexed extended accesses:

AB = rd*sz*(opA, opB),	**wr(dst, bus)**	opA	= indexed register or immediate value
AB = rx*sz*(opA, opB),	**wx(dst, bus)**	opB, dst	= indexed register or control register

data size:	sz = ['' , '1x32' , '2x16' , '4x8']	bus:	A (register)	B (register)
immediate value:	+1023...-1024	bus:	D (data-bus)	V (image-bus)
register:	r0...r63 (r63=noop)	bus:	M (multiplier)	F (ALU)
control register:	cr0..cr63	bus:	Q (barrel shifter)	U (UFU)

Immediate load accesses:

wr*msk*(dst,imm16)	dst	= register or control register

immediate 16: 0...65536
byte write mask: wr16HL, wr16H, wr16L, wr8_3, wr8_2, wr8_1, wr8_0, wr8_321, wr8_210, wr8_30 etc.

BARREL SHIFT / ROTATE UNIT INSTRUCTIONS

all functions executed at a single cycle throughput
modes: all functions operate on 4x8 bit, 2x16 bit and 32 bit

mnemonic	operation	
Q = shflog(data, A)	Q = shift logical **data** over **A** places,	Right if A is positive, Left if A is negative
Q = shfarh(data, A)	Q = shift arithmetic **data** over **A** places,	Right if A is positive, Left if A is negative
Q = rotate(data, A)	Q = shift logical **data** over **A** places,	Right if A is positive, Left if A is negative

THE FREE PIPELINE LEVEL INSTRUCTIONS (2)

ARITHMETIC AND LOGIC UNIT INSTRUCTIONS
all functions executed at a single cycle throughput
modes: all functions operate on 4x8 bit, 2x16 bit and 32 bit

mnemonic	operation	mnemonic	operation
F = clear	F = all bits '0'	**F = decr (R)**	F = R - 1
F = and (R,S)	F = R and S	**F = incr (R)**	F = R + 1
F = andrev (R,S)	F = (not R) and S	**F = decr (S)**	F = S - 1
F = copy (S)	F = S	**F = incr (S)**	F = S + 1
F = andinv (R,S)	F = R and (not S)	**F = subdecr (R,S)**	F = R - S - 1
F = noop (R)	F = R	**F = sub (R,S)**	F = R - S
F = xor (R,S)	F = R xor S	**F = subbor (R,S)**	F = R - S + carry
F = or (R,S)	F = R or S	**F = minus (S)**	F = - S
F = nor (R,S)	F = not (R or S)	**F = subdecr (R,S)**	F = S - R - 1
F = equiv (R,S)	F = R xnor S	**F = sub (R,S)**	F = S - R
F = invert (R)	F = not R	**F = subbor (R,S)**	F = S - R + carry
F = orrev (R,S)	F = (not R) or S	**F = minus (R)**	F = - R
F = copyinv (S)	F = not S	**F = add (R,S)**	F = R + S
F = orinv (R,S)	F = R or (not S)	**F = addincr (R,S)**	F = R + S + 1
F = nand (R,S)	F = not (R and S)	**F = addcar (R,S)**	F = R + S + carry
F = set	F = all bits '1'	**F = logic (R,S)**	F = three op logic function

Operand **R** can be selected from busses: **A**(register), **D**(data-memory), **M**(multiplier) and **Q**(barrel-shifter)
Operand **S** can be selected from busses: **B**(register), **V**(image-memory), **F**(ALU) and **U**(unary function unit)

UNARY FUNCTION UNIT INSTRUCTIONS
all functions executed at a single cycle throughput

mnemonic	operation	modes
U = pass (A)	pass A to U	4x8 bit, 2x16 bit, 32 bit
U = unary (A)	binary to unary conversion	4x8 bit, 2x16 bit, 32 bit
U = binary (A)	unary to binary conversion	4x8 bit, 2x16 bit, 32 bit
U = integer (Ad)	float to integer conversion	32 bit
U = fixed (Ad)	float to integer conv. variable offset	32 bit
U = float (Ad)	integer to float conversion	32 bit
U = floatFd (Ad)	integer to float conv. variable offset	32 bit
U = abs (X)	absolute value	4x8 bit, 2x16 bit, 32 bit
U = sign (X)	sign function	4x8 bit, 2x16 bit, 32 bit
U = notzero (X)	non zero function	4x8 bit, 2x16 bit, 32 bit
U = swap (X)	swap bits function	4x8 bit, 2x16 bit, 32 bit

Operand **X** can be selected from buses: **A**(register) and **F**(ALU)

THE FREE PIPELINE LEVEL INSTRUCTIONS (3)

MULTIPLIER / ACCUMULATOR INSTRUCTIONS
all functions executed at a single cycle throughput

mnemonic

M = mult (Ma, Mb, option) basic multiply operation (48 combinations)

option	data type	signs	modes
iuu	integer,	unsigned x unsigned	4x8 bit, 2x16 bit, 32 bit
ius	integer,	unsigned x signed	4x8 bit, 2x16 bit, 32 bit
isu	integer,	signed x unsigned	4x8 bit, 2x16 bit, 32 bit
iss	integer,	signed x signed	4x8 bit, 2x16 bit, 32 bit
nuu	normalised fixed point	unsigned x unsigned	4x8 bit, 2x16 bit, 32 bit
nus	normalised fixed point	unsigned x signed	4x8 bit, 2x16 bit, 32 bit
nsu	normalised fixed point	signed x unsigned	4x8 bit, 2x16 bit, 32 bit
nss	normalised fixed point	signed x signed	4x8 bit, 2x16 bit, 32 bit
fuu	fixed point	unsigned x unsigned	4x8 bit, 2x16 bit, 32 bit
fus	fixed point	unsigned x signed	4x8 bit, 2x16 bit, 32 bit
fsu	fixed point	signed x unsigned	4x8 bit, 2x16 bit, 32 bit
fss	fixed point	signed x signed	4x8 bit, 2x16 bit, 32 bit
guu	rounded norm. fixed point	unsigned x unsigned	4x8 bit, 2x16 bit, 32 bit
gus	rounded norm. fixed point	unsigned x signed	4x8 bit, 2x16 bit, 32 bit
gsu	rounded norm. fixed point	signed x unsigned	4x8 bit, 2x16 bit, 32 bit
gss	rounded norm. fixed point	signed x signed	4x8 bit, 2x16 bit, 32 bit

8 bit array operations 16 multiplies & 12 adds per cycle:

M = inproduct (Mb)	quadruple vector inproduct,	4x8 bit, all data types, signed/unsigned
M = matrixvec (Mb)	4 x 4 matrix times vector multiply	4x8 bit, all data types, signed/unsigned
M = loadpipe (Ma, Mb)	load data & coefficient pipe line	4x8 bit, all data types, signed/unsigned

accumulator ram access:

M = read_ram ()	read 96 bit (multi) word from the accumulator ram
M = write_ram ()	write 96 bit (multi) word to the accumulator ram

incremental functions for differential engine applications:

M = linearstep ()	incremental add	4x12 bit, 4x24 bit, 2x48 bit, 1x72 bit

general multiply accumulate functions:

M = macs (Ma, Mb)	multiply accumulate scalar	all 48 basic multiply options
M = macb (Ma, Mb)	multiply accumulate vector	all 48 basic multiply options

multiple 16 bit functions: 4 multiplies and 2 additions per cycle:

M = vectprod (Ma, Mb)	16 bit vector dot and cross products	all data types, signed
M = complex (Ma, Mb)	16 bit complex products	all data types, signed

Operand **Ma** can be selected from : **A**(register), **D**(data-memory), **M**(multiplier) and **Q**(barrel-shifter)
Operand **Mb** can be selected from : **B**(register), **V**(image-memory), **F**(ALU) and **U**(unary function unit)

Chapter

4. THE REGISTER FILE

The register file plays a central role in the processing philosophy of the Imagine. It contains 120 general purpose 32 bit registers and a Vector register file of 256 words of 32 bit with a wide range of special purpose options. This unit also serves as a port to a large number of control registers spread throughout the core processor.

120 General Purpose Registers:
For C type code and RISC assembly code. A 'Register plus Register to Register mode' and an 'Immediate plus Register to Register mode' are available. The latter supports operations with constants and the C stack relative addressing to local variables.

256 Vector Registers:
The 256 word Vector register file which is accessible with the Vector Index generators, enables the implementation of a large number of algorithms which are by nature less suitable for classic SIMD processors. It allows various forms of parallel conditional processing by means of conditional data flow mechanisms instead of conditional control flow. It fully supports the three basic data types of the Imagine: single 32 bit, double 16 and quadruple 8 bit words. It can generate addresses for all these sizes conditionally by using status information from the Status Register or Range Check Unit. This means for 8 bit words that it can generate 12 different register file addresses in each cycle: eight to read data and four to write data. The Vector Index generator supports besides conditional address generation also conditional byte write enabling, byte preset and byte reset.

128 Control Register entries:
The Imagine core processor, The 3D graphics unit and The Mask Generator contain many control registers which can be accessed in much the same way as the 120 general purpose register. The reads and writes to these control registers use two separated busses. A control register can be read and one can be written each cycle.

Overview of the Register file

4 THE REGISTER FILE

4.1 introduction

The Register File is a 120 + 256 word RAM with two read ports (A and B) and one write port (C). Two Read actions and one Write action can be performed each cycle. 120 entries are directly accessible (entry 0 through 119). The instruction code has three 7 bit address fields for these entries. References to entries 120 to 126 have a special meaning. Entry number 127 is interpreted as a no-op. The 256 vector registers are accessible via indices generated by the Vector index generator. The data read from the register file is placed in the A and B registers which are readable by the other data processing and I/O units. All eight internal buses can be selected as the source of the data to be written into the Register File. A read from a register which is written to in the same cycle loads the new value directly in the A or B register, bypassing the RAM.

4.1.1 the control registers

The address fields for the register file may be used alternatively to access the internal control registers which accompany many of the individual functional units. One control register can be read and one can be written each cycle. The B port is used to read and the C port is used to write to a control register. A total of 128 different control register addresses is available. Two Instruction code bits (B and C) differentiate between normal and control register accesses. The A port can load an 11 bit immediate value instead of a register value.

A port: read from *register* or *11 bit immediate* (range: -1024...+1023)
B port: read from *register* or *control register*
C port: write to *register* or *control register*

4.1.2 the vector index generators

The Vector Index Generator provides addresses (indices) to access the 256 word vector register file. It can generate individual indices for 8 bit or 16 bit components of the 32 bit words. Up to twelve different indices can be generated each cycle: four for the four bytes of port A, four for the four bytes of port B and four for the four bytes which are written via port C. The generator can use 8 bit data offsets for run time generated register address. It can generate Write Enables which control writing of individual bytes. The indices and write enables can be generated conditionally based on status information from the ALU and the Range control unit. This type of parallel conditional processing is used to perform various graphics and image processing algorithms at a very high sustained speed.

4.1.3 the access modes

The Register file has four main access modes:

♦ **register plus register to register**
A general or vector register is loaded in the A port register. A general, vector or control register is loaded in the B port register and any of the 8 internal buses is written into a general, vector or control register.

♦ **immediate plus register to register:**
An 11 bit sign extended value from the instruction word is loaded in the A register. A general or vector register is loaded in the B register and any of the 8 internal buses is written into a general or vector register.

♦ **16 bit constant load:**
16 bit constants can be written into any bytes of a general, vector or control register. The 16 bit is placed on both the highest and lowest 16 bit of the 32 bit control register bus while four byte write enables control the writes to the individual bytes.

♦ **The 32 bit constant load / merge:**
This mode loads a 32 bit constant directly into a general, vector or control register. The merge function allows bit field insertion by rotating a value on the A bus (immediate or register value) to the right bit position and then merge the selected bits (indicated via the 32 bit constant value) together with the value on the B bus (register or control register). Four byte write enables in the instruction control writes to individual bytes.

4.2 The control registers

One of the basic principles of the programming model of the Imagine 2 is to allow extra functionality with the use of control registers. The basic instructions do not use these registers and are completely defined by the instruction word without any side effects. Other instructions with extra capabilities do refer to control registers to see what the exact function should be. The registers above are used for this extra functionality.

cr0: REG_Control: The Register File Control register

<table>
<tr><td>‘0’</td><td colspan="3">ISIZEA
[2:0]</td><td>‘0’</td><td colspan="3">ISIZEB
[2:0]</td><td>‘0’</td><td colspan="3">ISIZEC
[2:0]</td><td>‘0’</td><td colspan="3">FIFOPTR.
[2:0]</td><td colspan="4">‘0000’</td><td colspan="4">W3 W2 W1 W0
(Write enables)</td><td>SE</td><td colspan="2">‘00’</td><td colspan="2">Write.
FUNCT</td><td>‘0’</td><td colspan="2">VIPTR.
[2:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr1: REG_Monitor: Data Size Monitor Register

<table>
<tr><td colspan="2">ASIZE
[1:0]</td><td colspan="2">BSIZE
[1:0]</td><td colspan="2">QSIZE
[1:0]</td><td colspan="2">FSIZE
[1:0]</td><td colspan="2">MSIZE
[1:0]</td><td colspan="2">USIZE
[1:0]</td><td colspan="2">DSIZE
[1:0]</td><td colspan="2">VSIZE
[1:0]</td><td colspan="11">‘00000000000’</td><td colspan="2">AxSIZE
[1:0]</td><td>‘0’</td><td colspan="2">BxSIZE
[1:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr2: REG_Vector: Vector Index control register

<table>
<tr><td colspan="2">‘00’</td><td colspan="2">BASEA
[1:0]</td><td colspan="4">OFFSETA
[2:0]</td><td colspan="2">‘00’</td><td colspan="2">BASEB
[1:0]</td><td colspan="4">OFFSETB
[2:0]</td><td colspan="2">‘00’</td><td colspan="2">BASEC
[1:0]</td><td colspan="4">OFFSETC
[2:0]</td><td colspan="2">SELBUS.
[1:0]</td><td colspan="3">SELSTA.
[2:0]</td><td colspan="3">DW IW EW</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr3: REG_Fifo: Write delay fifo entry , **REG_Indices:** Vector Indices entry

<table>
<tr><td colspan="32">Access to data from the Write Delay Fifo or Vector Indices A, B, C and flags
[31:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr4: REG_A_Indices: Vector indices for Read port A

<table>
<tr><td colspan="8">A3 INDEX
[7:0]</td><td colspan="8">A2 INDEX
[7:0]</td><td colspan="8">A1 INDEX
[7:0]</td><td colspan="8">A0 INDEX
[7:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr5: REG_B_Indices: Vector indices for Read port B

<table>
<tr><td colspan="8">B3 INDEX
[7:0]</td><td colspan="8">B2 INDEX
[7:0]</td><td colspan="8">B1 INDEX
[7:0]</td><td colspan="8">B0 INDEX
[7:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr6: REG_C_Indices: Vector indices for Write port C

<table>
<tr><td colspan="8">C3 INDEX
[7:0]</td><td colspan="8">C2 INDEX
[7:0]</td><td colspan="8">C1 INDEX
[7:0]</td><td colspan="8">C0 INDEX
[7:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr7: REG_C_Flags: Byte Write Enables, Presets, Resets and delayed status flags

<table>
<tr><td colspan="2">PRE 3
[1:0]</td><td colspan="2">RES 3
[1:0]</td><td>P3</td><td>R3</td><td>S3</td><td>W3</td><td colspan="2">PRE 2
[1:0]</td><td colspan="2">RES 2
[1:0]</td><td>P2</td><td>R2</td><td>S2</td><td>W2</td><td colspan="2">PRE 1
[1:0]</td><td colspan="2">RES 1
[1:0]</td><td>P1</td><td>R1</td><td>S1</td><td>W1</td><td colspan="2">PRE 0
[1:0]</td><td colspan="2">RES 0
[1:0]</td><td>P0</td><td>R0</td><td>S0</td><td>W0</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

The vector index generator has the capability to calculate up to 12 different 8 bit indices which are used to access individual bytes of the 256 word vector register file. These indices are visible in control registers cr4 to cr6. The Vector Index generator control register (cr2) contains the definition of the index generation. Besides indices it also can generate four byte write enables, visible in cr7: flags W[3:0], which can be used to disable the writing of individual bytes into the vector register file. In many cases we want to use the data itself for the index generation or the byte write enables. An input fifo on the write port can temporary delay data from 1 to 8 cycles before being written to vector register file during the time it takes to calculate indices and/or byte write enables. The
extended access also provides on the fly operations on data which is being written into a general, vector or control register and a function which allows the use of run time programmable data sizes for the A port and the B port. The data size monitor can save and restore the data sizes of the 8 buses during interrupts.

4.2 Register plus register to register mode:

- The A port is read from a general register or a vector register
- The B port is read from a general register, a vector register or a control register
- The value of one of the 8 internal buses is written to a general register, a vector register or a control register.

4.2.1 accesses to general purpose and control registers

The A port and B port registers are available for other functional units in the next cycle. The two ports have 2 bit size information besides the 32 data bits. Two bits in the instruction field define the size of the A and B read data. The size information of the data selected by the write port is not stored into the register file. The register file does not keep track of sizes but adds them during read operations.

Some examples of read operations:

```
A    = rd( r17);           B    = rd( cr121);
AB = rd( r37, cr53);       AB = rd4x8( r47, r48);
B    = rd2x16( r104);      B    = rd1x32( r57);
```

Register file read port size

0: size is 4x8	(quad_byte)
1: size is 2x16	(double_short)
2: size is 1x31	(single_word)
3: {reserved}	

The read function contains an optional size indicator '4x8', '2x16' or '1x32'. 32 bit accesses are default when no size indication is given. The normal registers are indicated by **r0 .. r119** while the control registers are indicated by **cr0 .. cr127**. Some examples of write operation:

```
wr( r89,  M);    wr( r39,   F);    wr( r15,  D);    wr( cr124, Q);
```

The write examples select data from buses M, F, D and Q: the Multiplier result, the ALU result, the Data Input from data memory and the Barrel Shifter. All eight on chip buses can be selected as a data source for the C write port of the register file. A write to a control register may occur while another is read on the B read port because there is a separated control register read and a control register write bus.

Write port bus selection

0:	select A bus data
1:	select B bus data
2:	select Q bus data
3:	select F bus data
4:	select M bus data
5:	select U bus data
6:	select D bus data
7:	select V bus data

4.2.2 vector register accesses

The 120 General purpose registers are accessed with values from 0 to 119. From the remaining 8 options we use 127 as a no-operation instruction. The remaining addresses are used to indicate that a vector index generator is used to generate the actual register address. There are 256 vector registers which can be accessed in this way. Indices have to be generated first before they can be used in the next cycle. Examples of generate-index instructions:

```
genad(A);           genad(A, B);        genad( wr );
genad();            genad(A, B, wr);    genad();
```

Special register addresses

120	extended indexed,	generate new index
121	extended indexed,	
122	load write fifo,	generate new index
123	load write fifo	
124	indexed access,	generate new index
125	indexed access	
126	no operation,	generate new index
127	no operation	

The index addresses are calculated with the vector index generators. This unit can calculate up to twelve different addresses each cycle plus four 'byte-write-enables' for the write port.

The way in which the index is calculated is defined by the *Vector Index generator control register* and described in detail in the chapter devoted to this unit. The generate index instruction can be combined with an indexed-access, an extended-indexed access or with a load-write-fifo function (write port only) The LSB of the special address should be ‘0’ to invoke this function. Four indices are generated per port (one for each byte). Multiple generate-index instructions for more than one port can be combined into a single instruction. If the port name is omitted (empty brackets) then all three ports are affected.

Special mnemonics are use to indicate the use of the special addresses. Instead of absolute addresses like **r12** or **r48** we us '**ri**', '**xi**' or '**fifo**' Implicit generation of new indices for a specific port is possible by adding **++** to the mnemonics. The effect is the same as a separate **genad()** function. Examples of indexed accesses:

```
A   = rd ( ri);         AB = rd4x8( ri, ri); AB = rd ( xi, cr12);        wr ( ri++, V);
B   = rd1x32( ri);      wr ( xi++, B);               AB = rd4x8 ( ri, xi++);    AB=rd4x8( ri, r17);
B   = rd2x16( xi++);    wr ( fifo, B);       wr ( xi );
```

'**ri**' = register indexed-access 　'**ri++**' = register indexed-access and generate new indices
'**xi**' = extended-indexed-access 　'**xi++**' = extended-indexed-access and generate new indices
'**fifo**' = load-write-fifo

4.2.3 The extended-indexed-accesses

The normal read and write functions can be executed with a number of extra options. It is not the instruction word but the Register file control register which determines the exact function.

Three extra functions are provided:
- individual *Byte write enabling* during a write to a vector register.
- '*On the fly operations*' on data before it is written into a vector register.
- Conditional Byte Presets and Byte Resets on data before it is written into a vector register.
- Run time programmable data sizes for the A port and the B port.

The functions are not visible in the instruction mnemonics since they are not defined by the instruction itself but by a control register. The mnemonic indicates that the function is an extended function (**xi** instead of **ri**) The optional functions are described in detail in the chapter of the 'Extended functions'.

4.3 Immediate plus register to register mode

- The A port is loaded with a 11 bit sign extended value from the instruction.
- The B port is either read from a general or vector register
- The value of one of the 8 internal busses is written to a general or vector register.

This mode allows the use of an 11 bit immediate value. 11 bits in the instruction word are placed in the 11 least significant bit locations of read port A. The upper 21 bits of read port A are sign extended (identical to the 11th bit) The read port B is loaded from a general register or vector register and the write operation stores a selected value in either a general register or vector register. Examples:

```
A  = rd( 0x325),  wr( r12, F);      AB = rd( -23, r56),  wr( r27, M);      AB = rd( 912, r4),  wr( r16, U);
```

The data size used for these access modes is always 32 bit. This size will be attached to the A port and B port data. The control registers can not be accessed. Both size bits in the instruction code and control register access flags in the instruction code are freed for the 11 bit immediate value. The most important use of this access mode is in RISC like C code where it provides single cycle immediate operations as: X = 4*Y or B = A<<19 or Q = P&0x3F and in single cycle Load operations with *Stack relative* and *Base relative* addressing where a small constant value is added to the stack- or base-pointer.

4.4 The 16 bit constant load.

A 16 bit value in the instruction word is used to load 16 bit values into general, vector or control registers. The 16 bit value is placed on the highest and lowest 16 bit of the 32 bit control register write and four byte write enables in the instruction word control the writing to individual bytes. The read port A and read port B registers are not modified. They will keep their contents. The C address from the instruction word is used as the write address.

Examples of 16 bit constant loads:

```
wr (r0, 14);             wr16H( r12, 0x12345678 >> 16);   wr16L (r12, 0x12345678 & 0xFFFF);
wr8_3 (cr24,  0x84<<8);  wr8_1 ( cr0, 0x0E<<8);       wr8_2  (cr1, 0x67);
wr8_0 (cr6,   0x3A);     wr16H( ri,   0x1234);            wr16HL(ri,  0 );
```

wr16HL:	write to 16 MSB and 16 LSB bits.	**wr8_3:**	write to bits 24..31. (upper 8 bit of data)
wr16H:	write to 16 MSB bits.	**wr8_2:**	write to bits 16..23. (lower 8 bit of data)
wr16L:	write to 16 LSB bits.	**wr8_1:**	write to bits 8..15. (upper 8 bit of data)
		wr8_0:	write to bits 0..7. (lower 8 bit of data)

the 8 bit versions can be combined into any arbitrary combination of bytes like: **wr8_321, wr8_210, wr8_30** et cetera (A '1' in the write enable field allows writing. IC3=we3, IC2=we2, IC1=we1, IC0=we0)

4.5 The 32 bit constant load / merge.

This function uses the complete 64 bit instruction word for itself so the other data processing units can not be active and they will hold their current values. The only active part of the register file is the write port and optionally the write index generator. The read ports and their index generators will hold their current value.

The 32 bit load function can store a 32 bit field from the instruction word directly into a general, control or vector register. The 32 bit merge function can perform a bit field insertion by rotating the A port register which contains the field to be inserted to the right bit positions and the use the 32 bit field from the instruction word to merge the selected bits with the value from the B port register which should contain the data into which the bit field is to be inserted.

Examples of 32 bit constant loads:

```
wr32( r109, 0x76543210 );            wr32( cr2 0x25252546 );
```

Example how to insert a 5 bit value from register **r49** to position 24 of control register **cr17**:

```
AB = rd( r49,cr17);   merge( cr17,  24,  0x1F << 24 );
```

Example how to merge the highest 16 bit of register r12 with highest 16 bit of register r59. Register r12 is shifted 16 positions down before merging. The result is stored in control register cr90.

```
AB = rd( r12,  r59);  merge( cr90,  -16,  0xFFFF  );
```

4.6 Vector index generators.

4.6.1 results of the index generators

The 256 word Vector Register File can be accessed with the use of the Vector index generators. A Vector index generator is a versatile unit which has numerous ways to construct the effective addresses for the Vector register. Each byte of read port A, read port B and write port C has its own Vector index generator: a total of twelve different 8 bit indices can be generated: **XY**index. Where **X** is register port A, B or C and **Y** is byte number 0, 1, 2 or 3. The results are visible in three control registers (cr4: REG_A_Indices, cr5: REG_B_ Indices, cr6: REG_C_Indices). A fourth control register (cr7: REG_C_Flags) contains 4 generated byte write enables for the write port, four byte Presets and four byte Resets.

4.6.2 control input for the Index generators

The 12 vector indices generators for the read ports A and B and write port C are controlled by three parameters per port plus two parameters for all three ports:

baseA[1:0], baseB[1:0], baseC[1:0] :	Select the A/B/C vector bases	(defined in cr2)
offsetA[2:0], offsetB[2:0], offsetC[2:0] :	Select the A/B/C vector offsets	(defined in cr2)
IsizeA[2:0], IsizeB[2:0], IsizeC[2:0] :	Size of the A/B/C indices (number of bits-1)	(defined in cr0)
selbus[1:0] :	Select 1 of 4 busses for the offset value	(defined in cr2)
selsta[2:0] :	Select 1 of 8 status flags for cond. operations	(defined in cr2)

The byte write enable generation uses the 4 selected status flags. The operation is controlled with two flags:

eW : Enable the generation (Byte Write enables = status)
iW : invert the status values when used as Byte Write enables
dW : use the delayed status flag for Write enables, Presets and Resets

4.6.3 data input for the Index generators

data and status which can be used for the index generation: data**Y**[7:0] :four bytes data selected from the A, B, M or V bus and status**Y**: four 1 bit status flags from the ALU_RC_Status register (cr15)

4.6.4 Index generator calculations

The calculation of the indices is defined by the base**X** [1:0] and offset**X** [2:0] fields. Together they construct the **XY**index [7:0] value from the current **XY**index value plus the selected input data **Y** [7:0] and status**Y** flag. The write port C can also use the current index of read port A for Read-modify-Write type operations. The Isize**X** [2:0] value defines the number of LSB bits from **XY**index [7:0] which can be modified. The remaining higher bits are write protected. The value 0 enables modifying bit 0 while 7 enables the modification of [7:0]

The functionality provided by BASE**X:** (**X** = register port A, B or C, **Y** = byte number 0, 1, 2 or 3)

0:	BASE_0	base **XY** = 0
2:	BASE_CURRENT_ADDR	base **XY** = **XY**INDEX (The current index is used as base)
3:	BASE_READ_A_ADDR	base **XY** = A**Y**INDEX (The read port A index used as base)

The functionality provided by OFFSET**X:** **X** = register port A, B or C, **Y** = byte number 0, 1, 2 or 3)

0:	NO_OFFSET	**XY**INDEX = base **XY** + 0
1:	ADD_1	**XY**INDEX = base **XY** + 1
2:	ADD_OFFSET	**XY**INDEX = base **XY** + data **Y**
3:	ADD_OFFSET_ADD_1	**XY**INDEX = base **XY** + data **Y** + 1
4:	ADD_1_IF_TRUE	**XY**INDEX = base **XY** + (status **Y** ? 1 : 0)
5:	ADD_1_IF_FALSE	**XY**INDEX = base **XY** + (status **Y** ? 0 : 1)
6:	ADD_OFFSET_IF_TRUE	**XY**INDEX = base **XY** + (status **Y** ? data **Y** : 0)
7:	ADD_OFFSET_IF_FALSE	**XY**INDEX = base **XY** + (status **Y** ? 0 : data **Y**)
12:	ADD_1_IF_OLD_TRUE	**XY**INDEX = base **XY** + (delayed status **Y** ? 1 : 0)
13:	ADD_1_IF_OLD_FALSE	**XY**INDEX = base **XY** + (delayed status **Y** ? 0 : 1)
14:	ADD_OFFSET_IF_OLD_TRUE	**XY**INDEX = base **XY** + (delayed status **Y** ? data **Y** : 0)
15:	ADD_OFFSET_IF_OLD_FALSE	**XY**INDEX = base **XY** + (delayed status **Y** ? 0 : data **Y**)

4.6.5 select the data bus used for the offset data

8 bit offsets can be taken from the A bus, B bus, M bus or the V bus with the SELBUS[1:0] field from the vector index generation control register. The actual offset data depends on the size of the chosen bus. When the bus size indicates single 32 bit word data size then the lowest 8 bits of the selected bus are used for all four indices. So a port will have identical offsets for each of its four individual bytes.

SELBUS[1:0] Select the bus for the offset

0:	B_BUS_OFFSET:	select the B bus
1:	M_BUS_OFFSET:	select the M bus
2:	A_BUS_OFFSET:	select the A bus
3:	V_BUS_OFFSET:	select the V bus

If the data size is 4x8 (quadruple 8 bit data), then the 8 bits are used from each individual byte of the 32 bit wide selected bus data. In this case all four offsets can be different for each of a ports for bytes. In case of a double 16 bit word data size, bits [7:0] are used for the lowest two of the four byte address generators, and bits [23:16] are used for the highest two. So a 16 bit word will have identical offsets for both it's bytes.

Select offset data depending on the bus size

	size=4x8	size=2x16	size=1x32
data **0** =	bit [8:0]	bit [7:0]	bit [7:0]
data **1** =	bit [15:8]	bit [7:0]	bit [7:0]
data **2** =	bit [23:16]	bit [23:16]	bit [7:0]
data **3** =	bit [31:24]	bit [23:16]	bit [7:0]

4.6.6 select the status for conditional index generation and byte write enables

The status information which is generated by the ALU or the Range unit and saved in the ALU_RC_Status register, (cr15) can be used for conditional index generation for the read ports A and B and write port C. Conditional index generation is discussed in the previous paragraphs. It is also used for conditionally writing to a vector register.

SELSTA [2:0] function

0:	STATUS_ZERO	status = ALU_RC_Status [24, 16, 8, 0]
1:	STATUS_MINUS	status = ALU_RC_Status [25, 17, 9, 1]
2:	STATUS_CARRY	status = ALU_RC_Status [26, 18, 10, 2]
3:	STATUS_SGNCMP	status = ALU_RC_Status [27, 19, 11, 3]
4:	STATUS_INSIDE	status = ALU_RC_Status [28, 20, 12, 4]
5:	STATUS_HIGHER	status = ALU_RC_Status [29, 21, 13, 5]
6:	STATUS_LOWER	status = ALU_RC_Status [30, 22, 14, 6]
7:	STATUS_WRONG	status = ALU_RC_Status [31, 23, 15, 7]

Conditional writing can be combined with the four *byte write enables* which can be used possible during extended register access instructions. If both are used together then a byte can only be written if both the *extended-byte-write-enable* is true ('1') (control register 0, bits [11:8]) and the *conditional-byte-write-enable* is true ('1'). The four conditional status bits are depending on the data size used during the operation which has generated them in the ALU or the Range unit. If the data type was quadruple 8 bit byte, then all four conditional enables are different because they were generated in four byte operations.
If the data type was 'double 16 bit short' then the upper and lower pair are identical because they stem from double 16 bit word operations. In the single 32 bit word operation all four conditional enables are identical.

4.6.7 select between the use of the current or delayed status

The read port A index generator result can be used by the write port C index generator for a number of Read-modify-Write operations. The read port A index generator will also save the selected status flags from the status register (cr15: ALU_RC_Status) into four flags of control register cr7 (REG_C_Flags). This means that the status flags can be re-used in the next cycle by another index generator for a conditional function. The use of the delayed status instead of the current status in the status register can be controlled individually for each index generator and for the generation of byte write enables, presets and resets

OFFSETA[3]: Select the delayed status for the read port A index generation.
OFFSETB[3]: Select the delayed status for the read port B index generation.
OFFSETC[3]: Select the delayed status for the write port C index generation.
DW: Select the delayed status for the write port C byte write enables, presets and resets generation.

4.6.8 generation of the byte write enables

The four selected status flags can be used as four byte write enables. A logic ‘1’ enables writing. The status bits can be optionally inverted before being used as write enables. Three flags in the vector index generation control register define the operation: EW, IW and DW.

EW: enable status for write enable
'0' Do not use the status flag. The byte write enables are set to ‘1’ (enabled)
'1' The selected status flags are used as a conditional byte write enable for the C port.

IW: invert status for write enable
'0' Do not invert the status: write enable if the status is true ('1').
'1' Invert the status: write enable if status is false ('0').
DW: invert status for write enable
'0' Use the current status from the ALU_RC_Status register, selected with the SELSTA[2:0] field from the REG_Vector control register.
'1' Use the four (delayed) Status flags from the REG_C_Flags register which are loaded by a genad(A) instruction from the ALU_RC_Status register, selected with the SELSTA[2:0] field from the REG_Vector control register.

4.6.9 generation of the byte presets and byte resets

Four Byte preset flags and Four Byte reset flags are generated by the four write port index generator. These flags can be used by extended indexed write accesses. E.g.: **wr(xi, F)** Normal indexed accesses do not use these flags. The preset flag will set the byte to ‘11111111’ while the reset flag will set the byte to ‘00000000’ The generation is controlled for each flag individually by a 2 bit field in control register cr7: REG_C_Flags. The PRE**Y**[1:0] fields generate the preset flags while the RES**Y**[1:0] fields generate the reset flags. (**Y** indicates the byte: 0...3)

Preset Flag generation options:		**Reset Flag generation options:**	
0	PRESET_NEVER	0	RESET_NEVER
1	PRESET_IF_TRUE	1	RESET_IF_TRUE
2	PRESET_IF_FALSE	2	RESET_IF_FALSE
3	PRESET_ALWAYS	3	RESET_ALWAYS

4.7 The extended functions.

The extended functions are defined by the Imagine instruction *plus* the contents of the control registers. Four extra functions can be added and mixed in any combination: Byte write enables, On the fly operations on write data operations, Byte presets / resets and Run time programmable bus sizes for the A bus and B bus.

4.7.1 byte write enables

The byte write enable option allows the writing of individual bytes to a vector register:
cr0 bit[8]: W0: if 1 { enable write to bits [7: 0] } else { disable write to bits [7: 0] }
cr0 bit[9]: W1: if 1 { enable write to bits [15: 8] } else { disable write to bits [15: 8] }
cr0 bit[10]: W2: if 1 { enable write to bits [23:16] } else { disable write to bits [23:16] }
cr0 bit[11]: W3: if 1 { enable write to bits [31:24] } else { disable write to bits [31:24] }

4.7.2 On the fly write Functions

Three simple 'On the fly operations' can be performed on the data before it is written into the register file. A two bit field in control register cr0 selects the function to use for the operation.

On the fly function: cr0 [4:3]

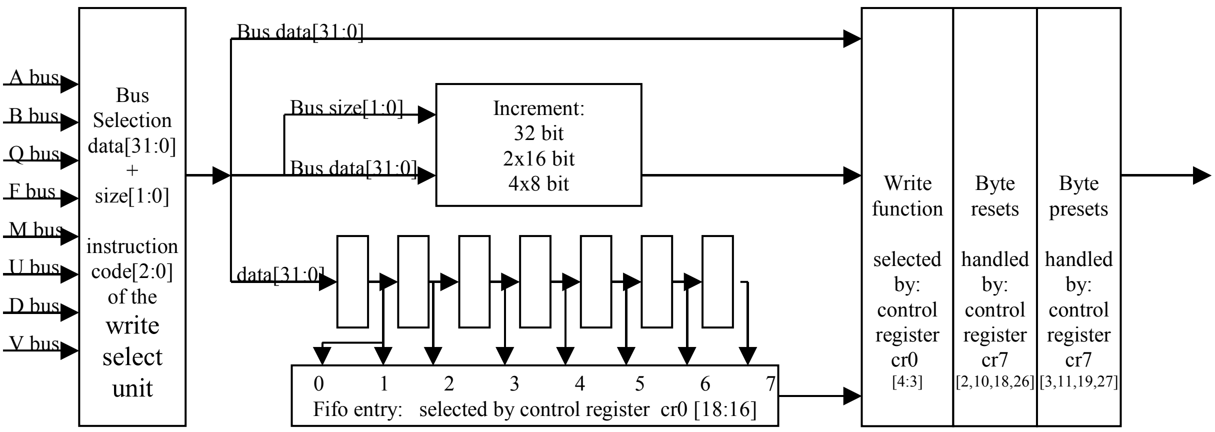
0: DATA_PASS
1: DATA_INCREMENT.
2: DATA_DELAY

.Function 0: NO-OPERATION
This function does not alter the data which is to be written into the selected register. Use can use the four byte write enables in case you want to write part of the entire word only.

Function 1: WRITE DATA INCREMENT
The use of this function is in counting functions. After an arbitrary classification a table entry should be incremented. If a classification is made each cycle for a number of cycles in a row, then the same entry might be used in two consecutive cycles: in which case the result of the first cycle is needed as the operand of the second. The on the fly increment provides this function. Increments are performed on bytes, 16 bit words or the complete 32 word depending on the data size of the selected bus

Function 2: WRITE DELAY FIFO
This function is used to delay the data in functions where the data itself determines in some way the vector index generation or write enables. The information from the vector index generators and the Data itself should arrive at the same time. This function can be used to introduce extra pipeline delay. The internal 32 bit wide and 7 stages deep shift register used for the delay is loaded by a *load fifo instruction*: e.g.: **wr(fifo, Q)** this function also increments the Fifo pointer (found in cr0: REG_Control) A value from the Fifo is read and written into a vector register (and the Fifo pointer is decremented) with the use of an *extended write instruction*. e.g.: **wr(xi)**. The 3 bit Fifo pointer has values from 0 through 7. A value of zero indicates that the Fifo is empty, reading the fifo will repeat the last read value. A value of N = 1..7 means that the Fifo is filled with N values. (the value 7 is not incremented, and the value 0 is not decremented)

Overview of the extended write data functions


Extended Functions
A bus
B bus
Q bus
F bus
M bus
U bus
D bus
V bus
Bus Selection data[31:0] + size[1:0]
instruction code[2:0] of the write select unit
Bus data[31:0]
Bus size[1:0]
Bus data[31:0]
Increment: 32 bit 2x16 bit 4x8 bit
data[31:0]
0 1 2 3 4 5 6 7
Fifo entry: selected by control register cr0 [18:16]
Write function selected by: control register cr0 [4:3]
Byte resets handled by: control register cr7 [2,10,18,26]
Byte presets handled by: control register cr7 [3,11,19,27]

4.7.3 application of the byte presets and byte resets

The byte preset and reset flags generated by the write port index generators and available in the REG_C_Flags control register (cr7) are applied during extended indexed writes. These writes use the ‘**xi**’ mnemonic like in wr(**xi**, F)

4.7.4 run time programmable data sizes

This is an extended read function for the read ports A and B: The data sizes are taken by default from the instruction code. The data sizes can also be taken from the two fields AxSIZE and BxSIZE in control register REG_Monitor (cr0), The SE flag (Size enable) in REG_Control [7] should be '1'

4.7.5 preserved for compatibility only

The following operations are provided for Imagine 1 compatibility only and will be **discontinued** in future versions of the IMAGINE.

-Extended accesses to the general purpose registers and control registers:

The extended functions (only those of the Imagine 1) can also be applied to the general registers and the control registers. A single bit in the instruction code changes the operation of all three ports to extended mode. The normal register and control register values can be used. This option is supplied for reasons of compatibility only and should not be used for any new code and replaced in old code with for instance the 32 bit load / merge function. The mnemonic used to indicate that the Three Port register file operates in extended mode are **rx** and **wx** instead of **rd** and **wr**:

Examples: **AB = rx(r12, r15), wx(r50, F); AB = rx(r12, cr12), wx(r110, B);**

-Indirect accesses to the Index pointer control registers:

The 4 control registers cr4: REG_A_Indices, cr5: REG_B_Indices, cr6: REG_C_Indices, cr7: REG_C_Flags are also indirectly accessible via control register cr3. The Fifo pointer (cr0: REG_Control[18:16]) should be 0 otherwise one of the Fifo registers is accessed. The VIPTR[1:0] selects between the for control registers. This field is post-incremented after a control register read or write access (but only if the Fifo pointer is 0)

-Moving sizes to the A or B bus:

The sizes can be added to the A bus and B bus data during extended read operations by setting SE to '1', SO to '1' and using the two select fields as 3 bit pointers into the array of bus sizes (A=0, B=1, Q=2, F=3, M=4, U=5, D=6, V=7).

- 1 Cycle Write delay register

The extended write function with the old 1 cycle On-the-fly-delay function still works but only with the old **wx(ri)** function, the new **wr(xi)** activates the full Write Delay FIFO function: The difference is that **wx(ri)** shifts the 7 stage Fifo registers and does not decrement the Fifo pointer while **wr(xi)** does not move the Fifo registers and does decrement the Fifo, The function **wx(ri)** should not be used for any new functions and replaced in old code.

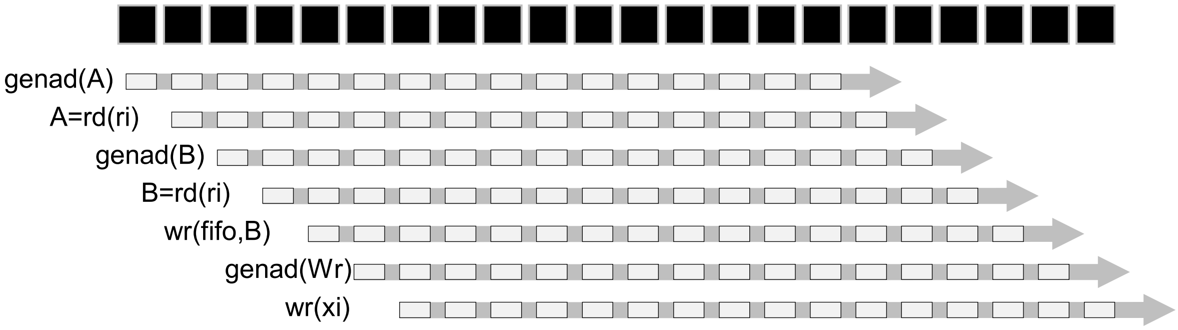
4.8 The 7 independent sub units of the register file

The register file consist of seven independent units. Each of the units has it's own copy of the instruction code register (the instruction bits which it needs) so the repeat functions can be executed independently for each of the seven units. The seven units are:

3 access ports: read port A, read port B, write port C
3 vector index generators for the three ports
1 write input bus selector and write FIFO control unit

The following example shows how all units repeat their own function N times. Where N is 16 in this example.

repeat, graph (example_graph)
;;
example_graph:
genad(A) => A= rd(ri) => genad(B) => B= rd(ri) => wr(fifo, B) => genad(Wr) => wr(xi);



4.8.1 read port A span of control

Read Port A controls the modification of the A bus[31:0] register plus the corresponding A size[1:0]

4.8.2 read port B span of control

Read Port B controls the modification of the B bus[31:0] register plus the corresponding B size[1:0]

Some Control register reads can modify other control register's state E.g.: auto increment reads may cause a pointer to be incremented. The Register File control registers have two such pointers: REG_Control [FIFOPTR], cr0[18:16] and REG_Control [VIPTR] , cr0[1:0] Both pointers can be incremented by reads from control registers REG_Fifo, cr3.

4.8.3 write port C span of control

Write Port C controls the modification of the General purpose registers, the Vector registers and the control registers.

Some Control register writes can modify other control register's state E.g.: auto increment writes may cause a pointer to be incremented. The Register File control registers have two such pointers: REG_Control [FIFOPTR], cr0[18:16] and REG_Control [VIPTR] , cr0[1:0] Both pointers can be incremented by writes to control registers REG_Fifo, cr3. REG_Control [FIFOPTR] can also be decremented by an extended indexed write to a vector register with data read from the Delay FIFO: wr(**xi**) Finally a write of a '1' to the monitor enable flag: REG_Monitor [ME], cr1[7] will load the 8 bus sizes in highest 16 bit of the same control register.

4.8.4 read port A index generator span of control

This unit generates the contents of REG_A_Indices, cr4 plus the four delayed status flags in control register REG_C_Flags[S3, S2, S1, S0], cr7[25,17,9,1]

4.8.5 read port B index generator span of control

This unit generates the contents of REG_B_Indices, cr5.

4.8.6 write port C index generator span of control

This unit generates the contents of REG_C_Indices, cr6 plus 12 flags in REG_C_Flags:

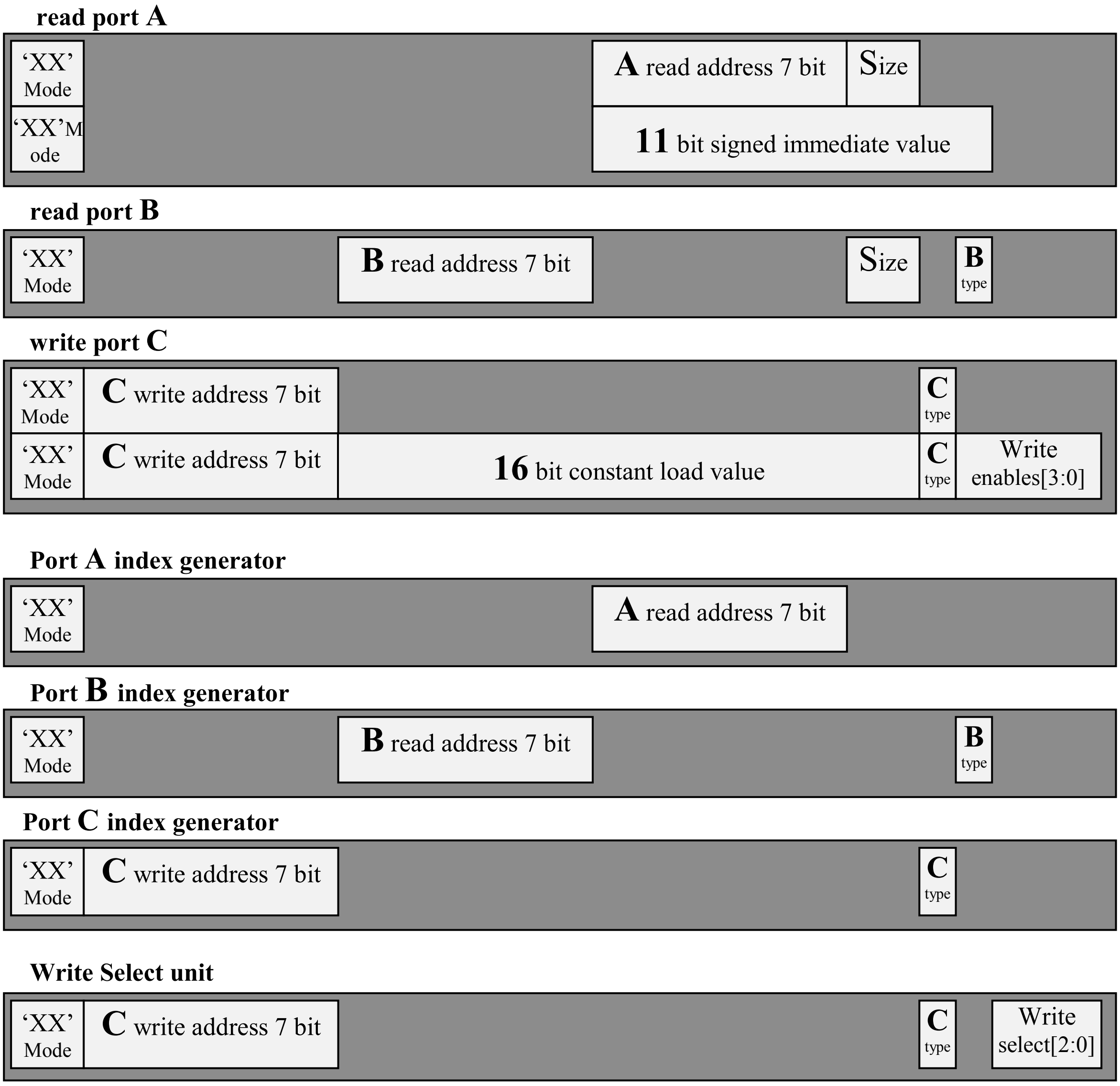
Four Byte write enables:	REG_C_Flags[W3, W2, W1, W0]	cr7[24,16,8,0]
Four Byte resets :	REG_C_Flags[R3, R2, R1, R0]	cr7[26,18,10,2]
Four Byte write enables:	REG_C_Flags[P3, P2, P1, P0]	cr7[27,19,11,3]

.

4.8.7 write Select Unit span of control

This unit selects the input bus during write operations. The only registers which it modifies are the registers of the write delay FIFO and the corresponding FIFO pointer REG_Control [FIFOPTR], cr0[18:16] which it may increment if it writes into the FIFO.

4.9 Instruction fields for each of the 7 sub-units of the register file



4.9.1 default values of instruction code fields

Not all of the fields are always available. A number of the fields have default values in case they are needed but aren’t supplied in the instruction field:

Size	IC[6:5]	the default value is: ‘10’	32 bit data
B	IC[3]	the default value is : ‘0’	register (not control register)
C	IC[4]	the default value is : ‘0’	register (not control register)
Wr enables	IC[3:0]	the default value is : ‘1111’	write all bytes

4.10 Events which modify the Register File’s control registers

All the control registers fields can be modified by writing to them with a control register write operation. Here we provide a detailed overview of the other events which modifies the control register fields.

4.10.1 events which modify REG_Control

ISIZEA[2:0] modified by a control register write only.
ISIZEB[2:0] modified by a control register write only.
ISIZEC[2:0] modified by a control register write only.

FIFOPTR[2:0] This pointer is auto incremented by a:
- Control register read to cr3: REG_Fifo and FIFOPTR[2:0] is not 0 or 7.
- Control register write to any byte of cr3: REG_Fifo and FIFOPTR[2:0] is not 0 or 7.
- Load-write-fifo instruction and FIFOPTR[2:0] is not 7.

This pointer is decremented by an:
- Extended Indexed Write and FIFOPTR[2:0] is not 0.

(The pointer stays the same if both incremented and decremented)

WE3,WE2,WE1,WE0 modified by a control register write only.
SE modified by a control register write only.
FUNCT modified by a control register write only.

VIPTR[1:0] This pointer is auto incremented by a:
- Control register read to cr3: REG_Fifo and FIFOPTR[2:0] is 0.
- Control register write to any byte of cr3: REG_Fifo and FIFOPTR[2:0] is 0.

4.10.2 events which modify REG_Monitor

ASIZE[1:0] loaded from the A bus by a control register write of ‘1’ to ME, cr1[7].
BSIZE[1:0] loaded from the B bus by a control register write of ‘1’ to ME, cr1[7].
QSIZE[1:0] loaded from the Q bus by a control register write of ‘1’ to ME, cr1[7].
FSIZE[1:0] loaded from the F bus by a control register write of ‘1’ to ME, cr1[7].
MSIZE[1:0] loaded from the M bus by a control register write of ‘1’ to ME, cr1[7].
USIZE[1:0] loaded from the U bus by a control register write of ‘1’ to ME, cr1[7].
DSIZE[1:0] loaded from the D bus by a control register write of ‘1’ to ME, cr1[7].
VSIZE[1:0] loaded from the V bus by a control register write of ‘1’ to ME, cr1[7].

ME can’t be modified, always returns a ‘0’.
SO not modified in case of a control register write of ‘1’ to ME, cr1[7].
ABUSSEL[2:0] not modified in case of a control register write of ‘1’ to ME, cr1[7].
BBUSSEL[2:0] not modified in case of a control register write of ‘1’ to ME, cr1[7].

4.10.3 events which modify REG_Vector

BASEA[1:0]	modified by a control register write only.
OFFSETA[2:0]	modified by a control register write only.
BASEB[1:0]	modified by a control register write only.
OFFSETB[2:0]	modified by a control register write only.
BASEC[1:0]	modified by a control register write only.
OFFSETC[2:0]	modified by a control register write only.
SELBUS[1:0]	modified by a control register write only.
SELSTA[2:0]	modified by a control register write only.
DW	modified by a control register write only.
IW	modified by a control register write only.
EW	modified by a control register write only.

4.10.4 events which modify REG_A_Indices

A3INDEX [7:0]	modified by a read port A index generation (only ISIZEA[2:0] bits are modified)
A2INDEX [7:0]	modified by a read port A index generation (only ISIZEA[2:0] bits are modified)
A1INDEX [7:0]	modified by a read port A index generation (only ISIZEA[2:0] bits are modified)
A0INDEX [7:0]	modified by a read port A index generation (only ISIZEA[2:0] bits are modified)

4.10.5 events which modify REG_B_Indices

B3INDEX [7:0]	modified by a read port B index generation (only ISIZEB[2:0] bits are modified)
B2INDEX [7:0]	modified by a read port B index generation (only ISIZEB[2:0] bits are modified)
B1INDEX [7:0]	modified by a read port B index generation (only ISIZEB[2:0] bits are modified)
B0INDEX [7:0]	modified by a read port B index generation (only ISIZEB[2:0] bits are modified)

4.10.6 events which modify REG_C_Indices

C3INDEX [7:0]	modified by a write port C index generation (only ISIZEC[2:0] bits are modified)
C2INDEX [7:0]	modified by a write port C index generation (only ISIZEC[2:0] bits are modified)
C1INDEX [7:0]	modified by a write port C index generation (only ISIZEC[2:0] bits are modified)
C0INDEX [7:0]	modified by a write port C index generation (only ISIZEC[2:0] bits are modified)

4.10.7 events which modify REG_C_Flags

PRE3[1:0]	modified by a control register write only.
PRE2[1:0]	modified by a control register write only.
PRE1[1:0]	modified by a control register write only.
PRE0[1:0]	modified by a control register write only.
RES3[1:0]	modified by a control register write only.
RES2[1:0]	modified by a control register write only.
RES1[1:0]	modified by a control register write only.
RES0[1:0]	modified by a control register write only.
P3, P2, P1, P0	modified by a write port C index generation.
R3, R2, R1, R0	modified by a write port C index generation.
S3, S2, S1, S0	modified by a read port A index generation.
W3, W2, W1, W0	modified by a write port C index generation.

4.11 Examples of vector operations with the register file

4.11.1 Example 1: Vectored 3 operand ROP with an 8x8 pattern

A typical MS window function is the pattern rectangle operation: On of the more complex versions performs an arbitrary logic function with a source rectangle, destination rectangle and an 8x8 pattern. This example archives this function with two vectors. The 8x8 pattern is stored in the vector register file. The size of the A indices is set to 3 bit, The generate index function increments the indices so the indices will wrap around after each eight accesses to generate the 8x8 pattern. The three operand logic function of the ALU is used which executes the logic function defined in the ALU_Logic control register

destination rectangle = function (rop3, source rectangle, destination rectangle, 8x8 pattern)

vector 1: Load values of the source image.

```
repeat,  graph ( pattern_rop3_load_source );
;;
pattern_rop3_load_source:
V= input  => DA= extended(), D= word(V);      // Load in non-cacheable on chip data memory
```

vector 2: Load values of the destination image, read 8x8 pattern, read source image perform any of 256 three operand logic functions and write the result back to the destination image.

```
repeat,  graph ( pattern_rop3_logic_operation );
;;
pattern_rop3_logic_operation:
DA=extended() => A=rd4x8(ri++) =>D=word(ul),V=input,U=pass(A) => F=logic(D,V,U) => V=output;
```

4.11.2 Example 2: Vectored parallel min/ max function

The current minimum (maximum) values are stored in the register file and read via the A port while new values are read in with the vector I/O unit for comparison. The new values are written into the delay FIFO to be conditionally written later after the calculation of the byte write enables. The subtract operation of the ALU provides the right status flags to determinate if the new values are smaller (larger) then the current ones, both for unsigned or signed compares. The genad(Wr) function generates the byte write enables and wr(xi) writes the delayed values from the FIFO into the vector register file. The genad function for read port A and the write port increments the read and write indices.

```
repeat,  graph ( minmax );
;;
minmax:
V= input, A= rd4x8(ri++) => wr (fifo, V),  F= sub(A,V) => genad(Wr) => wr(xi);
```

4.11.3 Example 3: Vectored parallel table look up function

This function uses a 256 entry translation table for a look up function which is useful for various purposes like pseudo color to real color conversion (1 value in, 4 values out), non-linear functions (4 values in, 4 values out) like gamma correction, histogram equalisation for contrast improvement for medical applications, solarisation effects for photoshop type applications. The input values are used as indices by the read port A index generator. The result is written back to memory.

```
repeat,  graph ( table_look_up );
;;
table_look_up:
V= input => genad(A) => A= rd4x8(ri) => V= output;
```

4.11.4 Example 4: Vectored parallel histogram function

This function can generate a 256 entry histogram from byte information. It counts the number of occurrences of a byte values. It can process two bytes values per cycle. The two byte values generate indices to two 16 bit locations which are read incremented and then stored back to the same location. The V= input function should use it's byte select function to move the two relevant bytes to bits[7:0] and [23:16] of the V bus and the V bus size should be set to 2x16. The On-the-Fly-increment function increments the count values. The Write port indices are copied from the read port A indices.

Example program: histogram without scaling.

```
repeat,  graph ( histogram );
;;
histogram:
V= input,  genad(A) => A= rd2x16(ri),  genad(Wr) => wr(xi, A);
```

The input values can be scaled into a certain range for the histogram, either to make the number of entries smaller or to reduce the number of input values into a 256 entry range. E.g. medical applications may scale 12 bit values into 8 bit values for histogram equalisation purposes. We can use the Multiplier to scale and use the M bus for the index offsets.

Example program: histogram with scaling.

```
repeat,  graph ( histogram );
;;
histogram:
V= input => M= mult(Q,V) =====> genad(A) => A= rd2x16(ri),  genad(Wr) => wr(xi, A);
```

4.11.5 Example 5: Vectored parallel add / subtract with saturate functions

Two vectors stored in the register file are added in this example. Pixel values are 8 bit (sometimes 16 bit) unsigned values with black level 0 and maximum level 0xFF (or 0xFFFF) Adding two images may not cause overflow: The results should be clamped to 0xFF (0xFFFF) Subtracting two images may not result in values less then 0: The results should be clamped to 0. Clamping to 0xFF (0xFFFF) is possible with the conditional Preset function and clamping to 0 is possible with the conditional reset function.

Add with saturate program:
The two vectors are read from the vector register file and added together in the ALU. The ALU produces a result plus carry flags in case of unsigned overflow. The result is written to the Write Delay fifo because we need a 1 cycle delay. The carry flags selected from the status register (cr15: ALU_RC_Status) are used to set the Preset flags in register cr7 (REG_C_Flags) In the last cycle we write the result from the Fifo, preset in case of carry, into the vector register

```
repeat,  graph ( add_saturate );
;;
add_saturate:
AB= rd4x8( ri++, ri++) => F= add(A,B) => wr (fifo, F), genad(Wr) => wr(xi);
```

Subtract with saturate program:
The two vectors are read from the vector register file and subtracted in the ALU. The ALU produces a result plus borrow (~carry) flags in case of unsigned under flow. The result is written to the Write Delay fifo because we need a 1 cycle delay. The inverse carry flags selected from the status register (cr15: ALU_RC_Status) are used to set the Reset flags in register cr7 (REG_C_Flags) In the last cycle we write the result from the Fifo, reset in case of borrow, into the vector register

```
repeat,  graph ( subtract_saturate );
;;
subtract_saturate:
AB= rd4x8( ri++, ri++) => F= sub(A,B) => wr (fifo, F), genad(Wr) => wr(xi);
```

4.11.6 Example 6: Vectored parallel run length encoder

This implementation operates on 4x8 bit data. It will produce two vectors: One vector with run length data and one vector with run length counts.

Example input vector with data which should be run length encoded:

2	2	2	2	116	2	2	123	123	123	2	167	2	2	2	2
2	2	116	116	116	2	123	2	2	2	2	167	167	167	2	2
2	116	2	2	116	2	123	2	2	2	2	167	2	2	167	2
2	116	2	2	116	2	2	123	123	2	2	167	2	2	167	2

Result Run length data vector:

2	116	2	123	2	167	2	-	-	-	-	-	-	-	-	-
2	116	2	123	2	167	2	-	-	-	-	-	-	-	-	-
2	116	2	116	2	123	2	167	2	167	2	-	-	-	-	-
2	116	2	116	2	123	2	167	2	167	2	-	-	-	-	-

Result Run length count vector: (count = runlength-1 which gives a range of 1 to 256)

3	0	1	2	0	0	3	-	-	-	-	-	-	-	-	-
1	2	0	0	3	2	1	-	-	-	-	-	-	-	-	-
0	0	1	0	0	0	3	0	1	0	0	-	-	-	-	-
0	0	1	0	1	1	1	0	1	0	0	-	-	-	-	-

vector 1: Load values for run-length encoding.

```
repeat, graph ( run_length_load );
;;
run_length_load:
V= input => DA= extended(), D= word(V);       // Load in non-cacheable on chip data memory
```

vector 2: Store the run length data in the vector register, store only once if a value occurs more then one time: The values which will be run length encoded are read back with the Data I/O unit on the D bus. The Vector I/O unit loads again the vector with values but 1 position shifted so the values on the D bus can be compared with their direct successor on the V bus. If a value is equal to it's successor then the count must be incremented (done in the 3rd vector) If it is not equal to it successor then we increment the write index and write the new value in the vector register. The zero flags of the ALU are used after a subtraction to test on equal or not equal.

```
repeat, graph ( run_length_values );
;;
run_length_values:
DA= extended( ) ==> D= word(ul), V= input => wr (fifo, D), F= sub(D,V) => genad(Wr) => wr(xi);
```

vector 3: Count the occurrences of a value (It's run length) increment write index if value is different. The start of this vector is equal to the previous. Values are compared with their direct successor. If they are equal then the count value which is stored in the vector register is incremented. If they are different then the index in the register file is incremented to point to the next count value which should be initialised to zero when written for the first time. This is done with the conditional clear function which also needs the zero flags of the ALU. It should clear on 'not equal' to initialise a new count value. The F= sub(D,V) function generates the status flags. The genad(A) increments the read address if not equal and loads the selected status flags in control register cr7. The genad(wr) in the next cycle re-uses the A-index for the write index and it uses the delayed status flags from cr7 for the conditional clear function. The count values will not overflow if counting is not continued more then 256 times. Count values should not be bigger then 256 anyway. They should fit in a single byte because what we want to do with this algorithm is compressing data. A new vector after 256 values should start again with new data and count indices.

```
repeat, graph ( run_length_counts );
;;
run_length_counts:
DA=extended() ==>D=word(ul),V=input =>F=sub(D,V) =>genad(A) =>A=rd(ri),genad(Wr) => wr(xi,A);
```

4.12 Interrupt processing:

- Saving and Restoring the data bus sizes of the A bus, B bus, Q bus, F bus, M bus, U bus, D bus and V bus:

The data sizes of the internal buses can be read and written back in control register REG_Monitor (cr1).

- Saving and Restoring the registers of the Write Delay Fifo:

The 7 individual registers of the Write Delay Fifo can be accessed via control register cr3: REG_Fifo. The selection between the various registers is made with the Fifo-pointer field in control register cr0: REG_Control bits [18:16] This field should have a value of 1..7 The field is post incremented by a control register read or write access. (The value of 0 is not incremented).

Chapter

5. BARREL SHIFT/ROTATE UNIT

The Barrel shift / Rotate Unit works on the basic data types of the Imagine: single 32 bit, double 16 bit and quadruple 8 bit. For all these formats it supports N-place shifting, logic as well as arithmetic, and N-place rotation.

fig. barrel shifter

5.1 operations

The Shift/Rotate unit can work on a single 32 bit word, double 16 bit words or quadruple 8 bit words. The word size is inherited from the source of the operand and stored together with the result in the Q register from where the result is sent to other functional units over the Q bus.

5.1.1 Operand select

The operand to be shifted can be selected from two buses: the B bus from the register file and the F bus from the ALU. The first one is used in typical register to register functions while the second uses the ALU result (the ALU also has the ability to redirect any of the other buses). The operand size is inherited from the B bus or the F bus.

5.1.2 Barrel shift functions

The three basic functions: rotate, shift logical and shift arithmetic are provided. The selected operand to be shifted (B bus or F bus) can have any word size (32, 2x16, 4x8).
The operator A is defined as a single two's complement word taken from the A bus. This word can be 32, 16 or 8 bit. In the last two cases the lowest word is used (A0..A15 and A0..A7).

5.1.3 Shift direction

If A is positive then the shift (rotate) direction is left. If A is negative then the shift (rotate) direction is right.
If A is larger than the maximum number of locations which can be shifted then the result will still be correct: all '0's or all '1's.

The bits shifted in by the shift logical are always '0'. The shift arithmetic should be seen as a multiplication by a power of 2 on a two's complement number. If we look at the case of a negative value for the operand (sign bit is '1') and a negative operand for the operator (shift right: divide by a power of 2) then we see that the number should stay negative and so '1's have to be shifted in at the left side.

Barrel shifter / rotator functions

32:30	Mnemonics	function	size
0	**Q=noop**	no operation	B size
1	**Q=rotate(B,A)**	rotate input B over A positions	B size
2	**Q=shflog(B,A)**	shift input B logical over A positions	B size
3	**Q=shfarh(B,A)**	shift input B arithmetical over A positions	B size
4	**Q=extended**		B size / F size
5	**Q=rotate(F,A)**	rotate input F over A positions	F size
6	**Q=shflog(F,A)**	shift input F logical over A positions	F size
7	**Q=shfarh(F,A)**	shift input F arithmetical over A positions	F size

5.1.4 The result register of the Barrel Shifter

The results of the Barrel Shift function unit are available in the Q bus register which can be used by other functional units, The register file or the I/O units This register is also accessible as a control register (BSH_Qbus, cr8)

Cr8 **BSH_Qbus:** Barrel Shifter Bus Registers

<table>
<tr><td colspan="32">32 bit Qbus result data (4x8, 2x16 or 1x32)
[31:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

5.1.4 The extended function of the Barrel Shifter

The functions of the barrel shifter can also be performed with a fixed shift factor which can be defined in control register BSH_Control (cr9). The 8 bit fixed shift factor is a signed byte (-128, +127). The higher bits are significant for the shift function: A word can be completely shifted out of the register leaving only zeroes or ones (Shift Arithmatic right of a negative number). The Bus can be selected with the BUS_SEL field (0=B bus, 1=F bus) and the Function can be selected with the Function Field (0=noop, 1=rotate, 2=shflog, 3=shfarh)

cr9 **BSH_Control:** Barrel Shifter Control register for the extended function

<table>
<tr><td colspan="18">'0000 0000 0000 0000 00'</td><td colspan="2">FUNCT
[1:0]</td><td colspan="3">'000'</td><td>bus
sel</td><td colspan="8">SHIFT FACTOR
[7:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

Cr8 **BSH_Qbus:** Barrel Shifter Bus Registers

<table>
<tr><td colspan="32">32 bit Qbus result data (4x8, 2x16 or 1x32)
[31:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

Chapter

6. ARITHMETIC & LOGIC UNIT

The Arithmetic and Logic Unit works on the basic data types of the Imagine: single 32 bit, double 16 bit and quadruple 8 bit. It has the entire set of the 16 possible logic functions and a second set of 15 additive type instructions like addition, subtraction, negation, increment, decrement et cetera. Addition and subtraction functions with saturation are available. A special function not found in other processors but of high value in Window graphics applications is the parametrised three operand logic function: a software parameter can select between any of the 256 possible logic operations on three operands.

fig. alu

The ALU can be used as a single 32 bit ALU, a double 16 bit ALU or a quadruple 8 bit ALU. The 9 bit instruction field consists of two sub fields: the operand source select field and the ALU instruction.

6.1 Operand Source select:

The two operands for the ALU, called the R and the S input, can both be selected from any of four inputs. In most cases the operand size is inherited from the S source, except for a number of operations that use the R source only; in these cases the R size is used.

6.2 ALU function:

The ALU provides three types of operations:

♦ The set of 16 elementary two operand logic functions. We use the X window convention for the mnemonics of the logic functions.

♦ A set of 15 add/ subtract/ increment/ decrement functions. Addition and subtraction with saturation are available. The **addsat()** function clips to 0xFF, 0xFFFF or 0xFFFFFFFF for 8, 16 or 32 bit operations in case of overflow while the function **subsat()** clips to 0 in case of underflow.

♦ A parametrised three port logic function which uses an eight bit control register to define any of the 256 elementary three operand logic functions.

Operand Select

Ic[41:38]	R input	S input
0	R = A bus	S = B bus
1	R = A bus	S = V bus
2	R = A bus	S = F bus
3	R = A bus	S = U bus
4	R = D bus	S = B bus
5	R = D bus	S = V bus
6	R = D bus	S = F bus
7	R = D bus	S = U bus
8	R = M bus	S = B bus
9	R = M bus	S = V bus
A	R = M bus	S = F bus
B	R = M bus	S = U bus
C	R = Q bus	S = B bus
D	R = Q bus	S = V bus
E	R = Q bus	S = F bus
F	R = Q bus	S = U bus

6.3 ALU instruction set

Logic functions

Ic[37:33]	Mnem	Operation	size
00	**F=clear**	F = all 0s	Ssz
01	**F=and(R,S)**	F = R and S	Ssz
02	**F=andrev(R,S)**	F = (not R) and S	Ssz
03	**F=copy(S)**	F = S	Ssz
04	**F=andinv(R,S)**	F = R and (not) S	Ssz
05	**F=noop(R)**	F = R	Rsz
06	**F=xor(R,S)**	F = R xor S	Ssz
07	**F=or(R,S)**	F = R or S	Ssz
08	**F=nor(R,S)**	F = (not R) and (not S)	Ssz
09	**F=equiv(R,S)**	F = R xnor S	Ssz
0A	**F=invert(R)**	F = not R {-R-1}	Rsz
0B	**F=orrev(R,S)**	F = (not R) or S	Ssz
0C	**F=copyinv(S)**	F = not S {-S-1}	Ssz
0D	**F=orinv(R,S)**	F = R or (not S)	Ssz
0E	**F=nand(R,S)**	F = (not R) or (not S)	Ssz
0F	**F=set**	F = all 1s	Ssz

Arithmetic functions

Ic[37:33]	Mnem	Operation	size
10	**F=decr(R)**	F = R-1	Rsz
11	**F=incr(R)**	F = R+1	Rsz
12	**F=decr(S)**	F = S-1	Ssz
13	**F=incr(S)**	F = S+1	Ssz
14	**F=subdecr(R,S)**	F = R-S-1	Ssz
15	**F=sub(R,S)**	F = R-S	Ssz
16	**F=subsat(R,S)**	F = R-S or 0*	Ssz
17	**F=minus(S)**	F = -S	Ssz
18	**F=subdecr(S,R)**	F = S-R-1	Ssz
19	**F=sub(S,R)**	F = S-R	Ssz
1A	**F=subsat(S,R)**	F = S-R or 0*	Ssz
1B	**F=minus(R)**	F = -R	Rsz
1C	**F=add(R,S)**	F = R+S	Ssz
1D	**F=addincr(R,S)**	F = R+S+1	Ssz
1E	**F=addsat(R,S)**	F = R+S or max*	Ssz

(*) saturation

Parametrised logic function

Ic[37:33]	Mnemonics	Operation	size
1F	**F=Logic(R,S,U)**	F = logic_function(R,S,U)	Ssz

6.4 Three port parametrised logic functions

ALU operation 1F invokes bit0..7 from the ALU control register (cr12) which can provide a parametrised logic function for the ALU: the eight bits in the register can define any arbitrary three operand logic function. Typical applications are the ROP functions of windows 95 where every graphics operation should be possible with an arbitrary logical function and a bitplane mask: F = ((R op S) & U) | (S & !U). Another example are Functions like shift/mask/merge and shift/mask/compare for the handling of compacted binary images. The Shift/Rotate unit provides the shift function and the ALU unit provides the three operand logic function:
F = (R & U) | (R & !U) -> Q = rotate (F,A);

control register cr13:

Inputs:	→ result:
Un=0, Sn=0, Rn=0	→ Fn=L7
Un=0, Sn=0, Rn=1	→ Fn=L6
Un=0, Sn=1, Rn=0	→ Fn=L5
Un=0, Sn=1, Rn=1	→ Fn=L4
Un=1, Sn=0, Rn=0	→ Fn=L3
Un=1, Sn=0, Rn=1	→ Fn=L2
Un=1, Sn=1, Rn=0	→ Fn=L1
Un=1, Sn=1, Rn=1	→ Fn=L0

6.5 ALU control register: logic_function

The L(x) bit determines the result value depending on the input values of the Un, Sn and Rn operands. This operation is independently executed for all n (0..31) bits of the result value F.

ALU_Logic: cr13

MASK_SEL			L7	L6	L5	L4	L3	L2	L1	L0
10	9	8	7	6	5	4	3	2	1	0

6.6 The ALU status register

ALU_RC_Status: cr15: Status flags from the ALU and Multipler / Accumulator

W3	L3	H3	I3	S3	C3	M3	Z3	W2	L2	H2	I2	S2	C2	M2	Z2	W1	L1	H1	I1	S1	C1	M1	Z1	W0	L0	H0	I0	S0	C0	M0	Z0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The Status register (control register cr15) contains sixteen bits from the ALU (b0..3 b8..11, b16..19, b24..27). These bits are the status flags generated by the ALU in the previous cycle. Each of the four 8 bit words has its own four status outputs:

Z	Zero	not (b0+b1+b2+..bn)
M	Minus	value of b_{msb}
C	Carry	from b_{msb} to b_{msb+1}
S	SgnCmp	Overflow xnor Minus

The definition of the status bits is depending on the F word size:

The instruction F = copy(F) does not change the contents of the F register and is also a nop for the status register flags: the Carry, SgnCmp and other flags will not change.

Zero	8 bit	16 bit	32 bit
Z3	z3	z3.z2	z3.z2.z1.z0
Z2	z2	z3.z2	z3.z2.z1.z0
Z1	z1	z1.z0	z3.z2.z1.z0
Z0	z0	z1.z0	z3.z2.z1.z0

(z0..z3 are the byte oriented zero flags).

Minus	8 bit	16 bit	32 bit
M3	m3	m3	m3
M2	m2	m3	m3
M1	m1	m1	m3
M0	m0	m1	m3

(m0..m3 are the byte oriented minus flags).

Carry	8 bit	16 bit	32 bit
C3	c3	c3	c3
C2	c2	c3	c3
C1	c1	c1	c3
C0	c0	c1	c3

(c0..c3 are the byte oriented carry flags).

Sgncmp	8 bit	16 bit	32 bit
S3	s3	s3	s3
S2	s2	s3	s3
S1	s1	s1	s3
S0	s0	s1	s3

(s0..s3 are byte oriented sgncmp flags).

6.7 Conditional Control Flow Processing:

The Program sequencer can use the ALU results for conditional operations. It can use Z0, M0, C0 and SO for conditional Jumps, Calls and Returns. The programmer and C compiler can make comparisons between 32 bit, 16 bit and 8 bit words (the latter two cases use the lowest sub word 0..15 and 0..7).

zero:
Zero flag: This flag is used for 'is equal' and 'is not equal' conditional expressions.

carry:
Carry flag: This flag is used for all **unsigned** conditional expressions like 'greater than' or 'smaller than or equal to' etc..

sgncmp:
Signed Compare: This flag is used for all **two's complement** conditional expressions like 'greater than'
or 'smaller than or equal to' etc.

Test on equal / not equal

	flag	**ALU**
X == Y	if zero	X-Y
X != Y	if not (zero)	X-Y

Unsigned integer equations

	flag	**ALU**
X >= Y	if (carry)	X-Y
X <= Y	if not (carry)	X-Y-1
X > Y	if (carry)	X-Y-1
X < Y	if not (carry)	X-Y

Signed integer equations

	flag	**ALU**
X >= Y	if (sgncmp)	X-Y
X <= Y	if not (sgncmp)	X-Y-1
X > Y	if (sgncmp)	X-Y-1
X < Y	if not (sgncmp)	X-Y

6.8 using status for conditional register access

The status information can be used internally in the Register file address generator for conditional addressing and conditional writing into the register file. The status bits are also available in control register **ALU_RC_Status** (= cr15). This register is readable and writable. A write operation overrules the new status flags from the ALU.

6.9 using status for the range mask:

The Mask Generator can use the ALU results to assemble the Range Mask which can be used for conditional vector write operations to external Image memory. The field **MASK_SEL** selects the four status bits which are send to the Mask Generator where they can be assembled into the 64x4 bit Range Mask.

MASK_SEL: status flags→ range mask

000:	**ALU_ZERO**
001:	**ALU_MINUS**
010:	**ALU_CARRY**
011:	**ALU_SGNCMP**
100:	**ALU_NOT_ZERO**
101:	**ALU _NOT_MINUS**
110:	**ALU _NOT_CARRY**
111:	**ALU _NOT_SGNCMP**

ALU_Logic: cr13

<table>
<tr><td colspan="5">'00000'</td><td colspan="3">MASK_SEL</td><td>L7</td><td>L6</td><td>L5</td><td>L4</td><td>L3</td><td>L2</td><td>L1</td><td>L0</td></tr>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

6.10 direct control register access to the F bus register:

The result register of the ALU: The F bus register is directly accessible as control register **ALU_Fbus** (= cr12)

cr12: **ALU_Fbus:** Accumulator F Bus Register

<table>
<tr><td colspan="32">32 bit F bus result data (4x8, 2x16 or 1x32)
[31:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

Chapter

7. MULTIPLIER / ACCUMULATOR.

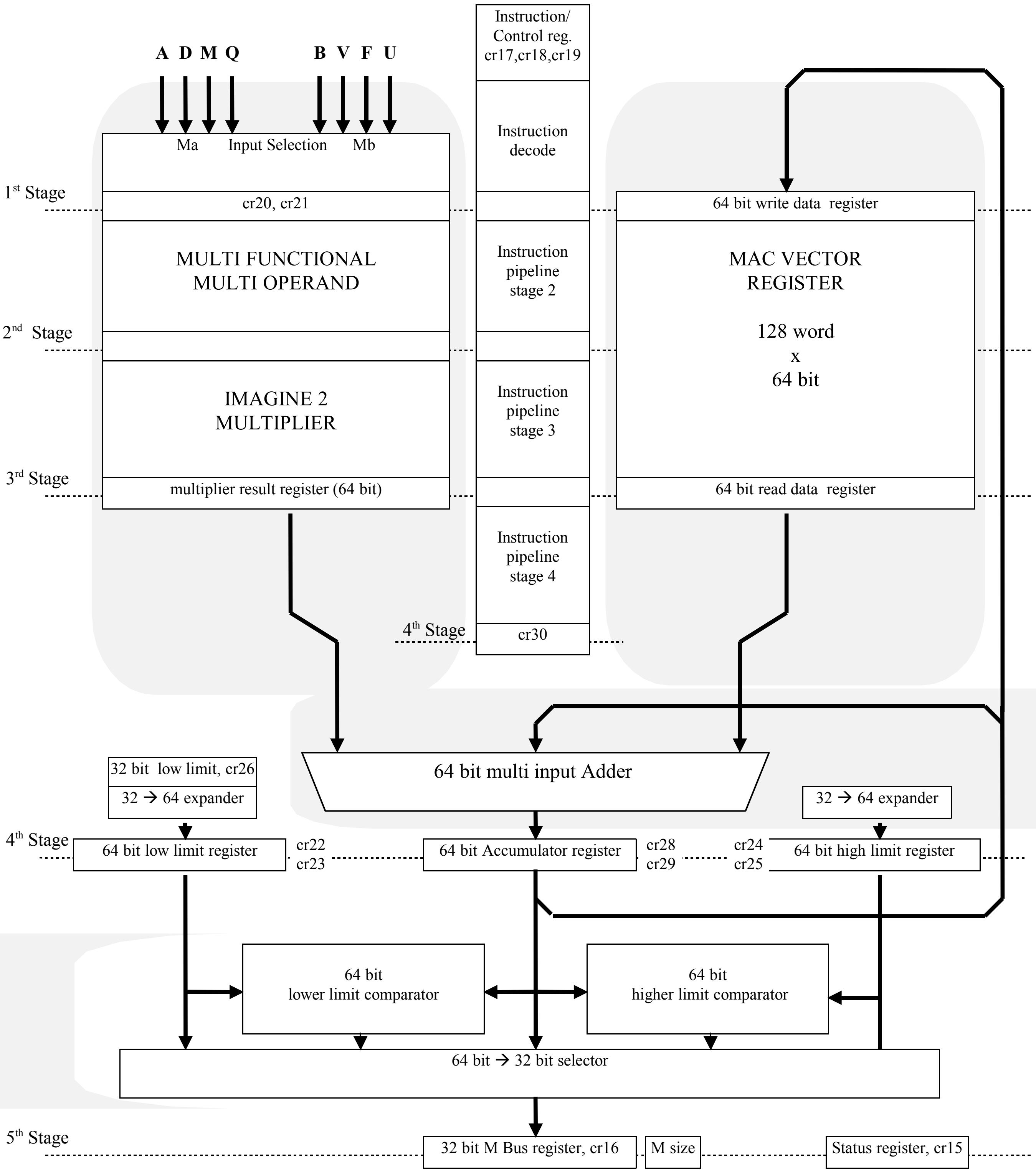
The Multiplier/Accumulator is a truly multifunctional unit and it supports various data types in an even more diverse way than the other functional units. The basic multiplication is supported in a regular and orthogonal way for different word sizes and data types. Like the other functional units it supports single 32 bit, double 16 bit and quadruple 8 bit operations. These operations are performed on integers, fixed point numbers, normalised numbers and so-called graphics numbers. Both operands can be independently signed or two's complement. It can apply either rounding or truncation.

The Extended function set holds more special sum of product functions like 8 bit 4x4 matrix times vector multiplication and quadruple (4x1) inproducts (both 16 multiplications, 12 additions), blend functions (8 multiplications and 6 additions), and 16 bit complex, vector dot and vector cross products.

The Accumulator stage twice the original word length: single 64 bit, double 32 bit and quadruple 16 bit. It can accumulate a single stream of multiplication results into a single result or it can do vector accumulation to accumulate N streams of M results into M final results. The Accumulator has the ability to work stand alone for incremental operations for "Digital Difference Engine" functions.

The Range Control Unit checks if data is within the limits of a range with an upper and a lower boundary. It can handle unsigned integers as well as two's complement numbers. It can replace out of range data with the limit values and/ or can generate a 2 dimensional pixel masker from the range check results. This masker is used in the IMAGINE 2 for masked write operations to the Image memory. The Boolean results of the Range clip unit are available in the Status register which can be used in the Three port register file for conditional addressing and write enabling.

Detailed overview of the IMAGINE 2 Multiplier /Accumulator



Internal 64 bit formats:

<table>
<tr><td colspan="4">64 bit data</td></tr>
<tr><td colspan="2">32 bit data</td><td colspan="2">32 bit data</td></tr>
<tr><td>16 bit data</td><td>16 bit data</td><td>16 bit data</td><td>16 bit data</td></tr>
</table>

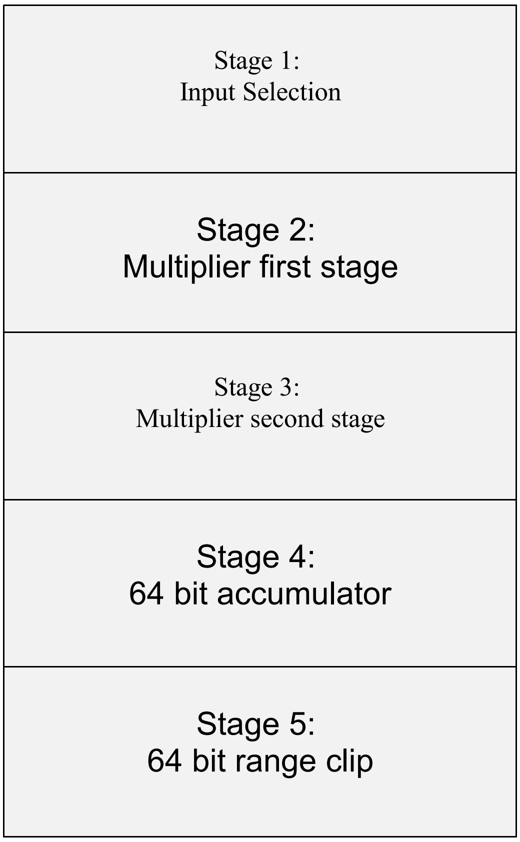
7.1 Multiplier / Accumulator

7.1.1 The multiplier accumulator

The Multiplier/Accumulator is a five stage pipelined unit with a wide variety of functions. The basic set of multiplications is directly executed by the instruction code. Much more advanced operations are performed by a combination of extended instructions and control register parameters. Instruction and control information are packed together and sent into the multiplier pipeline. Once an instruction is given, it will be executed until completed. This means that control registers can be changed without the risk of disturbing on-going operations (intermediate modification of internal multiplier registers which contain *data* such as the Accumulator register and the Compare registers of the Range Unit *do* change the outcome of ongoing operations).

7.1.2 The pipeline

The multiplier/accumulator has five internal pipeline-stages. This means that it takes five steps to complete a multiply / accumulation operation. Latency is the same for all multiply operations, with or without accumulation and optional range clipping. Since the MAC is pipelined, one result can be produced each cycle. 'One result' means a single 32 bit result, two 16 bit results or four 8 bit results. The type of multiplication can change each cycle. The instructions code starts the operation in the multiplier/accumulator pipeline. The result is available five cycles later.



7.1.3 multiplier operand select

The two inputs of the multiplier are referred to as the Ma and Mb operands. The output of the multiplier is stored in the M bus register from where it is made available to the other functional units in the Imagine.
The operands can be single 32, double 16 and quad 8 bit words

The internal results used by the accumulators are 64, double 32 and quad 16 bit. The operands have besides a wordsize a number of other attributes. An operand can either be unsigned or two's complement. It can have one of four formats: Integer, fixed point, normalised or graphics format. The outcome of the operation depends on these attributes as well.

The Multiplier operand select field:

50:47	Ma input sources	Mb input sources	Data size
0	**A bus** (registers)	**B bus** (registers)	Msz = Bsz
1	**A bus** (registers)	**V bus** (vector I/O)	Msz = Vsz
2	**A bus** (registers)	**F bus** (ALU)	Msz = Fsz
3	**A bus** (registers)	**U bus** (UFU)	Msz = Usz
4	**D bus** (data I/O)	**B bus** (registers)	Msz = Bsz
5	**D bus** (data I/O)	**V bus** (vector I/O)	Msz = Vsz
6	**D bus** (data I/O)	**F bus** (ALU)	Msz = Fsz
7	**D bus** (data I/O)	**U bus** (UFU)	Msz = Usz
8	**M bus** (multiplier)	**B bus** (registers)	Msz = Bsz
9	**M bus** (multiplier)	**V bus** (vector I/O)	Msz = Vsz
A	**M bus** (multiplier)	**F bus** (ALU)	Msz = Fsz
B	**M bus** (multiplier)	**U bus** (UFU)	Msz = Usz
C	**Q bus** (shifter)	**B bus** (registers)	Msz = Bsz
D	**Q bus** (shifter)	**V bus** (vector I/O)	Msz = Vsz
E	**Q bus** (shifter)	**F bus** (ALU)	Msz = Fsz
F	**Q bus** (shifter)	**U bus** (UFU)	Msz = Usz

7.2 The basic set of multiplier operations

Two function sets are supplied for the MAC (Multiplier/Accumulator) on the Imagine: the **Basic** set and the **Extended** set. The Basic function set includes all normal multiplication functions. 48 Different multiplications are possible in an orthogonal way.

7.2.1 The Basic Multiply options

The functions in the basic set are entirely defined by the instruction and the data size and do not depend of the contents of any of the control registers. The operand size chooses between 32x32=64, double 16x16=32 and quadruple 8x8=16 multiplications. The Data size used for the operation is taken from the Mb input operand. The multiplications can have an integer or one of various fixed point fractional formats. Both input operands can be independently signed or unsigned (the result will be signed if one ore more inputs are signed). Four bits in the instruction code select the various multiplication options if the basic function set is selected. Data Type and Sign information define the type of multiply operation. This information is stored in control register cr19 of the multiplier from where it can be used for the multiply/accumulate function:

M = macs()

There is a choice between three sizes:

single 32 bits multiplication
double 16 bits multiplication
quadruple 8 bits multiplication

There are four sign options:

unsigned multiplication,
signed multiplications
unsigned x signed
signed x unsigned

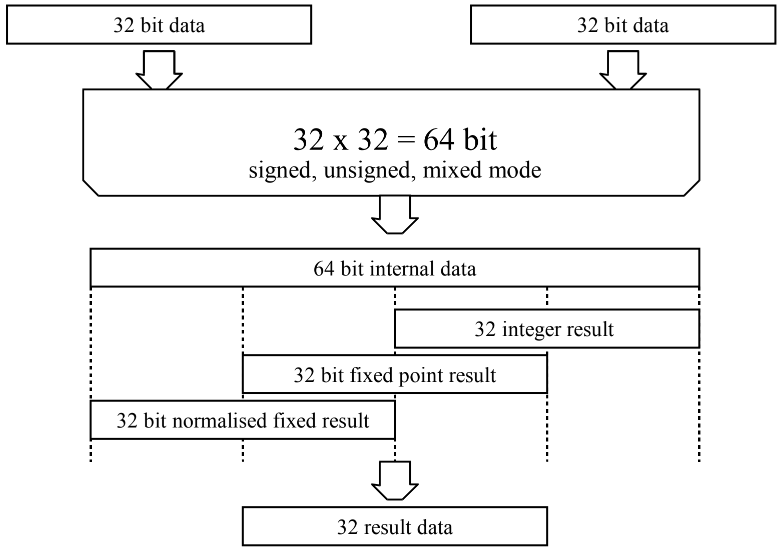
All operations are possible on four data types:

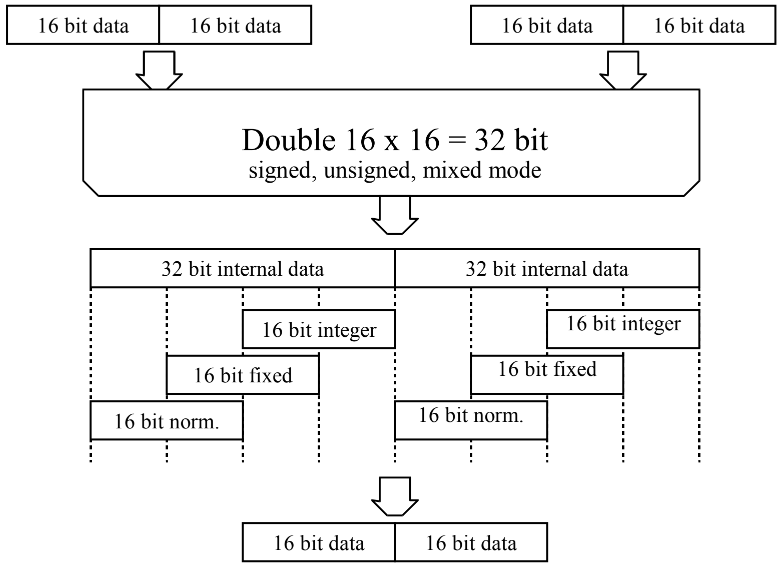
Integer format	(point at end)
Normalised format	(point at begin)
Fixed point format	(point halfway)
Graphics format	(point at begin)

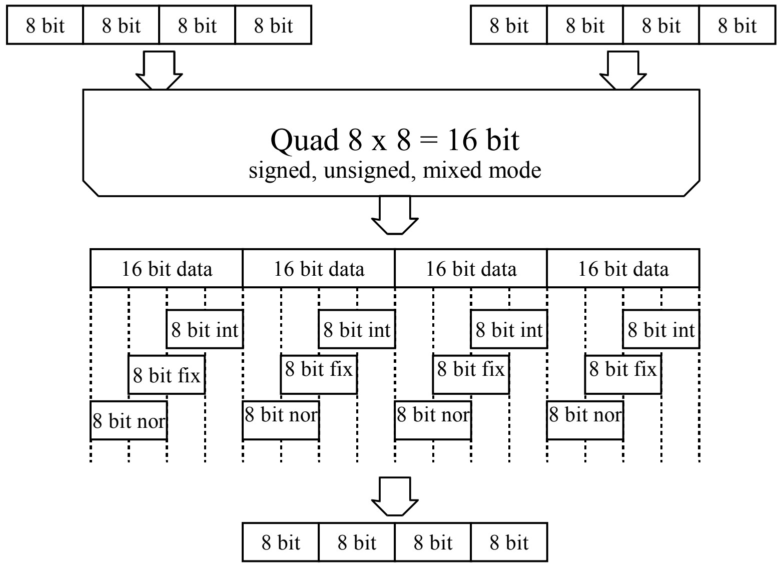
7.2.2 Multiplications defined in the basic set

[46:42]	Mnemonics	multiplier type	Ma sign	Mb sign
00	**M = mult (Ma,Mb,iuu)**	Integer format	unsigned	unsigned
01	**M = mult (Ma,Mb,ius)**	Integer format	unsigned	signed
02	**M = mult (Ma,Mb,isu)**	Integer format	signed	unsigned
03	**M = mult (Ma,Mb,iss)**	Integer format	signed	signed
04	**M = mult (Ma,Mb,nuu)**	Normalised format	unsigned	unsigned
05	**M = mult (Ma,Mb,nus**)	Normalised format	unsigned	signed
06	**M = mult (Ma,Mb,nsu)**	Normalised format	signed	unsigned
07	**M = mult (Ma,Mb,nss**)	Normalised format	signed	signed
08	**M = mult (Ma,Mb,fuu)**	Fixed Point format	unsigned	unsigned
09	**M = mult (Ma,Mb,fus**)	Fixed Point format	unsigned	signed
0A	**M = mult (Ma,Mb,fsu)**	Fixed Point format	signed	unsigned
0B	**M = mult (Ma,Mb,fss)**	Fixed Point format	signed	signed
0C	**M = mult (Ma,Mb,guu)**	Graphics format	unsigned	unsigned
0D	**M = mult (Ma,Mb,gus)**	Graphics format	unsigned	signed
0E	**M = mult (Ma,Mb,gsu**)	Graphics format	signed	unsigned
0F	**M = mult (Ma,Mb,gss)**	Graphics format	signed	signed

BASIC MULTIPLICATIONS







7.2.3 The multiplier operand types

Besides a size (32, 2x16 or 4x8) the operands have an operand type as well. The use of these types should simplify the use of basically integer calculations in pre-calculations for graphic algorithms. Fixed point calculations offer fractional calculations as opposed to pure integer calculations. The location of the binary point is essential in the multiplication operation, contrary to addition and subtraction operations.

If we define a fixed point value by **m.n**, where m is the number of bits before the binary point and n the number of bits behind, then addition type operations will produce **m+1.n** results and multiplication operations will produce **2m.2n** results: the word length of the result is doubled both before and after the binary point. This means that the result should be shifted **n** places to the right (except for integer numbers where n=0). The types specified for the Imagine are integer, normalised , and fixed point. Integer is defined as **p.0**, normalised as **0.p** and fixed point as **p/2.p/2** where p is the word length. If a multiplication is specified with a certain type then this may be interpreted as an integer multiplication with an implicit shift to the right: **p** places in case of a normalised fixed point multiplier and **p/2** places in case of a fixed point multiplier. Notice that the multiplier type specifies only one of the two operands, either Ma or Mb. The other one can be in any format **m.n** while the result value, after being sized to the original word length, gets the same **m.n** format. Which one of the two operands (Ma or Mb) is seen as having the type specified is merely a question of interpretation.

7.2.4 Internal and output formats

	Internal 64 bit Multiply Result	32 bit Selected M bus output
32 bit		
integer	[63:00]	[31:00]
fixed point	[63:00]	[47:16]
normalised	[63:00]	[63:31]
2x16 bit		
integer	[63:32] [31:00]	[47:32] [15:00]
fixed point	[63:32] [31:00]	[55:40] [23:08]
normalised	[63:32] [31:00]	[63:48] [31:16]
4x8 bit		
integer	[63:48] [47:32] [31:16] [15:00]	[55:48] [39:32] [23:16] [07:00]
fixed point	[63:48] [47:32] [31:16] [15:00]	[59:52] [43:36] [27:20] [11:04]
normalised	[63:48] [47:32] [31:16] [15:00]	[63:56] [47:40] [31:24] [15:08]

7.2.5 The Graphics data format

The graphics type has the point at the MSB position like the normalised data format, however with an essential difference: in many cases an n-bit-word should represent a value from 0% up to *and including* 100%. Some examples are the colour values RGB where Red=0 means 0% red and Red=255 means 100% red. Another example is the alpha-plane where α=0 means 100% transparency and α=255 means 0% transparency. A multiplication by 255 must be equal to a multiplication by 1.000 exactly. The graphics data format offers this option for 8, 16 and 32 bit multiplications, both unsigned and signed:

Unsigned multiplications:

8 bit:	**M = (Ma x Mb) x (256/ 255)**
16 bit:	**M = (Ma x Mb) x (2^{16}/ 2^{16}-1)**
32 bit:	**M = (Ma x Mb) x (2^{32}/ 2^{32}-1)**

Signed multiplications:

8 bit:	**M = (Ma x Mb) x (128/ 127)**
16 bit:	**M = (Ma x Mb) x (2^{15}/ 2^{15}-1)**
32 bit:	**M = (Ma x Mb) x (2^{31}/ 2^{31}-1)**

7.3 The extended multiplier functions

The Extended function set includes the more specialised functions like the 16 bit complex multiplication, the 4x4 matrix times vector multiplication et cetera. Many of them can also use the Accumulator stage and the Range Clip stage of the MAC. The two multiply/accumulate instructions, **M = macs()** and **M = macb()**, use multiplications from the basic set but allow the use of the Accumulator stage and the Range Clip stage. The second table indicates which resources a given instruction can use. It defines which fields of the **MAC_Control1** register (cr17) are used during the execution of the instruction. Some fields refer to two other registers containing control information. The **MAC_RamPtrs** register (cr19) is used when the **read** or **write** fields indicate an Access to the Accumulator Ram file and the coefficient registers. This control register contains
read and write pointers. The **MAC_Control2** register (cr18) is used if the Range Unit field (**range**) is TRUE (logical '1'). The Range Unit control register uses two 64 bit registers with a Low and High limit (cr22, cr23 and cr24, cr25). The data **size**, **type** and **sign** fields define the data format used for a certain multiplication function, e.g.: the matrix x vector product can be performed in 16 different ways. The **accu** field selects between the three operands for the Accumulator: the Multiplier result, the Accumulator Ram file and the Accumulator itself. The **pipe** field controls the two 4x4 register sets for the matrix operations. These are located in the first stage where the multiplier input operands are selected.

	Mnemonics	operation (Single cycle throughput)
10	**M = inproduct (Mb)**	quadruple vector inproduct (4 x 8 bit vectors)
11	**M = matrixvec (Mb)**	4x4 matrix vector multiplication (8 bit)
12	**M = blend (Ma,Mb)**	Open GL compatible blend function
14	**M = loadpipe (Ma,Mb)**	shift 4x4 matrix data and coefficient pipelines
15	**M = read_ram ( )**	read 64 bit word from the accu Ram into accumulator
16	**M = write_ram ( )**	write 64 bit word from the accumulator to accu Ram
17	**M = linearstep(*)**	incremental add 4x16 bit, 2x32 bit, 1x64 bit
18	**M = macs (Ma,Mb)**	multiply accumulate (scalar)
19	**M = macb (Ma,Mb)**	multiply accumulate (block)
1C	**M = vectprod (Ma,Mb)**	16 bit vector dot product and cross product
1D	**M = complex (Ma,Mb)**	16 bit complex multiply M = a*b - b*c + i(a*d + b*c)
1E	**M = nop**	no operation
1F	**M = halt**	halt MAC: freeze the entire MAC pipeline.

(*) = optional Mb operand for Range Unit: () or (Mb)

		Write	read	size	type	sign	accu	out	range	pipe
	Mnemonics	30-29	27-26	21-22	19-18	17-16	15-12	10-8	7	4-0
10	**M = inproduct (Mb)**	used	used	**4x8**	used	used	used	used	used	used
11	**M = matrixvect (Mb)**	used	used	**4x8**	used	used	used	used	used	----
12	**M = blend (Ma,Mb)**	used	used	**4x8**	**graph**	**unsign**	used	used	used	----
14	**M = loadpipe (Ma,Mb)**	----	----	**4x8**	----	----	----	used	used	used
15	**M = read_ram ()**	----	used	used	----	----	**ram**	used	used	----
16	**M = write_ram ()**	used	----	used	----	----	----	used	used	----
17	**M = linearstep (*)**	used	used	used	----	----	**a+r**	used	used	----
18	**M = macs (Ma,Mb)**	----	----	**bus**	**prev**	**prev**	**a+m**	----	----	----
19	**M = macb (Ma,Mb)**	used	used	used	used	used	used	used	used	----
1C	**M = vectprod (Ma,Mb)**	used	used	**2x16**	used	**sign**	used	used	used	----
1D	**M = complex (Ma,Mb)**	used	used	**2x16**	used	**sign**	used	used	used	----
1E	**M = nop**	----	----	----	----	----	----	----	----	----
1F	**M = halt**	----	----	----	----	----	----	----	----	----

(*) = optional Mb operand for Range Unit: () or (Mb)

7.4 Description of the multiplier operations

7.4.1 Operands for the multiplier

The two input values for the multiplier are referred to as Ma and Mb, the result is referred to as M. We use sub indices to distinguish between the 3 different word sizes. Single 32 bit words have no sub indices, double 16 bit words have the sub indices **H** and **L**, and the quadruple 8 bit words have sub indices **3,2,1** and **0**. The 4x4 matrix functions use two 4x4 sets of internal registers to provide operands: the P (Pipeline) registers and the c (coefficient) registers. These registers are enumerated with super indices: $c^{0..3}$ and $P^{0..3}$. Each of these contain four bytes which are enumerated with sub indices. Some examples:

single 32 bit: $\mathbf{Ma = Ma_{(31..0)}}$ $\mathbf{P^2 = P^2_{(31..0)}}$
double 16 bit: $\mathbf{Ma_H = Ma_{(31..16)}}$, $\mathbf{Ma_L = Ma_{(15..0)}}$
quad 8 bit: $\mathbf{Mb_3 = Mb_{(31..24)}}$, $\mathbf{Mb_0 = Mb_{(7..0)}}$ $\mathbf{c^3_2 = c^3_{(23..16)}}$

The intermediate internal multiplication results are defined by Mi. The Sub indices are applied in a similar way, with the exception that the intermediate results have a word length 1.5 times the input size: 12, 24 and 48 bits. Some examples:

$\mathbf{Mi_H = Mi_{(47..24)}}$, $\mathbf{Mi_2 = Mi_{(35..24)}}$, $\mathbf{Mi_0 = Mi_{(11..0)}}$
The value of the individual bits depends on the operand size as well as the operand type (Integer, Fixed point).

7.4.2 Basic operations

The multiplication functions from the basic set are defined as follows:

single 32 bit: $\mathbf{M = Ma.Mb}$
double 16 bit: $\mathbf{M_H = Ma_H.Mb_H}$, $\mathbf{M_L = Ma_L.Mb_L}$
quad 8 bit: $\mathbf{M_3 = Ma_3.Mb_3}$, $\mathbf{M_2 = Ma_2.Mb_2}$, $\mathbf{M_1 = Ma_1.Mb_1}$, $\mathbf{M_0 = Ma_0.Mb_0}$

Additional parameters are given within the instruction (fixed, integer..., unsigned, two's complement, mixed).

7.4.3 8 bit Matrix functions: Quad Inproduct

M = inproduct(Mb): Correlation and convolution, interpolated scaling and affine transformation.

The four 8 bit data words come from the Mb input steps as a four byte column from left to right through the multiplier array via the data pipeline registers. The values in the coefficient registers are constant (they are the four times four coefficients for the four inproducts).

$$\mathbf{M_3 = c^3_3.P^3_3 + c^3_2.P^2_3 + c^3_1.P^1_3 + c^3_0.P^0_3}$$
$$\mathbf{M_2 = c^2_3.P^3_2 + c^2_2.P^2_2 + c^2_1.P^1_2 + c^2_0.P^0_2}$$
$$\mathbf{M_1 = c^1_3.P^3_1 + c^1_2.P^2_1 + c^1_1.P^1_1 + c^1_0.P^0_1}$$
$$\mathbf{M_0 = c^0_3.P^3_0 + c^0_2.P^2_0 + c^0_1.P^1_0 + c^0_0.P^0_0}$$

The Quad Inproduct Pipeline Flow through:

$\mathbf{P^3_3=P^2_3,\ P^2_3=P^1_3,\ P^1_3=P^0_3,\ (P^0_3==Mb_3),}$
$\mathbf{P^3_2=P^2_2,\ P^2_2=P^1_2,\ P^1_2=P^0_2,\ (P^0_2==Mb_2),}$
$\mathbf{P^3_1=P^2_1,\ P^2_1=P^1_1,\ P^1_1=P^0_1,\ (P^0_1==Mb_1),}$
$\mathbf{P^3_0=P^2_0,\ P^2_0=P^1_0,\ P^1_0=P^0_0,\ (P^0_0==Mb_0),}$

7.4.4 8 bit Matrix functions: 8 bit Matrix Vector multiplication

M = matrixvec (Mb): Colour Space Conversion, DCT, iDCT, Interpolated Scaling and Affine Transform

$$M_3 = c^3{}_3.Mb_3 + c^3{}_2.Mb_2 + c^3{}_1.Mb_1 + c^3{}_0.Mb_0$$
$$M_2 = c^2{}_3.Mb_3 + c^2{}_2.Mb_2 + c^2{}_1.Mb_1 + c^2{}_0.Mb_0$$
$$M_1 = c^1{}_3.Mb_3 + c^1{}_2.Mb_2 + c^1{}_1.Mb_1 + c^1{}_0.Mb_0$$
$$M_0 = c^0{}_3.Mb_3 + c^0{}_2.Mb_2 + c^0{}_1.Mb_1 + c^0{}_0.Mb_0$$

7.4.5 8 bit Matrix functions: 8 bit Blend function

M = blend(Ma, Mb): Transparency, Non rectangular copies

$$M_3 = c^3{}_0.Ma_3 + c^3{}_1.Mb_3$$
$$M_2 = c^2{}_0.Ma_2 + c^2{}_1.Mb_2$$
$$M_1 = c^1{}_0.Ma_1 + c^1{}_1.Mb_1$$
$$M_0 = c^0{}_0.Ma_0 + c^0{}_1.Mb_0$$

Coefficients used during the blend operation

0	BLEND_CONSTANT	coefficient^3	coefficient^2	coefficient^1	coefficient^0
1	BLEND_ZERO	0.00	0.00	0.00	0.00
2	BLEND_ONE	1.00	1.00	1.00	1.00
3	SRC_COLOR	Ma_3/255,	Ma_2/255,	Ma_1/255,	Ma_0/255
4	INV_SRC_COLOR	1- Ma_3/255,	1-Ma_2/255,	1-Ma_1/255,	1-Ma_0/255
5	SRC_ALPHA	Ma_3/255,	Ma_3/255,	Ma_3/255,	Ma_3/255
6	INV_SRC_ALPHA	1- Ma_3/255,	1-Ma_3/255,	1-Ma_3/255,	1-Ma_3/255
7	DST_ALPHA	Mb_3/255,	Mb_3/255,	Mb_3/255,	Mb_3/255
8	INV_DST_ALPHA	1- Mb_3/255,	1-Mb_3/255,	1-Mb_3/255,	1-Mb_3/255
9	DST_COLOR	Mb_3/255,	Mb_2/255,	Mb_1/255,	Mb_0/255
10	INV_DST_COLOR	1- Mb_3/255,	1-Mb_2/255,	1-Mb_1/255,	1-Mb_0/255
11	SRC_ALPHA_SATURATE	1.00	alpha_sat	alpha_sat	alpha_sat
12	BOTH_SRC_ALPHA	source: SRC_ALPHA		destination:	INV_SRC_ALPHA
13	BOTH_INV_SRC_ALPHA	source: INV_SRC_ALPHA		destination:	SRC_ALPHA

alpha_sat = min (Ma_3/255, Mb_3/255)

7.4.6 Data Pipeline initialisation:

The loading of the MAC data pipeline to initialise matrix functions.

M = loadpipe(Ma, Mb)

Before the execution of matrix type functions, the internal data pipeline registers and or coefficient registers need to be filled. The **loadpipe** instruction services this purpose. No actions take place except for the loading of the coefficient registers and the pipeline registers. The action depends on the contents of the *Pipe* field of the MAC control register no 1. The output of the last stage of the pipeline is visible via control register cr21.

7.4.7 Accumulator file access

The accumulator ram file stores words which are twice as wide data sized used: 8 bit multiply results are accumulated in 16 bits, and 16 bit inputs in 32 bits and 32 bit multiply results become 64 bits. The **read_ram** and **write_ram** functions offer a facility to store and load these wider words from the accumulator ram file. The 64 bits accumulator register provides a wide data port to the accumulator ram file. The total accumulator file size is 128 words of 64 bits.

7.4.8 Reading data from the accumulator file

M = read_ram Data can be loaded directly from the accumulator file into the 64 bit accumulator register. The accumulator register is available as control register **MAC_Accu0** and **MAC_Accu1**. This function provides a facility to read the wider data words within the accumulator. The **read_ram** instruction takes five cycles like all MAC instructions. The actual transfer to the control register takes place at cycle 4 of the MAC instruction (which started at cycle 1). A block of data can be read from the accumulator at one cycle per read action if the delay is taken into account.

7.4.9 Writing data to the accumulator file

M = write_ram Data can be stored directly to the accumulator file from the 64 bit accumulator register. The accumulator registers are available as control register **MAC_Accu0** and **MAC_Accu1**. This function provides a facility to store the wider data words into the accumulator file. The **write_ram** instruction takes five cycles like all MAC instructions. The actual transfer to the accumulator Ram takes place at cycle 5 of the MAC instruction (which started at cycle 1). A block of data can be stored to the accumulator at one cycle per store action if the delay is taken into account.

7.4.10 Incremental Functions

M = linearstep The width of the accumulators is used for incremental calculations: a constant value is continuously added to a linear changing value. Second and higher order incremental calculations can be done in a multi step procedure (N+1 steps are needed for an Nth order interpolation, except for a linear interpolation which is a single step function). The quadruple 8 bit linearstep such as the colours in Gouraud shading can use the 16 bit double length accumulation to work with 8 bit accuracy behind the binary point. The Accumulator is used to add the accumulator register contents with an incremental value from the accumulator file.

7.4.11 The MAC functions: multiply accumulate (scalar)

M = macs(Ma, Mb) The multiply accumulate executes the latest executed multiply instruction from the basic set again and accumulates the result to the accumulator register. It does not use any information of the MAC control register.

7.4.12 The MAC functions: multiply accumulate (block)

M = macb(Ma, Mb) This instruction is a super set of the scalar multiply accumulate (macs(Ma, Mb)). It can read values from the accumulator file and write the results back again with optional incremented addresses. It uses the *write, read and accu* field from the MAC register.

7.4.13 16 bit vector product

M = vectprod(Ma, Mb) Mathematical definition:
The result M_H represents: the **internal or dot product** of the two vectors M_a and M_b where the H word corresponds with the X size and the L word corresponds with the Y size.
The result M_L represents: the **external or cross product** of the same two vectors.

$$M_H = Ma_H.Mb_H + Ma_L.Mb_L$$
$$M_L = Ma_H.Mb_L - Ma_L.Mb_H$$

7.4.14 16 bit complex product

M = complex(Ma, Mb) Mathematical definition:
The H words represent the **real** parts and the L words represent the **imaginary** parts in the operands as well as in the results. The ability to perform complex multiplications in a single cycle gives the Imagine excellent performance figures in many DSP tasks, most notably in Fast Fourier and other related Transformations

$$M_H = Ma_H.Mb_H - Ma_L.Mb_L$$
$$M_L = Ma_H.Mb_L + Ma_L.Mb_H$$

7.5 *Multiplier / accumulator operand formats.*

7.5.1 Multiplier input and output format definitions.

unsigned integer

8 bit	min 0	max 255
16 bit	min 0	max 65.535
32 bit	min 0	max 4.294.967.295

two's complement integer

8 bit	min -128	max +127
16 bit	min -32.768	max +32.767
32 bit	min -2.147.483.648	max +2.147.483.647

unsigned normalised fixed point

8 bit	min 0.000	max 255 / 256
16 bit	min 0.000	max 65.535 / 65.536
32 bit	min 0.000	max 4.294.967.295 / 4.294.967.296

two's complement normalised

8 bit	min -1.000	max +127 / 128
16 bit	min -1.000	max +32.767 / 32.768
32 bit	min -1.000	max +2.147.483.647 / 2.147.483.648

unsigned fixed point

8 bit	min 0.0	max 255 / 16
16 bit	min 0.0	max 65.535 / 256
32 bit	min 0.0	max 4.294.967.295 / 65.536

two's complement fixed point

8 bit	min -128 / 16	max +127 / 16
16 bit	min -32.768 / 256	max +32.767 / 256
32 bit	min -2.147.483.648 / ...	max +2.147.483.647 / 65.536

unsigned graphics data type

8 bit	min 0.000	max 255 / 255
16 bit	min 0.000	max 65.535 / 65.535
32 bit	min 0.000	max 4.294.967.295 / 4.294.967.295

two's complement graphics data type

8 bit	min -1.000	max +127 / 127
16 bit	min -1.000	max +32.767 / 32.767
32 bit	min -1.000	max +2.147.483.647 / 2.147.483.647

7.5.2 Internal format definitions

unsigned integer

8 → 16 bit	min 0	max 65.536
16 → 32 bit	min 0	max 4.294.967.295
32 → 64 bit	min 0	max 281.474.976.710.655

two's complement integer

8 → 16 bit	min -32.768	max +32.767
16 → 32 bit	min -2.147.483.648	max +2.147.483.647
32 → 64 bit	min -2^63	max +2^63 - 1

unsigned normalised fixed point

8 → 16 bit	min 0.000	max 65.535 / 65.536
16 → 32 bit	min 0.000	max 4.294.967.295 /4.294.967.296
32 → 64 bit	min 0.000	max (2^64 - 1) / (2^64)

two's complement normalised fixed point

8 → 16 bit	min -1.000	max +32.767 / 32.768
16 → 32 bit	min -1.000	max +2.147.483.647 / 2.147.483.648
32 → 64 bit	min -1.000	max +(2^63 - 1) / (2^63)

unsigned fixed point

8 → 16 bit	min 0.000	max 65.535 / 256
16 → 32 bit	min 0.000	max 4.294.967.295 / 65.536
32 → 64 bit	min 0.000	max (2^64 - 1) / (2^32)

two's complement fixed point

8 → 16 bit	min -2048 / 32	max +32.767 / 256
16 → 32 bit	min -8.388.608 / 2.048	max +2.147.483.647 / 65.536
32 → 64 bit	min -140.737.488.355.327 /.	max +(2^63 - 1) / (2^32)

unsigned graphics data type

8 → 16 bit	min 0.000	max 65.535 / 65.535
16 → 32 bit	min 0.000	max 4.294.967.295 /4.294.967.295
32 → 64 bit	min 0.000	max (2^64 - 1) / (2^64 - 1)

two's complement graphics data type

8 → 16 bit	min -1.000	max +32.767 / 32.767
16 → 32 bit	min -1.000	max +2.147.483.647 / 2.147.483.647
32 → 64 bit	min -1.000	max +(2^63 - 1) / (2^63 - 1)

7.6 The range clip unit

7.6.1 Operation

The Range control unit operates on the internal 64 bit results of the Multiplier/Accumulator. It compares the result with two values given by two 64 bit registers: **MAC_LoLimit0..1** (= cr22,cr23) and **MAC_HiLimit0..1** (= cr24,cr25) to check if the MAC output is within a predefined range. The values of MAC_LoLimit and MAC_HiLimit normally are 64 bit constants set via control register write operations. Alternatively they can be loaded with the values from **MAC_LoLim32** (= cr26) and **MAC_LoLim32** (= cr27). These 32 bit control registers contain both limits in 32 bit values compatible with the input and output format of the Multiplier / Accumulator and are expanded from 32 to 64 bit before stored in MAC_LoLimit and MAC_HiLimit. The third alternative is to load any or both 64 bit compare registers with the expanded Mb operand in order to obtain a variable limit.

7.6.2 Range clip activation

The Range Unit can be used by the extended multiplier operations and is activated by writing a logical '1' in the **RU** bit of control register **MAC_Control1** (= cr17). This bit activates the functions which are controlled by the Range control fields in control register **MAC_Control2** (= cr18)

7.6.3 Data size and data Type

The results of the MAC can have any of three data sizes, single 64 bit, double 32 bit and quad 16 bit. The results can be unsigned, signed and "balanced" signed. The Range Control Unit operates on all combinations of these types. The data type is always inherited from the MAC result. The compare registers have the same format as the MAC output: single 64 bit, double 32 bit or quad 16 bit. The compare operation provides four compare flags The result is **Inside** if the MAC output is higher or equal to the lower limit and lower or equal to the higher limit The result is **High** if it is higher than both range limits and it is **Low** if it is lower than both range limits. The result is **Wrong** if it is both higher than the higher limit and lower than the low range limit.

7.6.4 Range clip output

If the Range Controller is activated with the **RU** bit in **MAC_Control1** then it provides a range check on the value(s) from the Multiplier/Accumulator. If the Clip flag in **MAC_Control2** is a logical '1', then the MAC result is clipped to one of its limits. If the MAC output is too **High** it is replaced with the value in the high-limit register, if too **Low** it is replaced with the value from the low-limit register. If the result is **Wrong** then it always is replaced with the 'Higher limit'. If the Clip flag is '0' then the MAC output is passed unchanged.

7.6.5 The status word: ALU_RC_Status (=cr15)

The result of the comparisons is made available in the Status register of the Imagine: control register cr15. The Range Controller provides four of the eight flags assigned to each byte in the Status register:
If the condition is false then the flags are reset to logical '0'. Each of the four bytes in the Status words has eight flags: four from the ALU and four from the Range Controller.

Compare flags:

flags 4,12,20,28: MAC output **Inside** Range
flags 5,13,21,29: MAC output too **High**
flags 6,14,22,30: MAC output too **Low**
flags 7,15,23,31: MAC output **Wrong**

Status bits can be selected by operations in the Register File and independently in the Range Mask generator. These units select the four bits belonging to the same test (inside, higher, lower, wrong). The size of the result is taken into account during this selection. If the mode is quadruple 8 bit, then all the flags provided are independent and can all be different. In Single 32 bit mode all four fields will be identical (all four Inside flags, all four High flags etc.). In double 16 bit mode the highest two and the lowest two flags are identical.

ALU_RC_Status: cr15: Status flags from the ALU and Multipler / Accumulator

W3	L3	H3	I3	S3	C3	M3	Z3	W2	L2	H2	I2	S2	C2	M2	Z2	W1	L1	H1	I1	S1	C1	M1	Z1	W0	L0	H0	I0	S0	C0	M0	Z0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

7.6.6 The range mask generator

The Range Mask generator is another unit in the Imagine processor where selected range clip status flags can used to assemble a mask for masked vector writes to the Image memory. Four data lines and a strobe can transport the generated result to the Range mask generator each cycle. For each 16 bit word of the 64 bit result data, one of the four status flags is selected: Inside, Higher, Lower or Wrong.

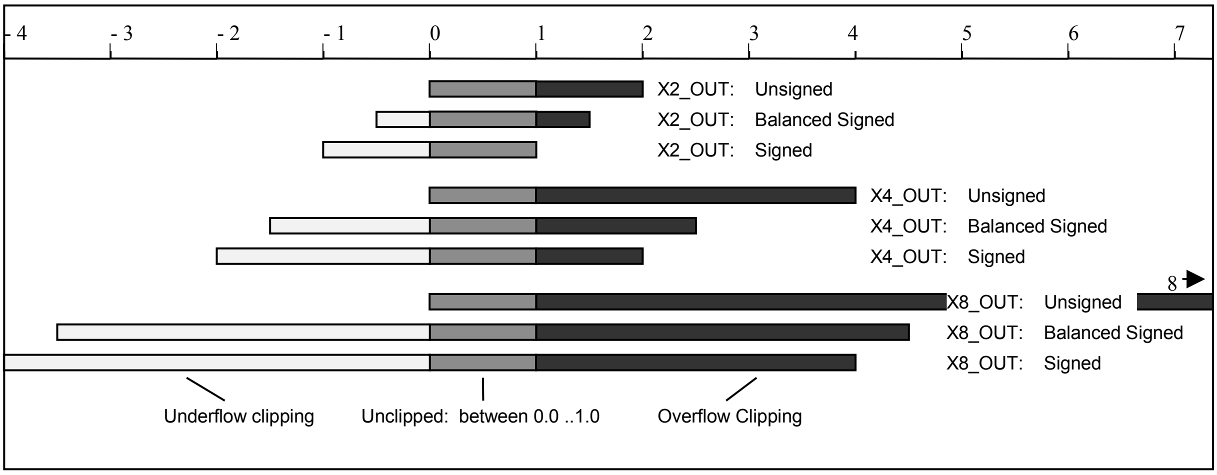
The flag selection is done with three bits from the **MAC_Control2** (cr18). Two bits select one of the four flags while the third bit can be used to invert the flag. The cycle which follows the comparison is used to transport the four selected values (one for each byte) to the Range Mask Generator. The activation of the Range Unit is the sign for the Mask generator to load the four flags into its Range Mask registers, Up to 64 results can be loaded in these registers. The Range Mask registers can be used as a (2D) mask for writing pixels into external Image Memory. A logical '1' is defined as write enable, a logical '0' as a write disable. The four bits are sent to the Range Mask generator in the cycle after the one which writes the contents of the M-bus register and the status register to the Range Unit.

7.6.7 Balanced signed compares

Balanced Signed compares can be used when there are only a small number of MSB bits available for over and underflow detection. This is the case with normalised numbers. The number of bits available depends on the Output shift factor: **X1_OUT**, **X2_OUT**, **X4_OUT** or **X8_OUT**. In these cases we have 0, 1, 2 or 3 bits available above the bits which will be placed on the M bus output of the Multiplier. Pixel values are given by one or more 8 bit values for grey scale or colour images. The 8 bit value represents an unsigned normalised fixed point value with a range from 0 to 1. Many calculations require multiplications with coefficients which can be both positive and negative. The result value will be in two's complement normalised fixed point format. This implies the need for a conversion from signed to unsigned representation which is handled during the output stage to the M bus register. (**X2_OUT)** This conversion is nothing more than a shift left by one position to shift out the sign bit. A negative result can be caused by negative overflow (underflow) but also by positive overflow.

A Balanced signed compare divides the area outside 0.0 and 1.0 in two equal parts for underflow and overflow. A signed number with one extra upper bit can represent values between -1.0 and +1.0. This is the default case for a signed normalised number. It can not detect overflow > 1.0. The balanced signed compare however can detect underflow between -0.5 and 0.0 and overflow between +1.0 and +1.5 Small negative numbers are considered to be the result of underflow while large negative numbers are considered to be caused by positive overflow. This method will correctly handle under- and overflows of up to 50%. Larger overflows and underflows can be handled by shifting out more bits to the left after the compares when converting the 4x16 bit intermediate values back to 4x8 bit values. The option **X2_OUT** performs a one bit shift appropriate for the sign conversion mentioned above while the options **X4_OUT** and **X8_OUT** shift out 2 and 3 bits. These options can be programmed in **MAC_Control1** (cr17). The balanced sign mode combined with **X4_OUT** will correctly handle under- and overflows of up to 150%. While The balanced sign mode combined with **X8_OUT** will correctly handle under- and overflows of up to 350%. The balanced signed compare approaches the normal signed compares when there are more and more MSB bits available. It becomes equal to signed compare for fixed and integer numbers.

Unsigned, Balanced signed and Signed mode ranges:



7.7 Overview of the multiplier control registers

cr16: **MAC_Mbus:** Multiplier Bus Registers

32 bit M bus result data (4x8, 2x16 or 1x32) [31:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr17: **MAC_Control1:** The Multiplier / Accumulator Control register

‘0’	AW [1:0]		‘0’	AR [1:0]		‘0’				MSIZE [1:0]		MTYPE [1:0]		MA sign	MB sign	‘0’	ACRMMU [2:0]			RN	SHIFT [2:0]			RU	‘0’	‘0’	PC	PP	PTT		
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr18: **MAC_Control2:** The Blending and Range Clip unit control register

‘00000000’								Blend Ma Coef [3:0]				Blend Mb Coef [3:0]				‘00000000’								SD	BS	ML	MH	Mask_sel			CL
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr19: **MAC_RamPtrs:**

BTYPE [1:0]		BA sign	BB sign	‘0000’				Coef Write Address [3:0]				Coef Read Address [3:0]				‘0’	Vector register ram write Address [6:0]							‘0’	Vector register ram read Address [6:0]						
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr20: **MAC_Coef:** Coefficient entry

8 x sign extension (8 x bit[23] when read)								8 higher coefficient bits [7:0]								8 bit coefficient value [7:0]								8 lower coefficient bits [7:0]							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr21: **MAC_Pipe:** Output of the 8 bit data pipeline

Pipeline register $\mathbf{P}^3_3$ [7:0]								Pipeline register $\mathbf{P}^3_2$ [7:0]								Pipeline register $\mathbf{P}^3_1$ [7:0]								Pipeline register $\mathbf{P}^3_0$ [7:0]							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr22,cr23: **MAC_LoLimit0, MAC_LoLimit1** 64 bit lower limit register

64 bit Low Limit Registers [63:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr24,cr25: **MAC_HiLimit0, MAC_HiLimit1,** 64 bit lower limit register

64 bit High Limit Registers [63:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr26,cr27: **MAC_LoLim32, MAC_HiLim32,** 32 bit limit registers

32 bit Low and High Limit Register [31:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr28,cr29: **MAC_Accu0, MAC_Accu1,** Accumulator register

Accumulator Register [63:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

7.8 Multiplier accumulator control register 1

cr17: MAC_Control1: The Multiplier / Accumulator Control register

'0'	AW [1:0]		'0'	AR [1:0]		'0'				MSIZE [1:0]		MTYPE [1:0]		MA sign	MB sign	'0'	ACRMMU [2:0]			RN	SHIFT [2:0]			RU	'0'	'0'	PC	PP	PTT		
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

7.8.1 The vector ram read / write control

These fields control reading and writing to the Accumulator vector RAM file. The read and write address are two independent seven bit fields in control register cr19. These address fields can be optionally post-incremented after a read or write access for vector processing.

7.8.2 The operand Data Size field

The Data Size field is set by the Operand Mb input if any of the basic multiplier operations is executed. A number of extended functions refer to this field for the Data Size of the operation.

7.8.3 The Data Type control field.

The Data Type field yields the same information as is provided by the 16 basic multiplication instructions. The four bits are set by the four bits from the multiplier function field in the instruction code of the basic operations. They select the multiplier operand type and select between unsigned, two's complement and mixed mode operation.

7.8.4 The Accumulator input selection

In all modes the adders have the choice of two out of three possible inputs: **AC:** The accumulator contents, **RM:** the read data the Ram and **MU:** The multiplier result. The result of the accumulation is stored into the accumulator register and can be written from there in to the Vector Ram. The **RN** bit will add ½ LSB to the multiplier result. (relative to the M bus output, for multiplication only)

7.8.5 Output shift factor

The Output selection takes 32 bits from the total of 64 bit Accumulator register for output on the M bus. The internal word size is twice the normal word size (4x8→4x16, 2x16→2x32, 32→64). The Data **type** determines the selection. Integer selects the lowest part of the result, Fixed point the middle part and Normalised the high bits. The output **shift** factor allows extended functions with extra selection options: x1, x2, x4 and x8 scaled output.

These options select the output bits from 0,1,2 or 3 bit positions lower in the 64 bit result data. This results in an extra scale factor of x1, x2, x4 or x8. The x2 scale factor can be generally used to shift out the sign (MSB) bit of signed normalised results and thereby converting them to unsigned normalised values.

AW: Accumulator Ram Write control
00: **WR_RAM_NOP:** Disable write
10: **WR_RAM:** Write to Ram
11: **WR_RAM_INCR:** Write, incr pointer

AR: Accumulator Ram Read control
00: **RD_RAM_NOP:** Disable Read
10: **RD_RAM:** Read from Ram
11: **RD_RAM_INCR:** Read, incr pointer

MSIZE: multiplier operand type:
00: **QUAD_BYTE**
01: **DOUBLE_SHORT**
10: **SINGLE_WORD**

MTYPE: multiplier operand type:
00: **INTEGER_FORMAT**
01: **NORMALISED_FORMAT**
10: **FIXED_PNT_FORMAT**
11: **GRAPHICS_FORMAT**

MA: Ma operand sign definition
0: **UNSIGNED**
1: **SIGNED**
MB: Mb operand sign definition
0: **UNSIGNED**
1: **SIGNED**

ACRMMU: Accumulator Inputs

000: **CLEAR_ACC**
001: **MULTIPLY**
010: **MAC_RAM**
011: **ADD_RAM_MULT**
100: **ACCUMULATOR**
101: **ACCUMULATE_MULT**
110: **ACCUMULATE_RAM**

RN: Round multiplication result
0: **NO_ROUND**
1: **ROUND**

SHIFT: Shift in output selections

100: **X1_OUT**
101: **X2_OUT**
110: **X4_OUT**
111: **X8_OUT**

The x4 and x8 options are useful to provide 1 or 2 bit extra for overflow /underflow testing. A 4x16 bit internal result may be defined as containing values between 0.000 to 8.000 or -4.000 to + 4.000 to allow larger over and underflows which can be detected and clamped by the Range Clip unit in the final stage of the Multiplier / Accumulator. A **X8_OUT** converts these values back to a 0.000 to 1.000 range.

7.8.6 The Range clip unit activation flag .

Enables or disables the Range Clip unit. The fields which control the behaviour of the range clip unit can be found in **MAC_Control2** (cr18)I

RU:	Range Unit activation flag
0:	**NO_CHECK**
1:	**RANGE_CHECK**

7.8.7 The pipeline control field

The pipeline control field controls the data transport through the 4x4 matrix registers The two sets of internal registers in the MAC are laid out in a 4x4 configuration (see the drawing on the next page). The data pipeline registers shift 4x8 bit data from the right to the left. The coefficient registers may shift 4x8 bit data from the bottom to the top for the 8 bit transpose (stippled arrows). Shifting is performed by the **loadpipe** function which uses the 'pipe' field in the MAC control register and by the **inproduct** function which shifts the data pipeline. The last data pipeline register is visible via control register cr21.

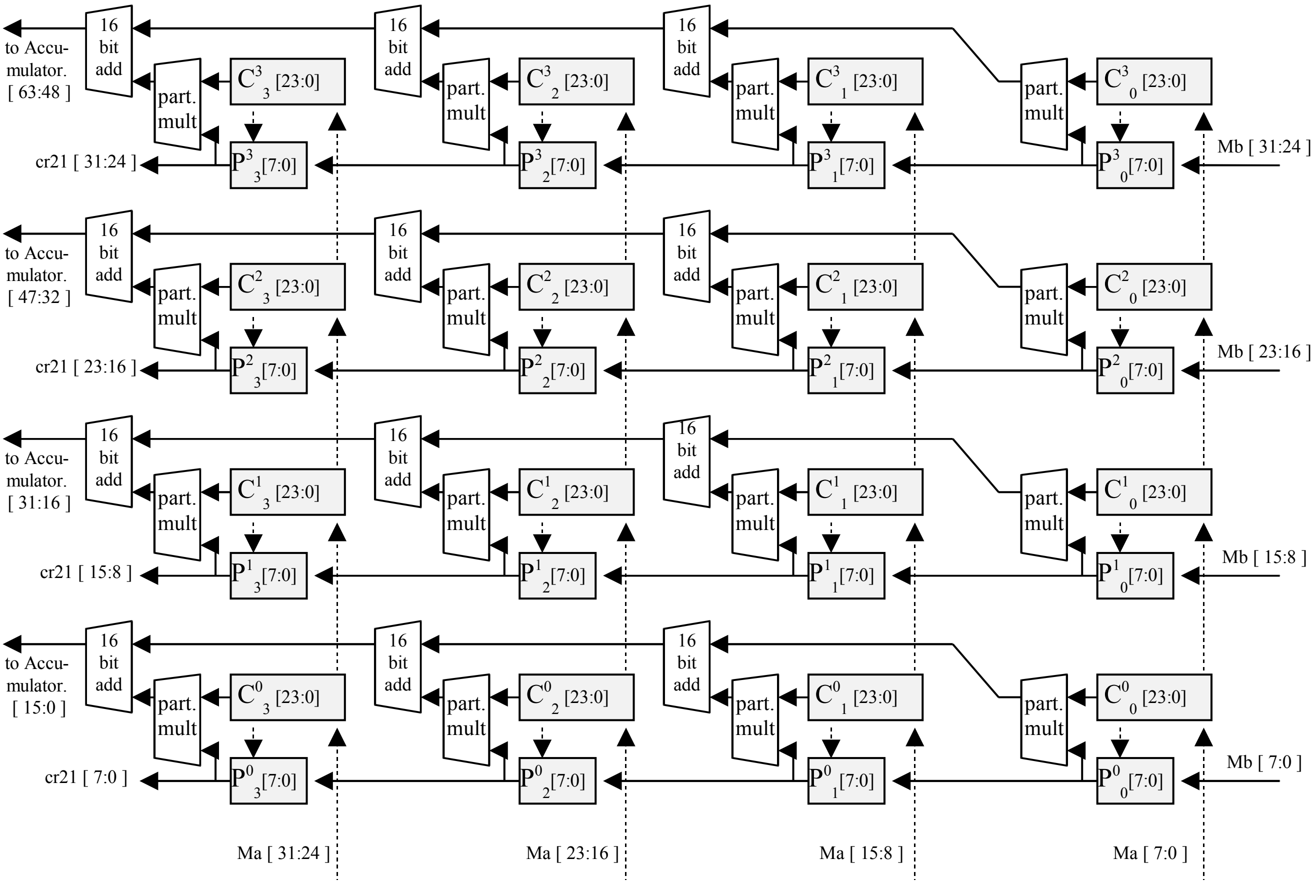
PC:	Coefficient registers
0:	**HOLD_COEF**
1:	**LOAD_COEF**

PP	8 bit Data Pipeline registers
0:	**HOLD_PIPE**
1:	**LOAD_PIPE**

PTT	Transpose control
000:	**RESET_TRANSPOSE**
100:	**SET_TRANSPOSE(0)**
101:	**SET_TRANSPOSE(1)**
110:	**SET_TRANSPOSE(2)**
111:	**SET_TRANSPOSE(3)**

7.8.8 Transposer operation

8 bit matrix transposition can be performed with the **loadpipe** function and the **PTT[2]** bit set to a logic '1'. Data is copied from the coefficient register set to the data pipeline register set once every four **loadpipe** operations. The two bit counter within the MAC control register is used in this function. The transfer takes place when the two bits **PTT[1:0]** are zero. These two bits are post incremented during a **loadpipe** if **PTT[2]** is logic '1'.



7.9 Multiplier accumulator control register 2

cr18: **MAC_Control2:** The Blending and Range Clip unit control register

‘00000000’								Blend Ma Coef [3:0]				Blend Mb Coef [3:0]				‘00000000’								SD	BS	ML	MH	MASK_SEL			CL
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

7.9.1 blend coefficient selection

Two 4 bit fields select the blending coefficients to be selected for the Ma and Mb input data of the multiplier. Options 0 through 10 implement all Open GL options while option 15 uses fixed coefficients

The blend coefficients

0000:	**BLEND_ZERO**
0001:	**BLEND_ONE**
0010:	**SRC_COLOR**
0011:	**ONE_MINUS_SRC_COLOR**
0100:	**DST_COLOR**
0101:	**ONE_MINUS_DST_COLOR**
0110:	**SRC_ALPHA**
0111:	**ONE_MINUS_SRC_ALPHA**
1000:	**DST_ALPHA**
1001:	**ONE_MINUS_DST_ALPHA**
1010:	**SRC_ALPHA_SATURATE**
1111:	**BLEND_CONSTANT**

7.9.2 range unit: 32 or 64 bit compares

The SD flag selects between the 32 bit limit registers and the 64 bit limit registers. The 32 bit options expands the 32 limit registers to 64 bit according to the used data **Size,** the selected data **Type**, and the selected **Shift** value. The expansion is the inverse operation of the 64 → 32 selection at the end of the multiplier before the result is placed in the 32 bit M bus register. The expanded results are placed in the 64 bit Limit registers.

SD: Single / Double width Compares

0:	**COMPARE_32**
1:	**COMPARE_64**

7.9.3 range unit:

Balanced signed compare:

Balanced signed compares divide the overflow and underflow range in two equal parts in cases where there are only few MSB bits available for overflow and underflow detection. Which is the case for normalised format operations. The three cases below have 1, 2 and 3 bits for overflow and underflow detection:
-signed normalised format + X2_OUT: → +/- 50% range
-signed normalised format + X4_OUT: → +/- 150% range
-signed normalised format + X8_OUT: → +/- 350% range
Balanced signed compares approaches normal signed compares when there are more and more MSB bits available. Is equal to signed compares for Fixed and Integer operations.

BS: Ballanced Signed Compares

0:	**UNBALANCED**
1:	**BALANCED**

7.9.4 range unit:

Dynamic Limits

The 64 bit Limit registers can be dynamically loaded with the Mb input data which is expanded to 64 bits first. Both Limit registers are individually controlled by **ML** and **MH**.

ML MH: Mb operand to Limit registers

00:	**HOLD_LIMITS**
01:	**LOAD_HILIMIT**
10:	**LOAD_LOLIMIT**
11:	**LOAD_LIMITS**

7.9.5 range unit:

Range Mask selection

The results of the Compares are stored in the **ALU_RC_Status** control register (cr15). The field **MASK_SEL**

MASK_SEL: Status Flags → Mask Generator

000:	**RANGE_INSIDE**
001:	**RANGE_HIGHER**
010:	**RANGE_LOWER**
011:	**RANGE_WRONG**
100:	**RANGE_NOT_INSIDE**
101:	**RANGE_NOT_HIGHER**
110:	**RANGE_NOT_LOWER**
111:	**RANGE_NOT_WRONG**

selects the four status bits which are send to the Mask Generator where they can be assembled into the 64x4 bit Range Mask.

7.9.6 range unit:

Output clipping

The CL flag determines if the output is either passed unmodified to the M bus output register or that is clipped to the Higher or Lower Limits if it is to large or to small.

CL: Clip MAC output data

0: **NO_CLIP**
1: **CLIP**

7.10 Multiplier accumulator pointer control register

This control register contains read and write pointers for the Vector register Ram and the coefficient registers. Most extended functions may access the vector register ram. These registers are typically used for vector accumulation, temporary vector storage and differential engine operations. The coefficients are used for three different 8 bit functions: **inproduct(), matrixvec()** and **blend().** These functions support convolution, color space conversion, YUV to RGB conversion, discrete cosine transformations, bicubic scaling, blending, mixing et cetera. The data words are 8 bit in these operations but the coefficient must be more accurate. The 16 coefficients (4x4) of the multiplier provide 16 bit accuracy.

cr19: MAC_RamPtrs:

BTYPE [1:0]		BA sign	BB sign	'0000'				Coef Write Address [3:0]				Coef Read Address [3:0]				'0'	Vector register ram write Address [6:0]							'0'	Vector register ram read Address [6:0]						
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

7.10.1 Vector register ram read and write pointers

The vector register is accessed in parallel to an extended multiplier operation. The read data will arrive at the same time by the accumulator as the multiplication result. The result of the accumulation will be written to the write address. Both pointers can work in post-increment mode to support vector operations.

Typical usage: vector accumulation:
Several vectors are accumulated into one vector. An operation is used for convolution (filtering), correlation, alpha blending, et cetera. Example: A 3x3 convolution is accomplished by two vector reads and one vector read/ write. The vectors undergo the **M = inproduct()** function and are accumulated in the Vector register ram. The resultant vector is written back to image memory. The result vector contains 64 x 4 = 256 pixels. The whole operation takes 1.2 microseconds at 200 MHz. The range clipper is used to clip the pixels into the right range.

7.10.2 Coefficient read and write pointers

The coefficients used in for example the convolution operations mentioned above should be written via control register **MAC_Coef** (cr20) to the right locations given by the coefficient read and write pointers. These pointers always operate in post-increment mode. The read and write order is:
$c^0_0 \rightarrow c^0_1 \rightarrow c^0_2 \rightarrow c^0_3 \rightarrow c^1_0 \rightarrow c^1_1 \rightarrow c^1_2 \rightarrow c^1_3 \rightarrow c^2_0 \rightarrow c^2_1 \rightarrow c^2_2 \rightarrow c^2_3 \rightarrow c^3_0 \rightarrow c^3_1 \rightarrow c^3_2 \rightarrow c^3_3 \rightarrow c^0_0 \rightarrow .....$

7.10.3 The data type and signs used for macs()

The simplest extended function is **macs()** which is used to sum a number of consecutive multiplier results. The data type and signs given in the start multiplication are store in BTYPE, BA and BB and used for the following **macs()** operations. These fields are updated by every basic multiply operation and are used exclusively by the function **macs().** Example:

```
AB = rd(r20,r30) –>  mult ( A,B, iss ) ;        // " iss " is stored in cr19
AB = rd(r21,r31) –>  macs( A,B ) ;
AB = rd(r22,r32) –>  macs( A,B ) ;
AB = rd(r23,r33) –>  macs( A,B ) ;
AB = rd(r24,r34) –>  macs( A,B ) –––––> wr( r40)
```

7.11 Multiplier accumulator coefficient register entry

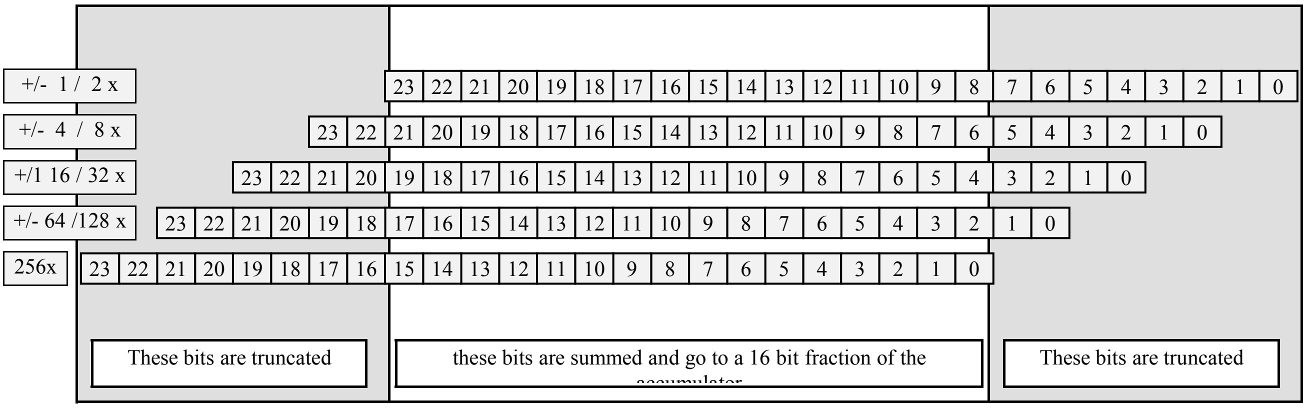
The coefficients are used for various 8 bit functions: **inproduct(), matrixvec()** and **blend().** The coefficients are 24 bit. The sign bit (bit 23) is extended to bits [31:24] when a coefficient is read back. The functions mentioned above are used for convolution, color space conversion, YUV to RGB conversion, discrete cosine transformations, bicubic scaling, blending, mixing et cetera. The data words are 8 bit in these operations but the coefficient must be more accurate. The 16 coefficients (4x4) of the multiplier provide more bits accuracy:

cr20: **MAC_Coef:** Coefficient entry

<table>
<tr><td colspan="8">8 x sign extension
(8 x bit[23] when read)</td><td colspan="8">8 higher coefficient bits
[7:0]</td><td colspan="8">8 bit coefficient value
[7:0]</td><td colspan="8">8 lower coefficient bits
[7:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

The drawing below shows which bits are significant in the partial 24x8 bit multiplication and how they result in a 16 bit value which goes to a 16 bit fraction of the accumulator. (The 16 bit shown here are summed together with three similar 16 bit results and this result then goes to the 16 bit accumulator fraction)

(This drawing will be removed later)



The coefficients used in for example the convolution operations mentioned above should be written via control register **MAC_Coef** (cr20) to the right locations given by the coefficient read and write pointers. These pointers always operate in post-increment mode. The read and write order is:

$\mathbf{c^0_0 \rightarrow c^0_1 \rightarrow c^0_2 \rightarrow c^0_3 \rightarrow c^1_0 \rightarrow c^1_1 \rightarrow c^1_2 \rightarrow c^1_3 \rightarrow c^2_0 \rightarrow c^2_1 \rightarrow c^2_2 \rightarrow c^2_3 \rightarrow c^3_0 \rightarrow c^3_1 \rightarrow c^3_2 \rightarrow c^3_3 \rightarrow c^0_0 \rightarrow .....}$

7.12 Multiplier accumulator 8 bit data pipeline output

The last four bytes of the 8 bit 4x4 data pipeline are visible as an output in this control register. Usage is typically a transpose or delay operation.

cr21: **MAC_Pipe:** Output of the 8 bit data pipeline

<table>
<tr><td colspan="8">Pipeline register $\mathbf{P^3_3}$
[7:0]</td><td colspan="8">Pipeline register $\mathbf{P^3_2}$
[7:0]</td><td colspan="8">Pipeline register $\mathbf{P^3_1}$
[7:0]</td><td colspan="8">Pipeline register $\mathbf{P^3_0}$
[7:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

7.13 The state save and restore register

This single 32 bit register should be saved an later restored during interrupts it can be read and written like any normal control register. **(MAC_Save, cr30)**

Chapter

8. UNARY FUNCTION UNIT

The Unary Function Unit

The UFU performs various single operand functions. Besides these functions found in the basic set, an extended set is supplied for IEEE 754 single precision floating point conversions. Fixed point and integer conversions as well as range checking can be done over the entire dynamic range as defined by the 32 bit floating point standard.

fig. unary function unit

8.1 UNARY FUNCTION UNIT

The basic Unary functions except the IEEE-754 functions work on a single 32 bit word, double 16 bit words or quadruple 8 bit words. The wordsize is inherited from the source of the operand and passed to the destination functional unit together with the result on the U-bus. Four basic operations can be applied on both the A bus and the F bus. The operand size which is used in these functions is inherited from the selected source bus (A bus or F bus).

The conversion functions handle IEEE 754 single precision floating point to integer/fixed point conversion and vice versa. It supports the exponent handling and error detection of basic operations like addition, subtraction and multiplications. All five IEEE 754 32 bit float point types are supported: normalised, denormalised, zero, +/- infinity and "not_a_number".

8.1.1 The result register of the UFU

The results of the Unary function unit are available in the U bus register which can be used by other functional units, The register file or the I/O units This register is also accessible as a control register (UFU_Ubus, cr32)

Cr32: **UFU_Ubus:** Unary Function Unit Bus Registers

<table>
<tr><td colspan="32">32 bit U bus result data (4x8, 2x16 or 1x32)
[31:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

8.1.2 The instructions of the UFU

The four bit field in the instruction word decodes the following 16 different instructions:

IC 62:59	Mnemonics	function	cycles	size
0	**U = noop**	no operation	1	Asz
1	**U = pass(A)**	pass value, init IEEE conversions	1	Asz
2	**U = unary(A)**	binary to unary conversion	1	Asz
3	**U = binary(A)**	unary to binary conversion	1	Asz
4	**U = integer(Ad)**	IEEE float 32 → integer conversion	2	32
5	**U = fixed(Ad)**	IEEE float 32 → fixed point conversion	2	32
6	**U = float(Ad)**	integer → IEEE float 32 conversion	2	32
7	**U = floatFP(Ad)**	fixed point → IEEE float 32 conversion	2	32
8	**U = abs(A)**	absolute value of A	1	Asz
9	**U = sign(A)**	sign of A (A<0: U=-1, A=0: U=0, A>0: U=1)	1	Asz
A	**U = notzero(A)**	set bits if A != 0	1	Asz
B	**U = swap(A)**	swap bits (A31->U0, A30->U1, A29->U2,...)	1	Asz
C	**U = abs(F)**	absolute value of F	1	Fsz
D	**U = sign(F)**	sign of F (F<0: U=-1, F=0: U=0, F>0: U=1)	1	Fsz
E	**U = notzero(F)**	set bits if F != 0	1	Fsz
F	**U = swap(F)**	swap bits (F31->U0, F30->U1, F29->U2,...)	1	Fsz

8.2 The basic unary functions

The following functions all operate on the three basic formats of the Imagine processor: single 32 bit words, double 16 bit words and quadruple 8 bit words. They are all executed within a single cycle.

8.2.1 Binary to Unary conversion: **U = unary(A)**

This function expects 1 ,2 or 4 two's complement number(s) and converts them to a unary representation: negative numbers always result into 0, and a number equal to and larger than the number of bits in a word (8, 16, 32) always results into "all 1s".

U = unary(A)

-1 →	00000000	3 →	00000111
0 →	00000000	4 →	00001111
1 →	00000001	5 →	00011111
2 →	00000011	6 →	01111111
		et cetera.	

8.2.2 Unary to Binary conversion: **U = binary(A)** (priority encoder)

This function expects 1, 2 or 4 unsigned value(s) and returns the position(s) with the first non-zero bit. It is the inverse function of the Binary to Unary conversion. The lowest result value is 0 and the highest result value is the number of bits in a word,
(8, 16 or 32).

U = binary(A)

00000000 → 0	00001xxx → 4
00000001 → 1	0001xxxx → 5
0000001x → 2	001xxxxx → 6
000001xx → 3	01xxxxxx → 7
	et cetera.

8.2.3

Absolute value: **U = abs(A), U = abs(F)**

The Absolute value function expects 1, 2 or 4 two's complement numbers. It returns the absolute value(s) of these number(s). An exceptional case are the maximal negative values which do not have a corresponding positive value. These will map to themselves.

8.2.4 Sign function: **U = sign(A), U = sign(F)**

This function expects 1, 2 or 4 two's complement number(s) and returns +1, 0 or -1 depending on the sign and the zero test.

8.2.5 Not zero function: **U = notzero(A), U = notzero(F)**

Returns 00, 0000 or 00000000 in case of X=zero and FF,FFFF,FFFFFFFF in case of X=not zero, depending on the wordsize.

8.2.6 Swap bits function: **U = swap(A), U = swap(F)**

Swap bit reverses the bit order of the bits in an 8, 16 or 32 bit word.
msb → lsb, msb$_{-1}$ → lsb$_{+1}$, msb$_{-2}$ → lsb$_{+2}$, ... , lsb → msb.
The highest bit will end up in the lowest place and the lowest will end up in the highest place. This operation is useful for bitmap and bitmask operations and FFT address calculation.

8.3 IEEE 754 floating point operations

8.3.1 Handling of floating point numbers:

The Imagine handles the 32 bit IEEE-754 floating-point operations with the aid of a small specialised conversion unit and it's standard integer ALU and Multiplier. This method proves to be only 1.5 to 2 times slower for general C programs compared to costly pipelined floating point hardware. A typical mix of instructions shows a mean execution time of 10 cycles/ floating point operation. The majority of C programs contain in general to much data dependencies to be handled efficiently by pipelined floating point hardware. The omission of pipelined floating point hardware in the Imagine however only applies to this version and is based purely on economy considerations.

The small conversion unit handles the right conversion from and to floating point numbers. It checks for floating point exceptions like overflow, underflow, not_a_number and it handles the exponent calculations for addition, subtraction and multiplication. It is used also to implement some more elaborated floating point operations like the 3D homogeneous coordinate transformation in a very efficient way. The basic floating point operations take the form of small macro routines which can either be called by, or included within, the program. The operations support all 5 formats defined by the IEEE-754 floating point standard.

8.3.2 IEEE 754 32 bit floating point definition

	sign	exponent	mantisse	value
format 1: not a number:	don't care	255	not 0	not a number
format 2: +/- infinity:	+ or -	255	0	+/- infinity
format 3: normal number:	+ or -	0 < exp < 255	0..7FFFFF	$(-1)^{sign}x2^{exp-127}x\ 1.mant$
format 4: very small number:	+ or -	0	0..7FFFFF	$(-1)^{sign}x2^{exp-126}x\ 0.mant$
format 5: zero:	+ or -	0	0	+/- 0

8.3.3 IEEE 754 32 bit floating point macro functions

32 bit Floating Point Macro functions

Mnemonics	**operation**	**→ result**	**cycles**
int_sf()	float	→ integer	1
float_sf()	integer to float	→ float	1
abs_sf()	absolute value	→ float	2
neg_sf()	negate	→ float	2
add_sf()	float + float	→ float	9
addint()	float + int	→ float	10
add3_sf()	float + float + float	→ float	11
sub_sf()	float - float	→ float	9
subint_sf()	float - int	→ float	10
rsubint_sf()	int - float	→ float	10
mul_sf()	float x float	→ float	12
mulint_sf()	float x int	→ float	13
mul3_sf()	float x float x float	→ float	17
div_sf()	float / float	→ float	26
divint_sf()	float / int	→ float	27
rdivint_sf()	int / float	→ float	26
matrix_4x4_sf()	full 4x4 matrix times vector multiplication	→ float	30
homogenous_tr_sf()	homogenous transform + perspective division	→ float	60

8.4 IEEE 754 floating point operation support register cr33

Cr33: UFU_IEEE floating point operation control register

ERR	OVF	UNF	NAN	MUL	UH	EX	EL	H exponent [7:0]								Fix to Float offset [7:0]								Float to Fix offset [7:0]							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

8.4.1 Float To Fix offset. cr33 [7:0]

Used in the function U = fixed(A). This offset either is added to the floating point exponent before conversion (EL='0') or replaces the exponent before conversion, (EL=='1').

8.4.2 Fix To Float offset. cr33 [15:8]

Used in the function U = floatFP(A). This offset either is added to the floating point exponent after conversion (EX='0') or replaces the exponent after conversion, (EX='1'). Details can be found in the examples on the following pages.

8.4.3 The H exponent. cr33 [23:16]

The H(idden) exponent is used to calculate the exponent during addition, subtraction and multiplication.

Addition/ Subtraction: The H exponent is replaced whenever one of the functions, U = pass(A), U = integer(A) or U = fixed(A) is performed and the value in the exponent field of the A data (A23..A30) is larger than the current highest exponent.

Multiplication: The H exponent is calculated with: H exponent = H exponent + A_bus[31:23] - 127

This action takes place during the functions U = pass(A), U = integer(A) and U = fixed(A).

8.4.4 EL: exponent offset usage in U = fixed() (see the Float to Fix offset)

8.4.5 EX: exponent offset usage in U = floatFd() (see the Fix to Float offset)

8.4.6

UH: Use H exponent

If this flag is true ('1'), the H Exponent field is used for the calculation of the exponent during floating point addition, subtraction and multiplication.

8.4.7 MUL: Use H exponent for add or multiply

If this flag is true ('1'), the H Exponent field is used for the calculation of the exponent of a product of 2 or more floating point values. otherwise it is used to calculate the sum (subtraction) of 2 or more floating point values.

8.4.8 NAN: Not a Number error flag

This error flag is set together with the ERR flag if a IEEE NAN value (Not A Number) is converted from float to integer or fixed.

8.4.9 UNF: Underflow error flag

This error flag is set together with the ERR flag if a fixed number is converted into a floating point value which is smaller then the smallest representable floating point value, with the exception of the integer value 0.

It is set without the ERR flag as a warning only in case of a float to integer or fixed value which is smaller the smallest representable integer value.

8.4.10 OVF: Overflow error flag

The overflow flag is set together with the ERR flag if a overflow occurs during any of the conversions.

8.4.11 ERR: Floating point error flag

The error flag is set whenever an error occurs in any of the conversions. This value is one cycle later visible in the sequencer status and control register where it can be tested for conditional jumps, calls, returns etc.

8.5 IEEE-754 floating point conversions

8.5.1 The pass instruction

U = pass(A)
The value of the A bus is passed to the U bus register and is available on the U bus one cycle later. The operand size from the A bus is passed unchanged to the U bus. The pass operation has one side effect: The A bus bits 23 to 30 represent the biased exponent in IEEE 754 single precision floating point numbers. These bits are compared with the contents of the *highest exponent* field from control register cr33. When they are higher the '*highest exponent' is replaced by the exponent of A*.

8.5.2 The IEEE 754 conversion instructions

U = integer(A)
U = fixed(A)
U = float(A)
U = floatFP(A)

These instructions take two cycles before they produce their result. They operate pipelined so a new conversion function may be launched each cycle The first cycle leaves the U bus register unchanged. The second cycle outputs the result via the U bus register. (It overwrites the result of any 1 cycle function executed in the same cycle) The first cycle of the conversion function is non interruptable which means that an interrupt service routine does not need to save the internal state of this unit

8.5.3 IEEE 32 bit floating point to integer

U = integer(A)
The function performs IEEE floating point to integer conversion. It will produce the right results for Normalised Numbers, the Normalised Zero. The so called NANs (Not A Numbers) are not supported.

8.5.4

IEEE 32 bit floating point to fixed

U = fixed(A)
This instruction is a superset of the U = integer(Ad) instruction. An 8 bit two's complements offset in register UFU_IEEE (cr33) is added to the exponent before the conversion takes place. This function can map an arbitrary floating point value into a useful fixed point range. The offset may also replace the exponent itself (when cr33 [24]: EL='1') (see the examples). If the UH flag is set, then the H exponent is as the exponent for conversion

8.5.5 Integer to IEEE 32 bit floating point

U = float(A)
Integer to Floating point conversion. The result is a Normalised floating point number as defined on the following page.

8.5.6 Fixed to IEEE 32 bit floating point

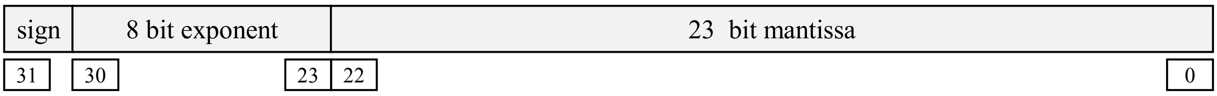
U = floatFP(A)
This instruction is a superset of the U = float(Ad) instruction. An 8 bit two's complement offset in control register UFU_IEEE (cr33) is added to the biased exponent at the end of the conversion. The offset can also replace the exponent (when cr33 [25]: 'EX' = '1')
(see the examples). If the UH flag is set, then the H exponent is added to the exponent of the resultant floating point value.

8.5.7 Some examples of floating point to integer conversions

- ♦ IEEE 32 bit floating point to integer.
- ♦ IEEE 32 bit floating point to fixed point with a programmable offset.
- ♦ IEEE 32 bit floating point to fixed point with a programmable offset without the exponent.

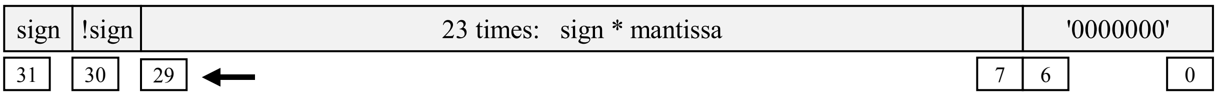
IEEE 32 bit floating point format

Numerical value of the floating point representation = $(-1)^{sign}.(0.1mantissa).2^{exp-126}$
Numerical value of the floating point representation = 0 if exponent = 0 and mantissa = 0
NANs (Not A Number) are not supported

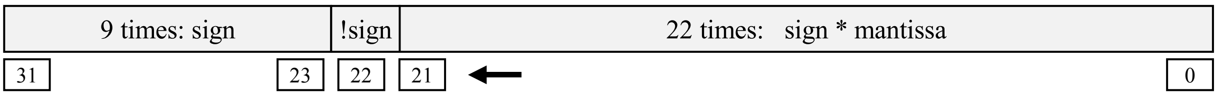


Floating point to integer / fixed point conversions (3 options):

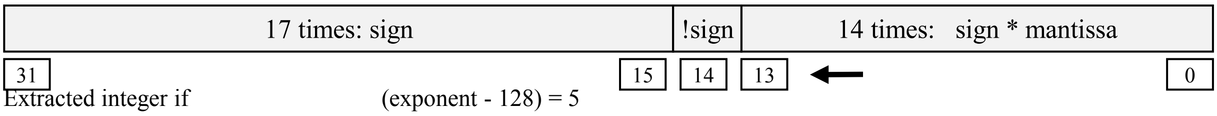
Extracted integer if	(exponent - 128) = 29
Extracted fixed point number if	(exponent - 128 + offset) = 29
Extracted fixed point number if	(offset) = 29



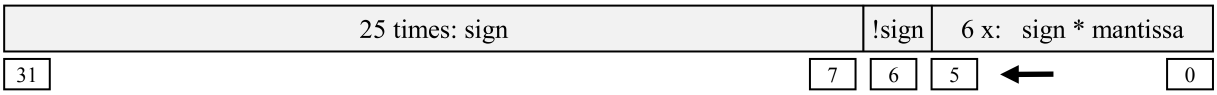
Extracted integer if	(exponent - 128) = 21
Extracted fixed point number if	(exponent - 128 + offset) = 21
Extracted fixed point number if	(offset) = 21



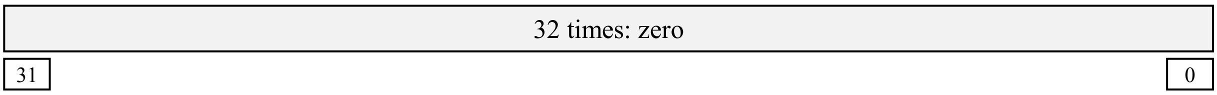
Extracted integer if	(exponent - 128) = 13
Extracted fixed point number if	(exponent - 128 + offset) = 13
Extracted fixed point number if	(offset) = 13



Extracted integer if	(exponent - 128) = 5
Extracted fixed point number if	(exponent - 128 + offset) = 5
Extracted fixed point number if	(offset) = 5



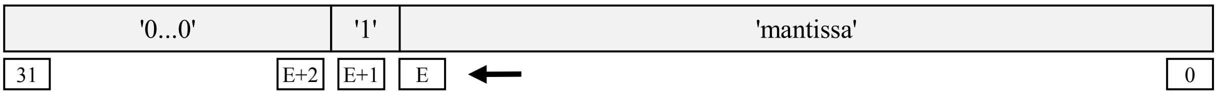
Extracted integer / fixed point if (exponent == 0 and mantissa == 0)



The Fixed point to floating point conversions (3 options):

The absolute value of an input number, shown below in the drawing, will be converted to:

S = sign	mantissa = 'MANTISSA' << (22-**E**)	exp = **E** + 128
S = sign	mantissa = 'MANTISSA' << (22-**E**)	exp = **E** + 128 + offset
S = sign	mantissa = 'MANTISSA' << (22-**E**)	exp = 128 + offset



The value of zero will in all three conversions be converted to: S = '0', mantissa = '0', exp = 0

Chapter

9. DATA I/O UNIT

The Data I/O unit handles the 32 bit bidirectional databus to perform load and store accesses to memory via the Data Cache or the Internal Peripheral Bus.
The Imagine Data I/O unit can perform accesses to bytes, 16 bit shorts and 32 bit words, both signed and unsigned. It supports linear addressing, with optional post and pre address increment or decrement, 2D addressing and 3D addressing.

The Data I/O unit is closely coupled with the 3D graphics pipeline to read perspective corrected textures fig. data i/o unit

9.1 general

The Data I/O unit handles random accesses to memory via the Data Cache or the Internal peripheral bus. It is split into a Data Access Unit which handles data addresses and a Data Transfer Unit which handles data Loads and Stores. It provides all access mechanisms needed for C generated code as well as special graphics and image processing functions. The Imagine can perform accesses to bytes, 16 bit shorts and 32 bit words, signed and unsigned. Words are always aligned to addresses with the two lowest address bits zero, and shorts are always aligned to even addresses. (The alignment hardware ignores the address bits.). There is direct support for 2D and 3D memory access and memory organisation. Both cache and the memory paging system is optimised for 2D and 3D pixel accesses in the associated memory modes.

9.1.1 Data memory organisation

The Data I/O unit can access all external memory (SGRAM or SDRAM) memory via the Cache. It can view memory as linear, 2 dimensional (8 modes) or 3 dimensional (8 modes). The eight 2 dimensional modes differ in the numbers of "bytes per row" which varies from 256, 512, ...32768. The eight 3 dimensional modes provides a selection of useful volumes with different X, Y and Z sizes.

There are sixteen different banks each of which can have a programmable dimension: 1D, 2D or 3D. This effectively means that memory is structured to optimise accesses for a certain data type. 2D structured memory accesses are single cycle within a certain rectangle but incur a penalty if the rectangle's border is crossed. 3D structured memory accesses are single cycle within a 3D volume but incur a penalty if the volume's boundaries are crossed.

The IPB bus is located in the high end of the 32 bit memory space. The internal Multi Media I/O units are located in I/O space 0. External I/O units which are connected to the 8 bit external peripheral bus are located in I/O space 1. The external EPROM which is also connected to the external peripheral bus can be accessed via I/O space 3.

Description	Access ID or I/O space	Address offset	Address range	organisation	Access types	memory page size
Programmable memory: Sixteen banks of 16 Mb each with programmable dimension (1D,2D or3D) and Access ID	ID = 0..7	0000.0000 0100.0000 . . . . 0e00.0000 0f00.0000	16 Mb	2D or 3D	1D, 2D, 3D	64k byte
IPB: multi media units	I/O space[0]	f000.0000	64 kb	linear	1D	none
IPB: external I/O	I/O space[1]	f001.0000	64 kb	linear	1D	none
IPB: external EPROM	I/O space[3]	f002.0000	128 kb	linear	1D	none

9.1.2 Data memory address types

The Data Memory address can be linear: byte oriented, internally 32 bit. All memory banks can be accessed via linear addresses independent of the are structured in a 1D, 2D or 3D way. The Data I/O unit also accepts 2D and 3D addresses which are provide with 2x16 bit words (2D) and 4x8 bit words (3D)

9.1.3 Internal data representation

The Imagine has three basic internal data formats: single 32 bit word, double 16 bit word and quad 8 bit word. These are all stored and loaded as 32 bit words in the Data memory and are stored on aligned addresses.
(The Vector I/O unit memory *can* do single cycle non-aligned accesses in vector mode.) Bytes and half words exist in memory. Inside the Imagine they are converted from and to 32 bits words during load and store operations to avoid explicit conversions in mixed mode operations and to consistently use status flags for conditional branching and calling. A byte loaded from external memory is loaded internally in the 8 least significant bits. An unsigned byte has the highest 24 bits set to zero while a signed byte has the highest 24 bits set to its sign bit. A half word from external memory is loaded in internally in the 16 least significant bits. The highest 16 bit are set to zero or one depending on the word type (signed/unsigned) and the sign bit itself.

9.2 Data Access function

The unit which controls the access of external data devices (address output) is relatively independent of the unit which handles the data transfers. It is controlled by a three bit field in the instruction code (Ic54..56) and can perform eight address operations. It selects an address from one of three internal busses and defines if the access is a READ or a WRITE access. The actual access will take place in the next cycle.

The Address can be taken from either the A bus, the F bus or the M bus. The A bus can provide an address directly from a register. The F bus provides the calculated addresses if one or more address pointers and offsets need to be added together.

The addresses provided by the M bus are more likely to be 2D or 3D addresses. 2D addresses are formed by a 2x16 bit word (Y,X) and 3D addresses are formed by a 4x8 word (Z,Y,X). The lower three bytes provide the co-ordinates. These accesses use the cache to archive a better performance

The **DA = extended** operation provides extra access functionality. It enables the use of the 3D graphics pipeline and supports auto increment and auto decrement modes for vector accesses. Both Post and Pre increment/decrement is supported

9.2.1 The use of the 3D graphics pipeline

The application of the **DA = extended** function with the **Use_PAG** flag set (cr37 bit 30: use Perspective Address generator) enables the use of the 3D graphics pipeline to generate perspective correct 1D, 2D or 3D read addresses into external memory. These modes bypass the cache and have their own highly optimised interface with the external memory controller. These modes can effectively load up to 4 pixels (texels) per cycle needed for bilinear, trilinear and quad linear interpolation. This mode also supports a wide range of texel format translations from 1 to 8 bit pseudo colors and 16 bit colors to 32 bit true color αRGB.

9.3 The Data transport function

The Unit which controls the data input and output transfers is controlled by a three bit field in the instruction word (Ic53..Ic51). The Unit can perform Loads and Stores from bytes, half words (16 bit), and words (signed and unsigned).

9.3.1 The data store functions

The STORE functions transfer internal data to an external device: the data memory or an I/O port. The STORE function is given in the same cycle as the data write access function. Otherwise it is not recognised as a STORE function.

The data is placed into a register (D bus register), from where it is written to the cache and simultaneously send to the memory write buffer. Byte and half words are aligned to the right byte positions depending on the two lowest address bits, (DA0,DA1). The four byte write enable lines (WR0*..WR*3) take care that only those bytes are modified which contain the byte or half word data. Examples:

DA = wrAd(A), D = byte(B);
DA = wrAd(F), D = word(M);
DA = Again, D = short(B);

The Data Store functions

53:51	**Mnemonic**	**Data transfer operation**	**size**
0	**D=D**	**no operation**	**Dsz**
1	**D=word(F)**	**store word from the F bus**	**1x32**
2	**D=word(M)**	**store word from the M bus**	**1x32**
3	**D=word(V)**	**store word from the V bus**	**1x32**
4	***D=long(B)***	***future store long from B bus***	
5	**D=word(B)**	**store word from the B bus**	**1x32**
6	**D=short(B)**	**store half word from the B bus**	**1x32**
7	**D=byte(B)**	**store byte from the B bus**	**1x32**

The data to be stored in memory can be taken from either the B, the F, the V or the M bus. The B bus is meant for typical Register to Memory transfers and used by the C compiler which makes use of the type conversions. The F, M and V busses are used more for graphics and Image processing.

9.3.2 The data load functions

The LOAD function takes data from the Data Input register and loads it into the D register. 'Load' includes zero or sign extension of bytes and shorts which

53:51	Data load function		size
0	D=D	no operation	Dsz
1	D=word(ul)	load unsigned word from input	1x32
2	D=short(ul)	load unsigned short from input	1x32
3	D=byte(ul)	load unsigned byte from input	1x32
4	*D=long(sl)*	*future load long from input*	
5	D=word(sl)	load signed word from input	1x32
6	D=short(sl)	load signed short from input	1x32
7	D=byte(sl)	load signed byte from input	1x32

The LOAD function is executed when the Data input register contains read data from a read access and no write operation is performed in the same cycle. some examples:

DA = rdAd(A) --> D = short (ul);
DA = rdAd(F) --> D = byte (sl);
DA = Again --> D = word (sl);

The input data from the cache (directly or after a cache line read from external memory in case of a miss) is processed by the data transfer unit depending on the type of load instruction and the address used to access the data:

Byte alignment: The bytes and half words are aligned to the least significant byte positions: Byte -> bit0..7, Half word -> bit0..15.

Sign Extension: The most significant bits above the loaded data are cleared or set depending on the datatype (signed/unsigned) and the sign bit of the loaded data.

The data is loaded into the D bus register from where it is available to other units in the Imagine.

9.3.3 The internal zero and sign extend functions

The Internal sign extension function is executed if neither a Write access nor a Read access is executed:

- No Write: Current DA instruction is either Nop, AD = rdAd(X) or AD = extended (read).
- No Read: No Read Data is waiting in the

53:51	zero and sign extend functions		size
0	D=D	no operation	Dsz
1	D=zextword(B)	(zero extend) word from B bus	1x32
2	D=zextshort(B)	zero extend short from B bus	1x32
3	D=zextbyte(B)	zero extend byte from B bus	1x32
4	*D=long(B)*	*future copy long from B bus*	
5	D=sextword(B)	(sign extend) word from B bus	1x32
6	D=sextshort(B)	sign extend short from B bus	1x32
7	D=sextbyte(B)	sign extend byte from B bus	1x32

The Load instruction performs all the operations needed on the standard data types stored in data memory: byte, half word and word both signed and unsigned. When 8 or 16 bit data is read into the 32 bit processor, then the higher 24 or 16 bits are sign or zero extended by the Data I/O unit. The sign/zero extension mechanism is also available for internal operations. This function can be used for register variables which are defined as byte's and short's as a preparation for certain instructions. some examples:

D = zextbyte(B);
D = sextshort(B);
D = sextshort(B);

9.4 Data I/O control registers

9.4.1 The D bus register

The D bus register contains results of read and internal sign/zero extend operations for bytes and shorts. It is accessible as a control register to simplify state save and restore operations.

Cr36,:DIO_Dbus, 32 bit D BUS register

<table>
<tr><td colspan="32">32 bit D BUS register
[31:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

9.4.2 The DIO_Control register

The extended functions of the DIO are controlled by this register
It support four sets of extended access operations:

- 2 Dimensional and 3 Dimensional accesses
- Linear accesses with post increment, pre-increment and post decrement, pre-decrement options
- Texture read accesses via the 3D graphics pipeline
- Scratch pad operations where part of the data cache is used as scratch pad.

See the paragraphs further on for a detailed explanation.

Cr37: DIO_Control: The Datatransfer Control register

<table>
<tr><td>‘0’</td><td>Use-PAG</td><td>WR</td><td colspan="4">‘0000’</td><td>SP</td><td colspan="2">‘00’</td><td>PO</td><td>IR</td><td>DR</td><td>‘0’</td><td colspan="2">size
[1:0]</td><td colspan="4">Z Coor Size
[3:0]</td><td colspan="4">Y Coor Size
[3:0]</td><td colspan="4">X Coor Size
[3:0]</td><td colspan="4">‘0000’</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

9.4.3 The DIO_Address register

The address register contains the latest address used and is applied in incremental address modes. The address can be pre-incremented , post-incremented pre-decremented or post-incremented or left unmodified for fifo and I/O accesses.

Cr38: DIO_Address, 32 bit Data address register

<table>
<tr><td colspan="32">32 bit bit Data address register
[31:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

9.4.4 The DIO_offset register

The DIO recognises 2D and 3D addresses by the data size used for the address. 2D addresses are recognised by their 2x16 bit size and 3D addresses by 4x8 bit size. These values represent XY and XYZ co-ordinates in a rectangle or volume which origin is defined by a linear offset given in the DIO_Offset register. The address in the offset is used by the memory management hardware to select how memory is structured (Linear, 2D or 3D) and what the Image stride is (number of bytes from one row to the next). Bits [27:24] select between 16 different areas of 16Mb in the 256Mb virtual memory space. The organisation of the virtual memory is governed by standard memory allocation function. (Malloc, Free for linear memory, CreateSurface for 2D et cetera) the programmer can rely on the memory pointers and additional parameters returned by these functions).

Cr39: DIO_Offset, 32 bit Address offset register

<table>
<tr><td colspan="32">32 bit Address offset register
[31:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

9.5 Data access unit: detailed operation description

9.5.1 Selected Address

The address can come from either one of three busses can be selected to provide the Data Address: bus A, bus F or bus M. The data size of the selected bus (32,2x16,4x8) determines how the 32 available data bits are translated into an address. A 32 bit word will be interpreted as a linear address. The 32 data bits are directly used as the address. The 3D graphics pipeline can supply perspective correct addresses for 1D, 2D and 3D textures in combination with the **DA = extended** function.

9.5.2 Higher dimensional addressing via the cache

Double 16 bit addresses words and quad 8 bit addresses words are handled differently as 1x32 bit addresses. They are used for two and three dimensional addressing. Firstly an area is allocated in the Memory which will be used to store 2D and 3D data. In 2 Dimensional addressing the two 16 bit words are used as an X and Y address pair. While the 3D addressing uses the lower 3 bytes as the X, Y and Z co-ordinates. These modes use the **DIO_Offset** control register as the pointer to the origin (0,0 or 0,0,0) of the 2D or 3D structure. The **DIO_Control** register provides masks for the X, Y and Z co-ordinates which limit the number of bits which can be used for these co-ordinates (1..15) The **size[1:0]** field is used (after the mask function) to translate the X coordinate into a byte address. The size can by byte (00), 16 bit short (01) or 32 bit word (10).

A bus, F bus or M bus with data size 2x16: 2D address

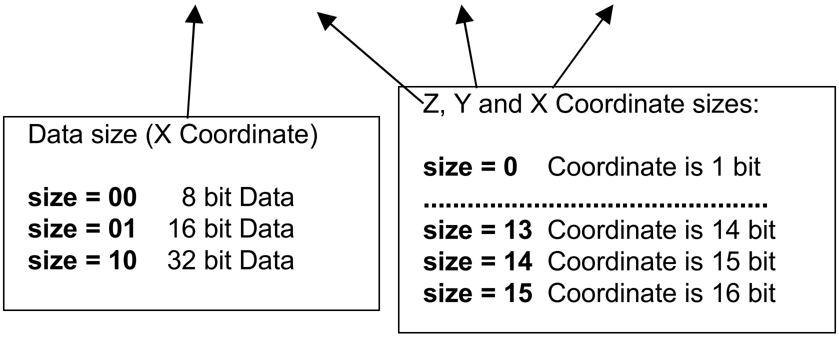
16 bit **Y** coordinate [31:16]																16 bit **X** coordinate [15:0]															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

A bus, F bus or M bus with data size 4x8: 3D address

Not used [31:24]								8 bit **Z** coordinate [23:16]								8 bit **Y** coordinate [15:8]								8 bit **X** coordinate [15:0]							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Cr37: DIO_Control: The Datatransfer Control register

‘0’	Use-PAG	WR	‘0000’				SP	‘00’		PO	IR	DR	‘0’	size [1:0]		Z Coor Size [3:0]				Y Coor Size [3:0]				X Coor Size [3:0]				‘0000’			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0



Data size (X Coordinate)

size = 00 8 bit Data
size = 01 16 bit Data
size = 10 32 bit Data

Z, Y and X Coordinate sizes:

size = 0 Coordinate is 1 bit
...
size = 13 Coordinate is 14 bit
size = 14 Coordinate is 15 bit
size = 15 Coordinate is 16 bit

Cr39: DIO_Offset, 32 bit Address offset register

32 bit Address offset register [31:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

9.5.3 The use of the 3D graphics pipeline with the extended function

The 3D graphics pipeline can read perspective MIP mapped texture data from external memory via it's own interface to the Memory Bus Controller. The Data Cache is bypassed and up to four pixels can be loaded per cycle as is needed for Bi linear interpolation. The 3D graphics pipeline supports many extra functions. It can translate almost any pixel format into the 32 bit aRGB values which are used internally for calculation. Fog and lighting coefficients calculated in the 3D graphics pipeline can be applied to the read texture data.

The memory accesses of the 3D graphics pipeline are controlled by Data I/O instructions. The addresses are not supplied directly by the programmer but are generated in the 3D perspective address generator of the 3D graphics pipeline. The function **DA = extended** starts the read access whenever the **UsePAG** flag (Use **P**erspective **A**ddress **G**enerator: bit 30) is set and the **WR** flag (**Wr**ite: bit 29) is zero.

Texture read accesses take longer than accesses to the data cache which take one cycle to generate the address, one cycle to read from the cache ram and one to translate the data. The external SDRAM or SGRAM memory itself will need at least 6 cycles when it is operated at half the clock speed in interleaved mode. The Memory Bus Controller needs a number of cycles and we want to fill the address fifo in the controller with at least a number of request to avoid bubbles (empty slots) in the pipeline which degrade performance. The optimal delay is in the order of 10 to 12 cycles. Reads are completely pipelined so one address can be send each cycle in vector mode.

```
DA  = rdAd(A)      - - >                     D = word (ul)    - >      Wr( ri++, D);
3DA = extended(A)  - - - - - - - - - - - - >  3D = word (ul)   - - - - >  Wr( ri++, D);
```

The function D = word(uI) is used to load the data into the D bus register it takes 4 cycles to translate one of 15 different Texture pixels into 32 bit aRGB, apply bilinear interpolation, lighting and fog calculations. The use of the keywords 3DA and 3D instead of DA and D is only to inform the assembler about the different behaviour of these instructions. Both options generate the same code.

Cr37: **DIO_Control:** The Datatransfer Control register

‘0’	Use-PAG	WR	‘0000’				SP	‘00’		PO	IR	DR	‘0’	size [1:0]		Z Coor Size [3:0]				Y Coor Size [3:0]				X Coor Size [3:0]				‘0000’			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

WR = 0: Read Operation

UsePAG = 1: Use Perspective Address Generator

9.5.4 Vector accesses with the extended function

The **DA = extended** function can be used to load and store Vectors (Streams of data) without the need for explicit supply of new addresses each and every access. These Vectors are elementary in the programming philosophy of the Imagine. Another use of these functions are the common Stack Push and Pop operations.

The 24 lowest word address bits, (bit 2 through bit 25, are (pre- or post) incremented / decremented each time when the function **DA = extended** is executed with a linear address (data size = 1x32) and one of the **IR**/**DR** bits in the **DIO_Control** control register, cr38, is set to a logical '1'. The **PO** bit selects between post- (**PO=1**) and pre- (**PO=0**) functionality. The **WR** flag determines the direction of the access (Read or Write)

Cr37: **DIO_Control:** The Datatransfer Control register

'0'	Use-PAG	WR	'0000'	Scr_ Pad	'00'	PO	IR	DR	'0'	size [1:0]	Z Coor Size [3:0]	Y Coor Size [3:0]	X Coor Size [3:0]	'0000'
31	30	29	28 27 26 25	24	23 22	21	20	19	18	17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0

DR = 1 Linear Address Decrement

IR = 1 Linear Address Increment

WR = 0 Read Operation
WR = 1 Write Operation

PO = 0 Pre-Increment / Decrement
PO = 1 Post-Increment / Decrement

Cr38: **DIO_Address,** 32 bit Data address register

address register [31:26] fixed	24 bit Linear address incrementer / decrementer [25:2]	fixed [1:0]
31 30 29 28 27 26	25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2	1 0

9.5.5 Scratch pad accesses

The **SP** (Scratch Pad) flag (**DIO_Control**, bit 24) disables cache operation if set and uses the on chip cache ram simply as on chip memory. Write operations to memory are not written trough to external memory and subsequent read operations can take place directly from these cache lines which are marked witch a 'scratch' flag. A '*cache line scratch flag clear*' signal is send to the cache when bit 24 in **DIO_Control** is reset.

Cr37: **DIO_Control:** The Datatransfer Control register

'0'	Use-PAG	WR	'0000'	SP	'00'	PO	IR	DR	'0'	size [1:0]	Z Coor Size [3:0]	Y Coor Size [3:0]	X Coor Size [3:0]	'0000'
31	30	29	28 27 26 25	24	23 22	21	20	19	18	17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0

SP = 0 Normal Cache operation
SP = 1 Scratch Pad operation

This function is useful in for instance vector operations which already read and / or write from memory via the vector I/O unit and which need temporary scratch memory via the Data I/O unit. Scratch Pad accesses do not degrade the bandwidth available to external memory.

Chapter

10. VECTOR I/O UNIT

The Vector I/O unit handles vector type accesses to and from external memory. It can handle incoming and outgoing data at the same time. It can convert 8 bit pseudo color and various 16 bit hi-color pixel data to an internal 32 bit αRGB value. The data is made visible on the internal V bus. It can select data from any of the eight internal data buses for vector output and convert the internal 32 bit αRGB to external 8 bit pseudo color or 16 bit hi-color using advanced dither and error propagation techniques. An transparent color range can be defined which sets the alpha value to 0 for input conversions while the (OpenGL) alpha test can be applied which is linked to the write enable signal of the pixels. The result of the Alpha test can also be applied by the Mask Generator. The output alpha value can be applied for transparency dithering in the 8 bit pseudo color mode. The unit contains a byte selector to swap byte channels on the fly during input and output operations.

fig. image i/o unit

10.1 Image I/O function select

The operation of the Vector I/O bus is controlled with the Vector I/O instruction field, together with the Vector I/O control registers. This unit Itself does not start read and write operations to image memory. These are initiated with a multiple cycle Vector Access instruction before I/O instructions are executed. Up to 128 input and /or output instructions can be executed in vector mode after a single **Vector** Access function. A single Input or Output function is executed in combination with a **Scalar** access function. See chapter 12 for more details.

INSTRUCTION CODE FIELD

58-57	Mnemonics	Vsize
00	{no op}	hold
01	**V=feedback**	Vsize
10	**V=input**	Vsize
11	**V=output**	Vsize

10.2 Output operation

Any of the eight internal busses can be selected for the output operation. Several functions can be applied on the output data.
- Output bus selection
- Byte selection
- True color to 16 bit color conversion with error diffusion
- True color to 8 bit conversion with advanced dithering
- True color to 8 bit alpha transparency dithering
- Alpha Tests

OUTPUT BUS SELECTION

VIO_Control1 [15:13]

OSOURCE		selected bus
000:	**A_BUS**	select A bus
001:	**B_BUS**	select B bus
010:	**Q_BUS**	select Q bus
011:	**F_BUS**	select F bus
100:	**M_BUS**	select M bus
101:	**U_BUS**	select U bus
110:	**D_BUS**	select D bus
111:	**V_BUS**	select V bus

10.2.1 Output source selection

OSOURCE[2:0] Image data output source. Data from any internal bus can be used for vector write operations to external memory.

10.2.2

Byte selection

Output byte selection is used if the **SO** flag (Select Output Bytes) of the **VIO_Control1** control register is set (cr45, bit 8) This feature can be used to reorder color formats, duplicate bytes, extend 8 bit to 16 bit pixels et cetera. There are four byte selectors **BY0, BY1, BY2** and **BY3**. These four 2 bit words select between the four bytes in the 32 bit selected output word. **BY0** determines the output of bits 0..7 while **BY3** selects the output for bits 24..31. The value '0' selects the lowest byte while the value '3' selects the highest byte.

SELECT OUTPUT BYTES

SO VIO_Control1 [8]

0:	**BYTE_SELECT_DIS**	no byte select
1:	**BYTE_SELECT_EN**	select bytes

example: color reformatting:
(αRGB -> RGBα or αBGR)

example: duplication:
(αRGB -> αααα)

example 8 bit pixels to 16 bit
(xxAB → AABB)
(xAxB → AABB)

BYTE SELECTION

VIO_Control1 [7:0]
operation for each byte

00:	**0**	select byte 0	(bits 7:0)
01:	**1**	select byte 1	(bits 15:8)
10:	**2**	select byte 2	(bits(23:16)
11:	**3**	select byte 3	(bits(31:24)

10.2.3 True color to 16 bit error diffusion:

Error diffusion can be selected individually on each of the four bytes. It is enabled with the **DE** flag (Diffuse Enable flag) of the **VIO_Control1** control register (cr45, 24). The Error diffusion works on bytes. One of four different diffusion sizes can be individually set for all four bytes. E.g.: 5 bit error diffusion means that the lowest 3 bit of the previous value are added to the new 8 bit value. The **DB** flag (Diffusion Begin flag) should be set to '0' if you start a diffusion operation. It disables diffusion for the first output after which it is set automatically to '1'

HICOLOR ERROR DIFFUSION ENABLE

DBDE: VIO_Control1 [25:24]

00:	**DITHER_DIS**	no error diffusion
01:	**DITHER_EN**	start diiffusion
11:	**DITHER_EN_BUSY**	continue diffusion

ERROR DIFFUSION

VIO_Control1 [23:16]
operation (per byte)

00:	**DIT8**	no error diffusion
01:	**DIT6**	6 bit error diffusion
10:	**DIT5**	5 bit error diffusion
11:	**DIT4**	4 bit error diffusion

10.2.4 True color to 16 bit color conversion:

After the Bus selection function, Byte selection function and the Error diffusion comes the Color format conversion. Among the formats supported are: **Targa** 16 bit HiColor, **XGA** 16 bit HiColor and a symmetric format with four components of 4 bit for αRGB. There is an extra 0555 format where bit 15 does not have any relation with alpha (which becomes always 0xff when read) and a 565 format where red and blue are swapped. Both these formats are Microsoft Direct3D HEL supported texture types.

COLOR_TARGA

A[7]	R[7:2]	G[7:2]	B[7:2]

COLOR_XGA

R[7:2]	G[7:1]	B[7:2]

COLOR_RGBA4

A[7:3]	R[7:3]	G[7:3]	B[7:3]

COLOR_0555

'0'	R[7:2]	G[7:2]	B[7:2]

COLOR_r565

B[7:2]	G[7:1]	R[7:2]

TRUE COLOR TO 16 BIT COLOR

OCOL: VIO_Control1 [28:26]

000:	**COLOR_PASS**	no hicolor conv.
001:	**COLOR_TARGA**	8:8:8:8 → 1:5:5:5
010:	**COLOR_XGA**	8:8:8:8 → 0:5:6:5
011:	**COLOR_RGBA4**	8:8:8:8 → 4:4:4:4
101:	**COLOR_0555**	8:8:8:8 → 0:5:5:5
110:	**COLOR_r565**	8:8:8:8 → 0:5:6:5 (red ←→ blue)

10.2.5 True color to 8 bit pseudo color

The Imagine 2 contains sophisticated logic to translate 32 bit True color to 8 bit pseudo color in a way which is relatively independent of the color look up table. A 384 entry reversed table is used with extra information for error correction. The conversion is enabled by setting the **PO** flag (Pseudo Output Conversion) in the **VIO_Control2** control register (cr46, bit 30)

PSEUDO COLOR CONVERSION ENABLE

PO: VIO_Control2 [30]

0:	**PSEUDO_DIS**	no conversion
1:	**PSEUDO_EN**	conversion enabled

10.2.6 True color to 8 bit dithering

The **CD** flag (Color Dithering) in the **VIO_Control2** control register (cr46, bit 28) must be set to enable the True color to pseudo color dithering process

PSEUDO COLOR DITHER ENABLE

CD: VIO_Control2 [28]

0:	**DITHER_DIS**	no color dithering
1:	**DITHER_EN**	color dithering enabled

10.2.7 True color to 8 bit dither matrix

There are two options for the dither matrix which can be used for true color to pseudo color generation. One is the well known ordered dither matrix. The size of the dither matrix used is 16 by 16 which is significantly larger then the more common 4x4 format. The ordered matrix has the disadvantage of showing visible '+' and 'x' structures in the rendered image. This is much less the case with the improved version which has a more random appearance The **MT** flag in **VIO_Control2** (bit 25) selects between both options

PSEUDO COLOR DITHER MATRIX

MT: VIO_Control2 [25]

0:	**ORDERED_DIT**	standard matrix
1:	**IMPROVED_DIT**	improved matrix

10.2.8 True color to 8 bit error correction

The pseudo color table stores pseudo colors based on a 3+3+3 bit rgb entry address. The errors of the pseudo color compared to its red, green and blue entry is also stored in the table. This error value can be used to correct output colors. The EC flag in **VIO_Control2** (bit 27) controls this feature .

PSEUDO COLOR ERROR CORRECTION

EC: VIO_Control2 [27]

0:	**ERR_CORR_DIS**	no error correction
1:	**ERR_CORR_EN**	correction enabled

10.2.9 Alpha Compare Test

The result of this test can be used to disable the writing of the pixel into external memory or to replaces the output color with the transparency color. The result (pass =1) also goes to the mask generator where it can be added to the range mask to disable e.g. the writing of the corresponding Depth value. The alpha value of the output color is compared with the reference alpha in control register cr47: **VIO_Alpha** bits [31:24]. The result can disable the writing of the pixels if it fails the test. It can be replaced with the transparent color value: cr49 **VIO_Transparent** if it fails the test The transparent color replaces the result from any color conversion, dithering error-diffusion et cetera

ALPHA COMPARE TEST

A_CMP: VIO_Control2 [30:28]

000:	**GL_NEVER**	passes never
001:	**GL_LESS**	alpha < ref.alpha
010:	**GL_EQUAL**	alpha == ref.alpha
011:	**GL_LEQUAL**	alpha <= ref.alpha
100:	**GL_GREATER**	alpha > ref.alpha
101:	**GL_NOTEQUAL**	alpha != ref.alpha
110:	**GL_GEQUAL**	alpha >= ref.alpha
111:	**GL_ALWAYS**	passes always

10.2.10 Alpha Dithering

The result of this test can be used to disable the writing of the pixel into external memory or to replace the output color with the transparency color. The alpha value of the output color is compared with an 8 bit dither value. Writing is disabled or the transparency color is selected if the alpha is smaller compared to the test value or if the alpha value is zero.

10.2.11 Write Disable

Writes to external memory can be disabled via various test These test are selected with the **WRdis** field (Write Disable) in control register cr47: **VIO_Alpha** bits [19:16]. Bit 16 set to '1' determines that a failed alpha test disables writing, Bit 17 set to '1' disables writing if alpha is zero and bit 19 set to 1' disables writing if the alpha is smaller then the dither value or zero.

WRITE DISABLE TESTS

WRDIS[3:0] VIO_Alpha[19:16]

bit 0:	use Alpha test
bit 1:	use Alpha zero
bit 3:	use Alpha dither

10.2.10 Transparency color

The transparent color can replace the actual internal color during writes to external memory via various test These test are selected with the **TCout** field (Write Disable) in control register cr47: **VIO_Alpha** bits [19:16]. Bit 12 set to 1 selects the transparent color when the alpha test fails Bit 13 set to '1' selects the transparent color when alpha is zero and bit 15 set to 1' selects the transparent color when the alpha is smaller then the dither value or zero.

TRANSPARENT OUTPUT COLOR

TCOUT[3:0] VIO_Alpha[15:12]

bit 0:	use Alpha test
bit 1:	use Alpha zero
bit 3:	use Alpha dither

10.3 Input instruction.

The input instruction loads the external data available on the Image bus into the V bus register/driver. Several functions can be applied on the incoming data before it is stored in the V-bus register:

- 16 bit color to 32 bit true color format conversion
- 8 bit pseudo color to 32 bit true color conversion
- Alpha generation with color key range
- Byte selection
- Data size definition

TRUE COLOR TO 16 BIT COLOR

ICOL: VIO_Control1 [31:29]

000:	**COLOR_PASS**	no hicolor conv.
001:	**COLOR_TARGA**	8:8:8:8 ← 1:5:5:5
010:	**COLOR_XGA**	8:8:8:8 ← 0:5:6:5
011:	**COLOR_RGBA4**	8:8:8:8 ← 4:4:4:4
101:	**COLOR_0555**	8:8:8:8 ← 0:5:5:5
110:	**COLOR_r565**	8:8:8:8 ← 0:5:6:5 (red ←→ blue)

10.3.1 16 bit input color conversion:

Incoming 16 bit colors can be expanded to 32 bit true color αRGB. The color format is defined by bits [31:29] of the **VIO_Control1** control register (see table above). The lower bits which are added during the expansion are copied from the most significant bits of the 1, 4, 5 or 6 bit color components. This ensures that the 8 bit results components are spread evenly over the entire range of 00 to FF.

PSEUDO COLOR CONVERSION ENABLE

PI: VIO_Control2 [31]

0:	**PSEUDO_DIS**	no conversion
1:	**PSEUDO_EN**	conversion enabled

SELECT INPUT BYTES

SI VIO_Control1 [9]

0:	**BYTE_SELECT_DIS**	no byte select
1:	**BYTE_SELECT_EN**	select bytes

10.3.2 8 bit input color conversion

8 bit pseudo colors can be translated to 32 bit true color via the 256 entry Color Look Up table. This option is enabled by setting the **PI** flag: **VIO_Control2[31]**.

10.3.3 Alpha generation by color key range

A 32 bit, 16 bit or 8 bit color which is translated to 32 bit αRGB can be tested on a αRGB transparency range. This feature is particular useful for 16 bits colors who do not have any alpha information. 8 bit pixels are converted via the color look up table which itself already allows an alpha to be assigned to each individual pseudo color. The lowest and highest transparent value for red, green and blue (and alpha are contained within control registers cr50 and cr51: **VIO_ColorKeyLo** and **VIO_ColorKeyHi**. The individual color components can be enabled by the four **ColorKeyEn** flags in the **VIO_Alpha[11:8]** control register. The Alpha value is passed unchanged is none of the four flags is set. If 1 or more is set then the alpha is set to zero if the enabled color components are within their transparency range otherwise the alpha is set to 0xff (opaque). The **Edge Alpha** value in **VIO_Alpha[7:0]** can replace the alpha values of non-transparent pixels which come directly before or after a transparent pixel. **Edge Alpha** must be 0xff to disable this function.

BYTE SELECTION

VIO_Control1 [7:0]
operation for each byte

00:	**0**	select byte 0 (bits 7:0)
01:	**1**	select byte 1 (bits 15:8)
10:	**2**	select byte 2 (bits(23:16)
11:	**3**	select byte 3 (bits(31:24)

10.3.4 Byte selection:

The byte selection field from the image I/O control registers can be applied by the input instruction to perform a byte swap operation on the bytes coming from the Color Format Conversion unit. Each byte of the V-bus register can be loaded with each of the four input bytes.

10.3.5 Data Size definition:

The two bit word size information which will be attached to the input data and stored in the V bus register is determined by **Size**:

DATA SIZE DEFINITION

VSIZE: VIO_Control1 [11:10]

00:	**QUAD_BYTE**	4x8 bits data
01:	**DOUBLE_SHORT**	2x16 bits data
11:	**SINGLE_WORD**	1x32 bits data

10.4 Feedback instruction

The feedback instruction does not perform any communication with the outside world. It uses the capability of the image I/O port to select any of the eight internal databuses and to perform a Byte Selection in the same way as the output instruction. The result is visible after one cycle in the V bus register.

10.5 Simultaneous input and output

The Imagine2 can perform simultaneous in and output. The program below reads 32 bit data from memory, does a table look up operation in the register file and writes the result back to external memory. The read data comes from the input fifo while the write data is stored in the output fifo and then passed to external memory. The external memory bus speed is twice as high as the VIO speed so the complete operation can be finished within a single vector.

```
repeat,  graph ( table_look_up );
;;
table_look_up:
V = input => genad(A) => A = rd4x8(ri) => V = output;
```

10.6 Setting up the translation tables

The VIO contains two tables for the conversion between pseudo color to 32 bit true color and visa versa. Both tables can be accessed via control register: **VIO_Control2** the table addresses are given by bits [8:0] and the selection between the two tables is made by **TS** (bit 9) '0' selects the 256 entry pseudo color to true color conversion table while '1' selects the 384 entry true color to pseudo color table. These tables can be read if the **RD** flag (bit 10) is set. This flag needs to be '0' if the tables are used in normal operation. The read and write operations use control register cr48: **VIO_PseudoData** as the entry point. Both read and write operations are auto incremental. Write but also Reads can be done in vector mode. A modification of the table index or the TS flag when the RD flag is set or setting the RD flag itself will initiate a two cycle pre-load mechanism to read entries two entries in advance.

10.6.1 The contents of the pseudo color to true color table.

This table contains 256 entries with alpha, red, green and blue values for each of the 256 possible palette entries.

Bits [31:24]	=	Alpha [7:0]
Bits [23:16]	=	Red [7:0]
Bits [15:8]	=	Green [7:0]
Bits [7:0]	=	Blue [7:0]

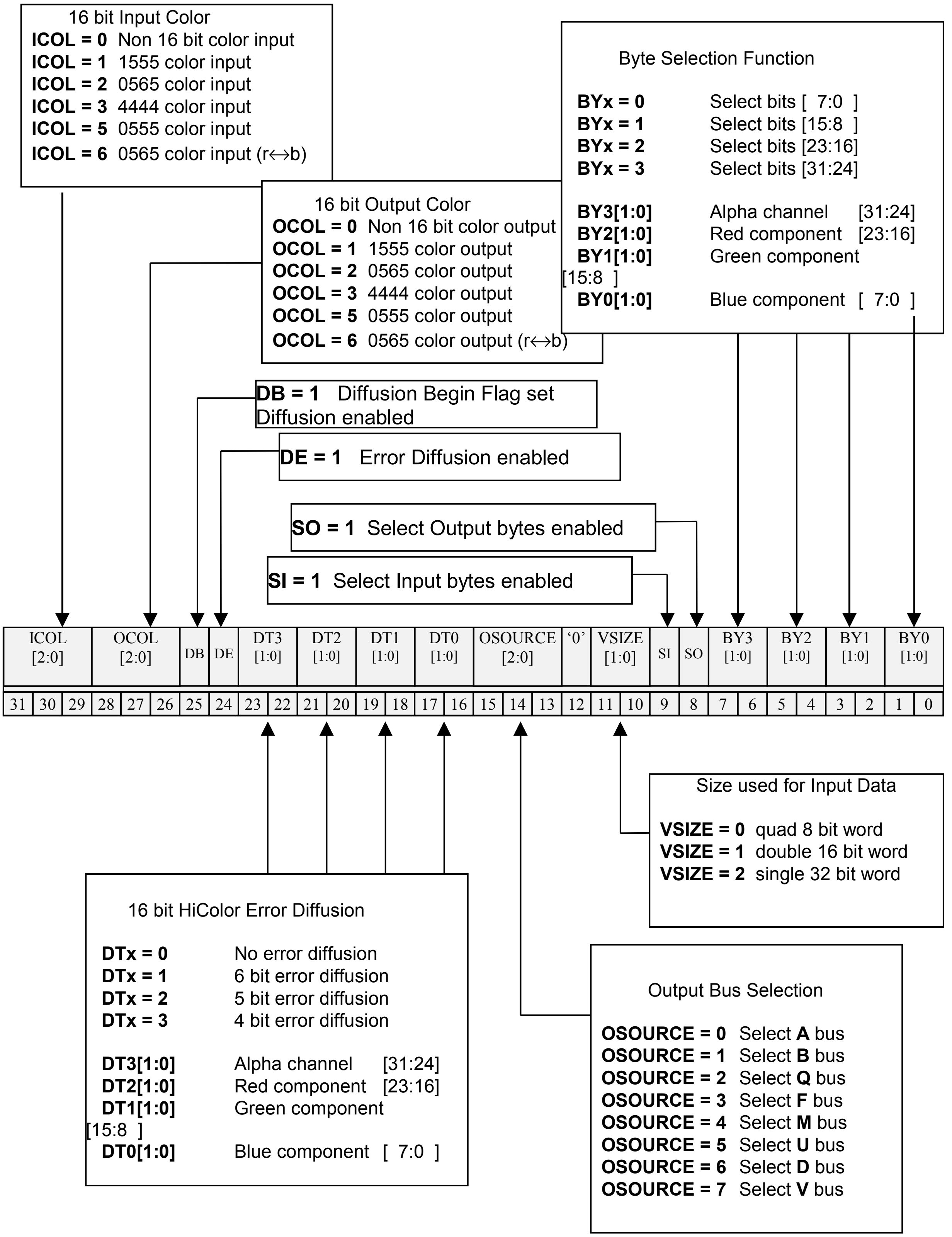
10.6.2 The contents of the true color to pseudo color table.

The table which contains information to translate true color information to 8 bit pseudo colors. It has 384 entries based on a 3 bit Red, 3 bit Green and 3 bit Blue index. The table index used during translation is given by Red + 8 x Green + 64 x Blue. This limits the number of blue entries to 6. The number of entries per color component is given by **REDNO[2:0]**, **GREENNO[2:0]** and **BLUENO[2:0]** in control register cr46: **VIO_Control2**. Each entry contains the best fitting pseudo color. The error value for red, green and blue compared to the ideal values based on the entry address: **RED_error[5:0]**, **GREEN_error[5:0]** and **BLUE_error{5:0]** plus three left/right neighbour field for red, green and blue: **RED_next[1:0]**, GREEN_next[1:0] and **BLUE_next[1:0]**. The format of the errors is signed 2.4 where the binary point separates entry numbers and sub entry values. Values higher then binary 1.1111 and lower then binary (minus) -10.0000 must be clamped to these values. The neighbour bits detect if the neighbour above (in the red or green or blue direction) or below has a red, green or blue component which is identical (or nearly identical) to the one of this entry. Bit 1 corresponds to the entry above while bit 0 corresponds with the entry below.

10.7 The control registers of the VIO

10.6.1 The Vector I/O Control register no. 1

Cr45: **VIO_Control1:** The Vector I/O Control register no. 1



16 bit Input Color
ICOL = 0 Non 16 bit color input
ICOL = 1 1555 color input
ICOL = 2 0565 color input
ICOL = 3 4444 color input
ICOL = 5 0555 color input
ICOL = 6 0565 color input (r↔b)

16 bit Output Color
OCOL = 0 Non 16 bit color output
OCOL = 1 1555 color output
OCOL = 2 0565 color output
OCOL = 3 4444 color output
OCOL = 5 0555 color output
OCOL = 6 0565 color output (r↔b)

Byte Selection Function

BYx = 0	Select bits [7:0]
BYx = 1	Select bits [15:8]
BYx = 2	Select bits [23:16]
BYx = 3	Select bits [31:24]
BY3[1:0]	Alpha channel [31:24]
BY2[1:0]	Red component [23:16]
BY1[1:0]	Green component [15:8]
BY0[1:0]	Blue component [7:0]

DB = 1 Diffusion Begin Flag set Diffusion enabled

DE = 1 Error Diffusion enabled

SO = 1 Select Output bytes enabled

SI = 1 Select Input bytes enabled

ICOL [2:0]			OCOL [2:0]			DB	DE	DT3 [1:0]		DT2 [1:0]		DT1 [1:0]		DT0 [1:0]		OSOURCE [2:0]			'0'	VSIZE [1:0]		SI	SO	BY3 [1:0]		BY2 [1:0]		BY1 [1:0]		BY0 [1:0]	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Size used for Input Data
VSIZE = 0 quad 8 bit word
VSIZE = 1 double 16 bit word
VSIZE = 2 single 32 bit word

16 bit HiColor Error Diffusion

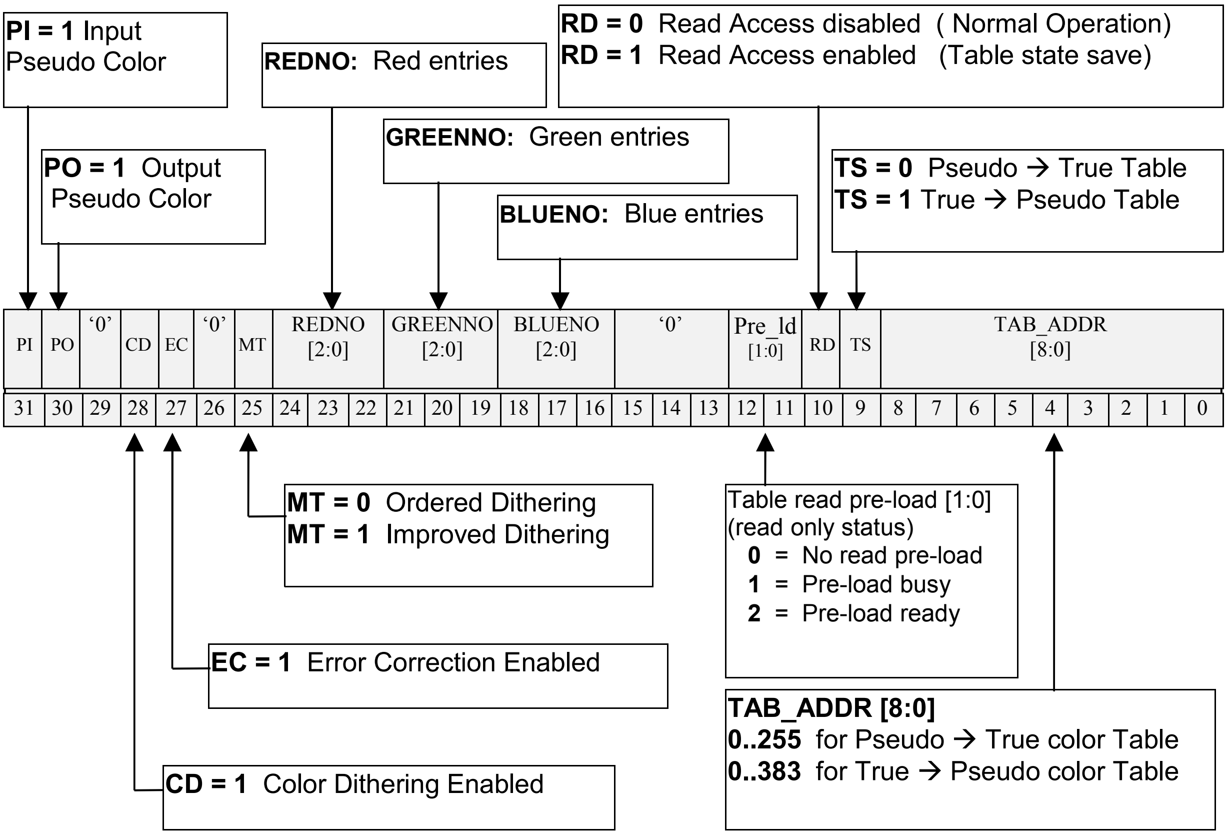
DTx = 0	No error diffusion
DTx = 1	6 bit error diffusion
DTx = 2	5 bit error diffusion
DTx = 3	4 bit error diffusion
DT3[1:0]	Alpha channel [31:24]
DT2[1:0]	Red component [23:16]
DT1[1:0]	Green component [15:8]
DT0[1:0]	Blue component [7:0]

Output Bus Selection

OSOURCE = 0	Select **A** bus
OSOURCE = 1	Select **B** bus
OSOURCE = 2	Select **Q** bus
OSOURCE = 3	Select **F** bus
OSOURCE = 4	Select **M** bus
OSOURCE = 5	Select **U** bus
OSOURCE = 6	Select **D** bus
OSOURCE = 7	Select **V** bus

10.6.2 The Vector I/O Control register no. 2

Cr46: **VIO_Control2:** The Vector I/O Control register no. 2



PI = 1 Input Pseudo Color

PO = 1 Output Pseudo Color

REDNO: Red entries

GREENNO: Green entries

BLUENO: Blue entries

RD = 0 Read Access disabled (Normal Operation)
RD = 1 Read Access enabled (Table state save)

TS = 0 Pseudo → True Table
TS = 1 True → Pseudo Table

MT = 0 Ordered Dithering
MT = 1 Improved Dithering

EC = 1 Error Correction Enabled

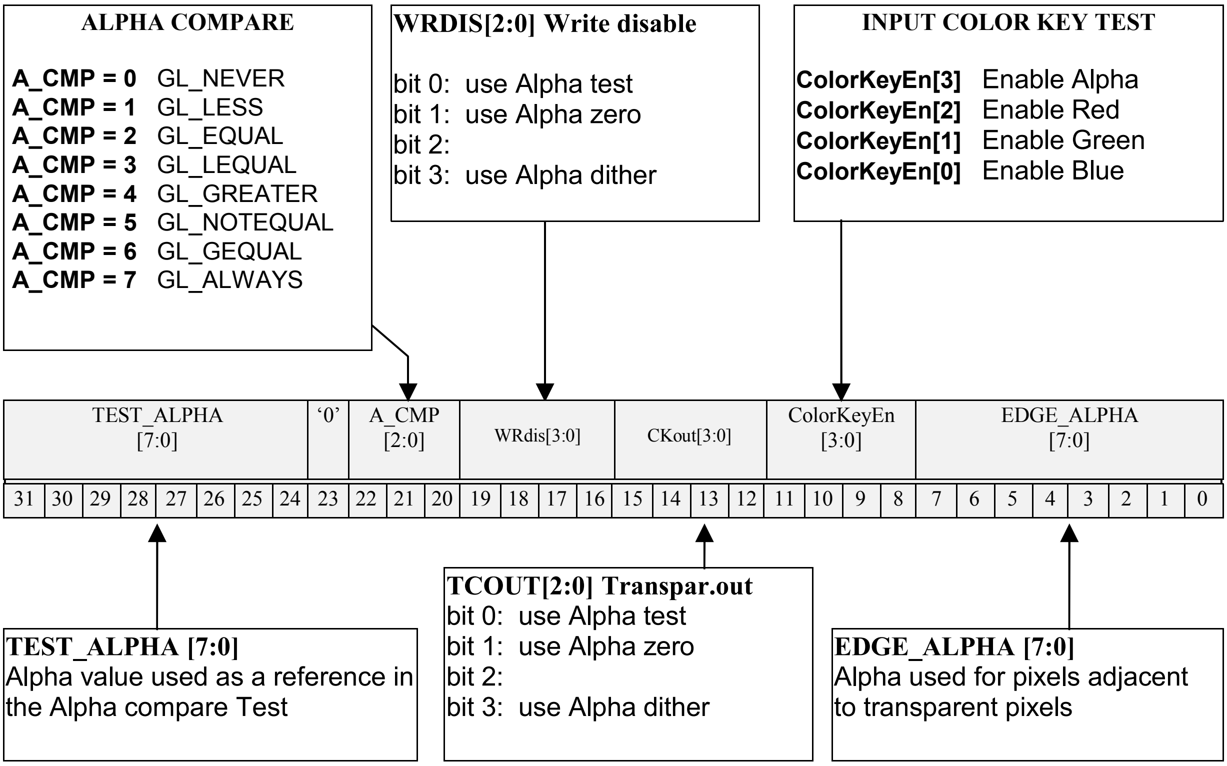
CD = 1 Color Dithering Enabled

Table read pre-load [1:0]
(read only status)
0 = No read pre-load
1 = Pre-load busy
2 = Pre-load ready

TAB_ADDR [8:0]
0..255 for Pseudo → True color Table
0..383 for True → Pseudo color Table

10.6.3 The alpha test and alpha generation control register

Cr47: **VIO_Alpha:** Alpha test and Alpha generation Control register



ALPHA COMPARE

A_CMP = 0	GL_NEVER
A_CMP = 1	GL_LESS
A_CMP = 2	GL_EQUAL
A_CMP = 3	GL_LEQUAL
A_CMP = 4	GL_GREATER
A_CMP = 5	GL_NOTEQUAL
A_CMP = 6	GL_GEQUAL
A_CMP = 7	GL_ALWAYS

WRDIS[2:0] Write disable

bit 0: use Alpha test
bit 1: use Alpha zero
bit 2:
bit 3: use Alpha dither

INPUT COLOR KEY TEST

ColorKeyEn[3]	Enable Alpha
ColorKeyEn[2]	Enable Red
ColorKeyEn[1]	Enable Green
ColorKeyEn[0]	Enable Blue

TEST_ALPHA [7:0]
Alpha value used as a reference in the Alpha compare Test

TCOUT[2:0] Transpar.out
bit 0: use Alpha test
bit 1: use Alpha zero
bit 2:
bit 3: use Alpha dither

EDGE_ALPHA [7:0]
Alpha used for pixels adjacent to transparent pixels

10.6.4 The pseudo ←→ true color conversion tables entry

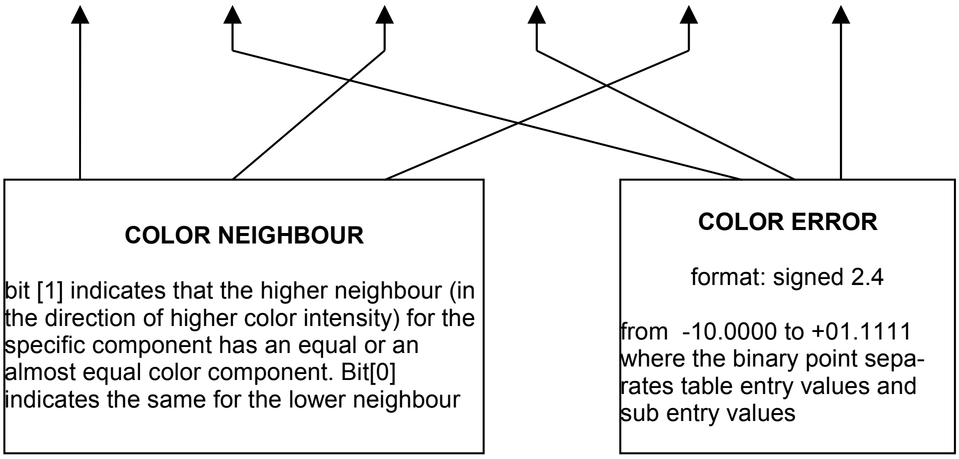
Cr48: **VIO_PseudoData:** The Vector Pseudo ←→ True Color Table Data

The Pseudo Color → True Color Table's Data format

<table>
<tr><td colspan="8">ALPHA component
[7:0]</td><td colspan="8">RED component
[7:0]</td><td colspan="8">GREEN component
[7:0]</td><td colspan="8">BLUE component
[7:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

The True Color → Pseudo Color Table's Data format

<table>
<tr><td colspan="8">PSEUDO color
[7:0]</td><td colspan="2">RED
next
[1:0]</td><td colspan="6">RED
error [5:0]</td><td colspan="2">GREEN
next
[1:0]</td><td colspan="6">GREEN
error [3:0]</td><td colspan="2">BLUE
next
[1:0]</td><td colspan="6">BLUE
error [5:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>



COLOR NEIGHBOUR

bit [1] indicates that the higher neighbour (in the direction of higher color intensity) for the specific component has an equal or an almost equal color component. Bit[0] indicates the same for the lower neighbour

COLOR ERROR

format: signed 2.4

from -10.0000 to +01.1111 where the binary point separates table entry values and sub entry values

10.6.5 The transparent output color

Cr49: **VIO_ColorKeyOut:** The Output Color Key

<table>
<tr><td colspan="8">Alpha or Pseudo output
color key [7:0]</td><td colspan="8">Red low output
color key [7:0]</td><td colspan="8">Green low output
color key [7:0]</td><td colspan="8">Blue low output
color key [7:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

10.6.6 The transparent color input range

Cr50: **VIO_ColorKeyLo** Lowest Transparent values of the Input and Output

<table>
<tr><td colspan="8">Alpha or Pseudo low level
color key [7:0]</td><td colspan="8">Red low level
color key [7:0]</td><td colspan="8">Green low level
color key [7:0]</td><td colspan="8">Blue low level
color key [7:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

Cr51 **VIO_ColorKeyHi** Highest Transparent values of the Input and

<table>
<tr><td colspan="8">Alpha or Pseudo high level
color key [7:0]</td><td colspan="8">Red high level
color key [7:0]</td><td colspan="8">Green high level
color key [7:0]</td><td colspan="8">Blue high level
color key [7:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

Chapter

11. THE PROGRAM SEQUENCER

The Program Sequencer
is responsible for the control flow of a program running on the IMAGINE. It is optimised for both high level language processing and specialised assembly code. It handles Jumps, Calls, Returns and Repeat functions. Together with the ALU, all C language expressions like A==B, A!=B, A>B, A>=B, A<B, A<=B, are handled with a single ALU function and a single control flow function.
The IMAGINE can control complex Multi Media systems. It handles Interrupts and return from interrupts for many real time functions like Video I/O, Audio I/O, and several communication channels.

11.1 The program sequencer instruction word

The Program Sequencer instruction determines the value of the 24 bit instruction address pointer. The instruction address points to 64 bit wide instructions so the 24 bits cover an address range of 128 Mbyte.
(The current instruction set allows future 32 bit addresses)

Instruction code Ic[63:50]

Sequencer: ‘1100’				Funct group		Address mode			Condition select				not con
63	62	61	60	59	58	57	56	55	54	53	52	51	50

Instruction code Ic[23:0]: jump, branch, call, subr

Absolute or Relative Address Offset [23:0]																							
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Instruction code Ic[23:0]: continue, push, pop, jump_reg, branch_reg, call_reg, subr_reg, call_int, return, access_IC

FVS	IAI	IRD	IWR	‘0000’				FLAG MODIFICATION FIELD [2:0]								FLAG VALUE FIELD [2:0]							
								MI 2	MI 1	‘0’	‘0’	‘0’	UF 2	UF 1	UF 0	MI 2	MI 1	‘0’	‘0’	‘0’	UF 2	UF 1	UF 0
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Instruction code Ic[23:0]: repeat instruction for vector processing

RCS	‘0’	Start repeat zone [5:0]						‘00’		Last in repeat zone [5:0]						Fixed repeat count option [7:0]							
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Function group field (Ic58..59)
One of four major groups of instructions can be selected with instruction bits IC[59:58] The jump, call and return are familiar instructions which will be specified in detail in this chapter. The repeat function plays an important role in vector processing type operations, including vectors with variable length.

Function group:

0	**jump** instructions
1	**call** instructions
2	**return** instructions
3	**repeat** instruction

Address Mode field (Ic55..57)
The Sequencer instructions select from seven address modes to obtain a destination address (see table)

The Instruction word can provide a 24 bit absolute address or signed offset. This is the type of instruction which is mostly used for code generation: the destination address is known during compilation time. Within a program, the jumps and calls should always use relative addresses to be re-locatable. The address register can also provide a 24 bit absolute address or signed offset for the cases where the destination address is not known during compilation. This occurs frequently in C programs with calls using pointers to functions. The function to be called is given during run-time and not known during compilation time. A calculated goto instruction is used by optimising compilers to implement fast switch statements. In this case the jump instruction should be executed with the calculated destination address stored in the address register.

Sequencer Address Modes

0	PC := PC + 1
2	PC := Interrupt table register
3	PC := Top of (Internal) Stack
4	PC := Address register
5	PC := PC + Address register
6	PC := Immediate Address
7	PC := PC + Immediate Offset

The fifth way of obtaining an destination address is using the interrupt table register. This option is provided to enable high speed real time processing in Multi Media applications.
Ranging from medium to complex systems with many different simultaneous interrupts from both video and audio I/O. A number of interrupts coming from different real time tasks can be handled in a single pass of circa 1.0 microsecond without the need of saving and restoring the processor state over and over again. The interrupt table register contains the start of the interrupt service routine table in its highest bits (23:8). The lower bits are defined by the interrupt waiting to be serviced (see description further on).

The Conditional Execution Field

Allows the program sequencer instruction to be conditional depending on status information from the ALU or the sequencer control/status register. The contents of the following registers are set conditionally: the **Program Counter**, the Internal Micro **StackPointer**, the **Flags** in the sequencer control register and the **Address Register**.

The Condition field (Ic51..54)

The actual condition to be used by the jump, call or return function is selected with this field. It selects between the individual status bits coming from the ALU or status bits from the Sequencer status register.

The 'Not' field (IC 50).

If an instruction (jump, call, return) is executed conditionally, then the value of the status bit used for the conditional instruction can be used as it is, or can be inverted with the use of the 'Not' field. If the status bit is not inverted the instruction will be executed if the instruction is true. Otherwise, when the status bit is inverted, the instruction will be executed if the condition is false.
Not = '0' Execute **if** (condition), (cond.='1')
Not = '1' Execute **if not** (condition)

Conditional control flow options:

	Pos. Condition	Neg. Condition
0	**if** (always)	**if not** (always)
	------------------------ User flags ---------------------------	
1	**if** (user_flag0)	**if not** (user_flag0)
2	**if** (user_flag1)	**if not** (user_flag1)
3	**if** (user_flag2)	**if not** (user_flag2)
	------------------------ ALU flags ---------------------------	
4	**if** (zero)	**if not** (zero)
5	**if** (negative)	**if not** (negative)
6	**if** (carry)	**if not** (carry)
7	**if** (sgncmp)	**if not** (sgncmp)
	---------------- Vector Processing flags ------------------	
9	**if** (repeat_smaller)	**if not** (repeat_smaller)
10	**if** (im_mask_empty)	**if not** (im_mask_empty)
11	**if** (im_access_busy)	**if not** (im_access_busy)
	----------------- Floating Point flag------------------------	
12	**if** (float_error)	**if not** (float_error)
	----------------------- Interrupts -----------------------------	
14	**if** (interrupt1)	**if not** (interrupt1)
15	**if** (interrupt2)	**if not** (interrupt2)

C Language compatibility

A C compiler or an assembly code programmer can implement all conditional jumps and calls like X==Y, X!=Y, X>Y, X>=Y, X<Y, X<=Y for both signed and unsigned numbers with just two instructions: a single ALU and a single Sequencer instruction. The table specifies the individual cases.

The Vector Processing Flags

These flags are typically used in assembly code programs. The Image mask can be checked to see if any of the up to 256 pixels covered by it, needs to be written to the frame buffer (im_mask_empty/im_mask_filled). The entire Vector write may be skipped if empty. The Image Bus Access test is used to wait for the end of a Vector read or write operation.

The Interrupt pending flags

These are typically used during interrupt service routines with provisions for advanced Multi Media systems.

C compare functions:

Equation:	ALU	Condition
X == Y	X-Y	**if** (zero)
X != Y	X-Y	**if not** (zero)
signed:		
X >= Y	X-Y	**if** (sgncmp)
X <= Y	X-Y-1	**if not** (sgncmp)
X > Y	X-Y-1	**if** (sgncmp)
X < Y	X-Y	**if not** (sgncmp)
unsigned:		
X >= Y	X-Y	**if** (carry)
X <= Y	X-Y-1	**if not** (carry)
X > Y	X-Y-1	**if** (carry)
X < Y	X-Y	**if not** (carry)

11.2 Sequencer control registers

cr52: **SEQ_Status:** Sequencer status / control

IP2	IP1	‘0’	FPE	IAB	IMZ	RRS	‘0’	MI2	MI1	‘000’			UF2	UF1	UF0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr53: **SEQ_PrCounter:** Program Counter

Program Counter [23:0]																							
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr54: **SEQ_Address:** The Address register

Address register [23:0]																							
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The Address micro stack, sixteen levels deep

Address stack [23:0]																							
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr55: **SEQ_Interrupt:** The Interrupt Routine Pointer

Interrupt Table Address [23:0]																LEV	Interrupt Vector [3:0]				S/H	‘00’	
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr56: **SEQ_Repeat:** The Repeat Count

Vector Length [15:0]															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr57: **SEQ_MaxRepeat:** The Max Repeat Count

Vector Stride [7:0]							
7	6	5	4	3	2	1	0

cr60 and cr61: **ICA_Low, ICA_High:** The Instruction Cache access registers

Instruction Cache data for low level cache access [31:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	

control register	assembly code name	REGISTER FUNCTION	word size	byte access b3 b2 b1 b0	default at reset
cr52	SEQ_Status	Status and Control Register	16 bit	--- --- ro ro	0x00C0
cr53	SEQ_PrCounter	Program Counter	24 bit	--- ro ro ro	0x000000
cr54	SEQ_Address	Address Register	24 bit	--- rw rw rw	0x000000
cr55	SEQ_Interrupt	Interrupt Table Register	24 bit	--- rw rw ro	
cr56	SEQ_Repeat	Total Vector Length	16 bit	--- --- rw rw	0x0000
cr57	SEQ_MaxRepeat	Vector Stride	8 bit	--- --- --- rw	0x3F
cr59	SEQ_Test	Sequencer Test Register	30 bit	ro ro ro ro	
cr60	ICA_Low	Instruction Cache Low	32 bit	rw rw rw rw	
cr61	ICA_High	Instruction Cache High	32 bit	rw rw rw rw	

11.3 The control register functions

SEQ_Status: The Status / Control Register cr52

The Status and control Register contains a number of flags which are related to the program flow of the Imagine. The Interrupt handler provides/uses 4 flags. There are 3 user flags. The repeat count, the Image Mask generator, the Floating point handler and the Vector access unit also provide flags.

SEQ_PrCounter: The Program Counter cr53

The Program counter holds the address which is used to access the Instruction memory. The Jump, Call, Return and Repeat instructions use this register to control the program flow. The register is read-only for control register accesses. The PC contents leads the actual executed instruction by two cycles. This means that any change of the PC due to a Jump, Call or Return happens to be two cycles before the instruction is loaded and being executed. These two cycles are called *branch delay cycles, and are typical for RISC processors.* The Instructions they execute are called *branch delay instructions.* It is clear that another Jump, Call or Return is not recommended in a branch delay cycle.

SEQ_Address: The Address Register cr54

The Address Register is readable and writable by external access. The contents can be used as an absolute address or a signed offset for Jumps and Calls. This feature is used to Jump and Call to destination addresses which are not known during compilation time. The Address Registers Internal Stack access function: A value in the Address Register can be Pushed to the Stack and vice versa: the Top of Stack value can be Popped to the Address Register (see PUSH and POP). The Control Flag Restore function: Bits [7:0] can be restored to the corresponding control flags of the Control/Status register. The Instruction cache access function: The Address register contains the Instruction address during low level read/write accesses to the Instruction cache.

The Micro Address Stack

The Imagine contains a small 16 entry internal stack to temporary save Program Counter addresses. It can be used by interrupt service routines, Assembly code and (not excluded) by optimising compilers. The Programmer can access the Stack with the use of the Address register. A value in the Address Register can be Pushed to the Stack while the Top of Stack value can be Popped back into the Address Register. This feature is used by high level language function calls to save the return address on an external stack and to restore it at the end of the call.

SEQ_Interrupt: The Interrupt Service Routine Pointer cr55

This register contains the start address of the Interrupt to be served. The highest 16 bits (23:8) are writable by external access. The lowest 8 bits are provided by the pending interrupt: The Interrupt level and the Interrupt Vector. The least significant Table Entry address bit indicates if the actual branch to the service routine is done by hardware or software. Software Interrupt Jumps and Calls can be applied to handle several waiting interrupts in one pass without the need to Save and Restore the Processor state for each interrupt.

SEQ_Repeat: The Repeat Register cr56

This register is used by the Repeat function and by Vector accesses to external Memory. The Value defines the total (variable) *Vector Length*. The Repeat function catches instructions in a socalled repeat catch range and repeats these functions a programmable number of times (in a range from 1 to 256 times). This enables variable length vector processing (all Interrupts are disabled during both catching and repeating). The maximum repeat value is 32768 (stored as N-1 = 32767).

SEQ_MaxRepeat: The Maximum Repeat Count Register cr57

This register is used to split long vectors into vectors with length ([cr57] + 1) plus a final (smaller) 'tail' vector. This is the socalled *Vector Stride*. Register cr57 is set to its default value of 0x3F (repeat=64) during reset.

ICA_Low and ICA_High: The Instruction Cache access registers cr60,cr61

The 64 bit value contained in both registers can be written to the Instruction cache memory. This is typically used in the boot process where a small on chip boot rom will load the boot program from an external EPROM into the instruction cache. These registers are also used for low level reads from cache ram.

11.4 The control flow instructions

11.4.1 The jump instructions

The JUMP Instruction

The 24 bit address in the instruction word is used for the new address. In case of an absolute address JUMP, the value replaces the current value of the program counter. The jump will be effectuated after the two branch delay instructions. If the jump is relative the 24 bits value from the instruction word is added to the instruction address of the last branch delay instruction: the current instruction address plus two. Within programs the relative jumps should be used while absolute jumps should be used for system functions. The mnemonics differentiate between the two: relative jumps are called **branch**es while absolute jumps are simply referred to as **jump**s.

Mnemonics:

```
jump (label);
branch (label);
if (condition),  jump (label);
if (condition),  branch (label);
if not (cond.),  jump (label);
if not (cond.),  branch (label);
```

The JUMP REGISTER Instruction

The 24 bit address register, (SEQ_Address: cr54), is used for the new address. In case of an absolute address JUMP REGISTER, the value replaces the current value of the program counter. The jump will be effectuated after the two branch delay instructions. If the jump is relative the 24 bits value from the address register is added to the instruction address of the last branch delay instruction: the current instruction address plus two. It can be combined with the **flags()** function: The 24 bit data field can be used independently to set/reset any number of flags in the status/ control register.

Mnemonics:

```
jump_reg;
branch_reg;
if (condition),  jump_reg;
if (condition),  branch_reg;
if not (cond.),  jump_reg;
if not (cond.),  branch_reg;
```

The JUMP INTERRUPT Instruction

Provided together with the Call Interrupt for quick response interrupt service processing in complex Multimedia designs. The jump can test the pending of a second interrupt while servicing one. The jump has the same effect as a serviced interrupt but has the advantage that the state of the processor does not have to be saved and restored between these interrupts.

Mnemonics:

```
jump_int;
if (condition),  jump_int;
if not (cond.),  jump_int;
```

The 24 bit interrupt table register (SEQ_Interrupt: cr55) is used for the new address. The Interrupt table address (bits [23:8]) is combined with information from a pending interrupt: interrupt level and interrupt vector. The 'least' significant bit (bit2) is set to '1' which means that the software entry for the interrupt is used. The resulting address replaces the current value of the program counter. The jump will be effectuated after the two branch delay instructions and is always absolute. It can be combined with the **flags()** function: The 24 bit data field can be used independently to set/reset any number of flags in the status/ control register.

The CONTINUE Instruction

This is the default sequencer instruction: a dummy JUMP instruction. It uses the address mode: PC+1 and it's mnemonics **continue** is optional. It should be used when you want to modify the (timings-critical) bits in the status and control register. (The equivalent dummy instructions for the CALL and RETURN functions are: PUSH STACK and POP STACK.)

Mnemonics:

```
[continue]
flags(....);
flags_restore;
if (condition), flags(...);
if not (cond.), flags(...);
```

11.4.2 The call instructions

The CALL Instruction

The 24 bit address in the instruction word is used for the new address while the contents of the Program Counter + 1 is pushed on the small internal Stack which holds eight words. This address can be used by the function call handling software for a future return. Highly efficient assembly code can use the tiny Stack for function calling without overhead in timing critical inner loops (up to six levels with two levels reserved for interrupts). In case of an absolute address Call, the value replaces the current value of the program counter. The call will be effectuated after the two branch delay instructions. If the call is relative then the 24 bits value from the instruction word is added to the instruction address of the last branch delay instruction: the current instruction address plus two. Within programs the relative jumps should be used while absolute jumps should be used for system functions. Absolute calls are simply **call**s while relative calls are referred to as **subr**s.

Mnemonics:

call (label);
subr (label);
if (condition), call (label);
if (condition), subr (label);
if not (cond.), call (label);
if not (cond.), subr (label);

The CALL REGISTER Instruction

The 24 bit address register (SEQ_Address: cr54) is used for the new address while the contents of the Program Counter + 1 is pushed on the small internal Stack which holds eight words. The CALL REGISTER Instruction is identical with the normal CALL instruction. The Call can be absolute and relative. It is used in situations in which the destination address is not known during compilation time. A common example are function calls which use the pointer to a function to execute the call. The software determines during Run time which version of a certain function will be used. A function may have several versions because it drives different devices. Another reason may be a 'global' parameter which defines various quality levels of rendering. This function can be combined with **flags()**.

Mnemonics:

call_reg;
subr_reg;
if (condition), call_reg;
if (condition), subr_reg;
if not (cond.), call_reg;
if not (cond.), subr_reg;

The CALL INTERRUPT Instruction

Is provided together with the Jump Interrupt for quick response interrupt service processing in Multimedia designs with two or more display formats. The call can test the pending of a second (Line-) interrupt while servicing one. The call has the same effect as a serviced interrupt but has the advantage that the internal state of the processor does not have to be saved for the second time.

The 24 bit interrupt register (SEQ_Interrupt: cr55) is used for the new addres Counter + 1 is pushed on the internal Stack. The Interrupt table address (2 from a pending interrupt: interrupt level and interrupt vector. The 'least' significant bit (bit2) is set to 1 which means that the software entry for the interrupt is used. The resulting address replaces the current value of the program counter. The jump will be effectuated after the two branch delay instructions. The CALL on INTERRUPT is by definition an absolute call. This function can be combined with **flags()**.

Mnemonics:

call_int;
if (condition), call_int;
if not (cond.), call_int;

The PUSH STACK Instruction

This instruction does not influence the program flow. The Address Register is pushed onto the internal Stack (16 deep) instead of the Program Counter. High level Function calls use it to return to an externally saved return address.

Mnemonics:

push, [flags()];
if (condition), push, [flags()];
if not (cond.), push, [flags()];

11.4.3 The return instructions

The RETURN Instruction

The return mechanism is used to continue operation from the point on which a function call or an interrupt became effective. The Top Of Stack is popped from the tiny internal stack and placed into the Program Counter. The Return will be effectuated after the two branch delay instructions. The program will continue at the address which is popped from the Top Of Stack to the Program counter. C code which uses pointers to functions for run-time depended function calling should use the return mechanism for this type of function calls. If the return is used to return from an Interrupt the Reset function should be applied
to reset the interrupt mask in the status register of the corresponding level.

Mnemonics:

return [, flags()];
if (condition), return [, flags()];
if not (cond.), return [, flags()];

The POP STACK Instruction .

This instruction does not influence the program flow. The Address Register instead of the Program Counter is popped from the internal Stack. High level Function call's use it to obtain the return address to save it on an external stack. This function can be combined with **flags()**.

Mnemonics:

pop [, flags()];
if (condition), pop [, flags()];
if not (cond.), pop [, flags()];

11.4.5 The repeat instruction

The REPEAT Instruction

This instruction is used by for vector (stream) processing. Processing continues normally until the repeat range is reached. Instructions within the repeat range are catched and repeated a variable number of times. The last instruction of the repeat range also holds the program counter. Two 6 bit values define the range which may start at any instruction from 3 cycles up to 67 cycles after the repeat instruction and end at any other in that same range provided that the last instruction is equal to or after the first instruction. as the target instruction. The assembler has four ways of obtaining the start and end offsets: The **graph** option uses a single label in the assembly where it expects to find a graph. The last instruction of the graph is also the last instruction of the range. The **range** options requires two labels for both the first and the last instruction of the range. The **after** option defines a single instruction range (start == last) and requires the actual number of instructions between the repeat and the instruction which should be repeated. The **label** option also defines a single instruction range but uses the label of this instruction to calculated the offset.

Mnemonics:

Multiple Instruction catch range:

repeat, graph (label);
repeat, range (label1,label2)
repeat_fixed (N), graph (label);
repeat_fixed (N), range (lab1,lab2);

Single instruction catch range:

repeat, after (W);
repeat, label (label);
repeat_fixed (N), after (W);
repeat_fixed (N), label (label);

N = 1..256, W = 3..67

The target instruction will be repeated for a given number of times. The repeat function is limited to a maximum of 2562 cycles. During both waiting and repeating all interrupts are masked. The repeat count can be hard coded as immediate data within the instruction word or it can be taken as a run_time variable from the repeat count register (SEQ_Repeat: cr56) where it can be stored by software (if SEQ_Repeat == 0 then the instructions are executed 1 time: they are 0 times repeated). A simple mechanism is provided to operate on vectors of arbitrary length: A long vector is subdivided in vectors of the length defined by SEQ_MaxRepeat: cr57 and a smaller 'tail' vector at the end. This maximum repeat count is the Vector Stride. Higher values in the repeat count register (up to 32768) will be replaced by this maximum repeat count. The repeat register is decremented by the maximum repeat count each time a conditional sequencer instruction refers to the **RRS** flag of the SEQ_Status register. This flag (Repeat Register Smaller) can be used to test if the contents of the repeat register is still larger than the maximum repeat count before decrementing it. In this case another repeat is needed. The Repeat function is always executed unconditionally.

11.5 Sequencer usage

11.5.1 The branch delay slots in the instruction address generation

The Imagine has a branch delay of 2 instructions. This means that a Jump, a Branch or a Call becomes effective after two more instructions following the instruction which caused the branch. These two instructions are called the *branch delay* instructions. They fall into the *branch delay slots*. Branch delay slots are typical for RISC processors. The number of delay slots is an indication for the level of pipelining in the instruction address generation and fetching.
The Timing figures below show that the current instruction (= fetched instruction) trails the value of the Program Counter by two cycles. These two cycles are used to output the instruction address, to access the instruction cache and to load the instruction into the instruction register. (The instruction is called *currently executed* instruction or *fetched* instruction when loaded into the instruction register.)

clock cycle	Contents of Program Counter	Currently Executed Instruction	Instruction cache addr. registers	Fetched Instruction	
170	N - 1	instr(N-3) (continue)			
171	N	instr(N-2) (continue)	N - 1		
172	N + 1	SUBTRACT(A,B)	N	instr(N-1)	
173	N + 2	JUMP X IF ZERO	N + 1	instr(N)	= JUMP X
174	X	instr(N+1) (b_delay 1)	N + 2	instr(N+1) delay 1	
175	X + 1	instr(N+2) (b_delay 2)	X	instr(N+2) delay 2	
176	X + 2	FIRST X instruction.	X + 1	instr(X)	= First X
177	X + 3	instr(X+1) (continue)	X + 2	instr(X+1)	

11.5.2 The usage of the internal program counter stack

The Internal stack consists of eight registers: a TOS (Top Of Stack) and seven further registers. It handles subroutine calls on high level (C language) and low level assembly code. High level calls use the TOS register as the place where the return address can be found to continue after exiting a C function. The return is executed by writing the externally saved value from TOS register back and performing a Return function.

Low level calls in the inner loops of assembly library functions can use the Tiny Stack for function calling without any overhead.
The Program Counter can be saved up to Six levels (the two remaining levels are reserved for interrupt calls). The TOS is obtained by the POP STACK function which moves the TOS register to the Address register. It can be restored with the PUSH STACK function which moves the Address register to the TOS.

Interrupt handling is the third task of the internal Stack. An Interrupt causes the Program Counter to be saved on the Internal Stack. The contents of the Interrupt table register is placed in the PC. A return from interrupt pops the value from the TOP register back to the Program Counter. A number of very frequent interrupts can be handled completely without state save and restore because of the internal micro stack..

11.5.3 Using the Imagine's ALU status for conditional control flow

The example above shows how status information from the Imagine ALU can be used for conditional control flow. The chapter on the ALU in the Imagine device specification manual shows how all typical C equations like A==B, A!=B, A>B, A>=B, A<B and A<=B for both signed and unsigned data types can be
translated into a combination of one ALU function followed by a conditional control flow instruction. If there are any other functions between the ALU function and the Control flow function, then the ALU should execute Nops to preserve the Status information.

11.5.4 The usage of the immediate data in the instruction field

Instruction code Ic[23:0]: jump, branch, call, subr

Absolute or Relative Address Offset [23:0]																							
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Instruction code Ic[23:0]: continue, push, pop, jump_reg, branch_reg, call_reg, subr_reg, call_int, return, access_IC

FVS	IAI	IRD	IWR	‘0000’				FLAG MODIFICATION FIELD								FLAG VALUE FIELD							
								MI 2	MI 1	‘0’	‘0’	‘0’	UF 2	UF 1	UF 0	MI 2	MI 1	‘0’	‘0’	‘0’	UF 2	UF 1	UF 0
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Instruction code Ic[23:0]: repeat instruction for vector processing

RCS	‘0’	Start repeat range [5:0]						‘00’		Last in repeat range [5:0]						Fixed repeat count option [7:0]							
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Option 1

The 24 bit dataword is either a 24 bit absolute address or a 24 bit signed offset depending on the addressing mode.

Option 2

The control flags (bits 0..7) in the control/status register can be individually modified with this option. A '1' in the *Flag Modify Field* allows the corresponding control flag [7:0] in the SEQ_Status register to be changed to the value in the *Flag Value Field*. Another way to modify the control flags and the state flag is the use of the corresponding 8 least significant bits of the SEQ_Address register. This option is typically used to restore the control flags before a return from interrupt. This option is selected with **FVS** = '1'. The 8 control flags are set with the values of the corresponding bits in the address register as is the value of the State Flag which also is saved in case of an interrupt.
A modification with the *Flag Modify/Value fields* has the highest priority so all nine control/state flags may be restored from the address register while individual flags are set with higher priory in the same instruction.

Option 2 contains a three bit field which is used for direct read and write operations in the instruction cache ram. (IAI, IRD, IWR) The use of this field is explained elsewhere.

Option 3

This option defines the repeat range:

If **RCS**: (Repeat Count Select) = '0' then SEQ_Repeat used during a repeat operation. If the repeat count is larger than then SEQ_MaxRepeat (cr57) then this value is used instead.
If **RCS**: (Repeat Count Select) = '1' then bits [7:0] of the Instruction Code are used during a repeat operation.

11.6 The program sequencer mnemonics

	THE **JUMP** FUNCTIONS	fun, mode
jump (label)	Jump Absolute to the address in the Instruction word	00 110
branch (label)	Jump relative, Add Instruction word offset to PC	00 111
jump_reg	Jump Absolute to the address in the address register	00 100
branch_reg	Jump relative, Add Address register offset to PC	00 101
jump_int	Software Jump to Interrupt	00 010
[**continue**]	'Default Dummy Jump' to PC+1, can be combined with flag modification	00 000

	THE **CALL** FUNCTIONS	fun, mode
call (label)	Call Absolute to the address in the Instruction word	01 110
subr (label)	Call relative, Add Instruction word offset to PC	01 111
call_reg	Call Absolute to address in the address register	01 100
subr_reg	Call relative, Add Address register offset to PC	01 101
call_int	Software Call Interrupt	01 010
push	Push Address register to the internal Stack	01 000

	THE **RETURN** FUNCTIONS	fun, mode
return	Return from Subroutine, Pop PC from Internal Stack	10 011
pop	Pop Address register from the Internal Stack	10 000

	THE **REPEAT** FUNCTIONS	fun, mode
repeat	Repeat instruction (Variable length Vector operations)	11 000

	THE **INSTRUCTION CACHE ACCESS**FUNCTIONS	fun, mode
set_IC_address	Place the contents of the Address register on the Instruction Address Bus	01 100
access_IC	Read \ Write the Instruction word on the selected location and continues.	10 011

	ORTHOGONAL OPERATION: DEFINE CONDITION FIELD
if (condition),	Condition field: can be combined with any instruction except the repeat function
if not (condition),	negated Condition field: can be combined with any instruction except the repeat function

ORTHOGONAL OPERATION: SET ADDRESS/DATA FIELD (IC0..23)

option 1

(label)	Define Contents of Ic0..23 as Address field or Relative Address offset, Can be combined with Jump, Branch, Call and Subr.

option 2

, flags (flags) Define Contents of Ic8..15 as a Flag Modify Field and Ic0..7 as the Flag Value Field
flags: **mask_int1, unmask_int1, mask_int2, unmask_int2, set_user_flag_0, reset_user_flag0,set_user_flag1, reset_user_flag1,set_user_flag2,reset_user_flag2.**

.

, flags_restore	Bits [7:0] from SEQ_Address (cr54) are restored in SEQ_Satus (cr52) (Ic23='1')
_read	Combined with access_IC: Read the Instruction code into the Imagine's user IC registers
_write	Combined with access_IC: Write the Instruction code from the Imagine's user IC registers
++	Combined with access_IC: Increment the SEQ_Address register

option 3

_fixed (count)	Define Repeat Count Field (Ic0..Ic4). This field is combined with the Repeat Instruction.
, after (count)	Define Start Range and Last in Range fields of the Repeat instruction.(fields are equal)
, label (label)	Identical to **after** but calculated as the difference of the instruction addresses.
, graph (label)	Start range is defined by label. Last instruction in the graph becomes the Last in range.
, range (lab1,lab2)	Start range is defined by label 1. Last in range is defined by label 2.

11.7 Vector processing control flow

11.7.1 Variable length vector processing

The Imagine has powerful variable length vector processing facilities which are briefly described here. Vector processing is implemented with the following properties of the Imagine:

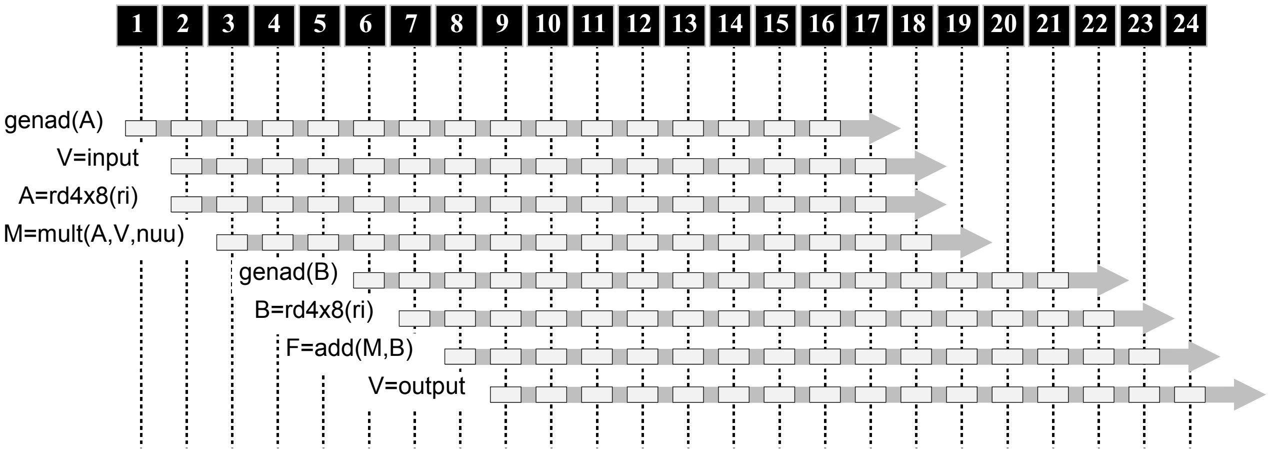
- ♦ The Repeat Instruction of the Sequencer
- ♦ Vector processing functional units
- ♦ Vector type register and memory access

11.7.2 The repeat instruction

During the processing of a variable length vector, the same operation is repeated for a variable number of times. A catch range can be defined where in instructions are catched and repeated. The repeat catch range is defined by two 6 bit numbers in the instruction word. The range must lay betwee 3 and 67 cycles away from the repeat instruction itself. The instructions in this range are executed from 1 to 256 times. Larger vectors can be subdivided into smaller sized ones. The total Vector Length is defined in SEQ_Repeat while the vector stride is defined in SEQ_MaxRepeat. The Program counter itself is repeated during the "Last in range" instruction. While the other instructions hold their own instruction fixed during repeat time.

11.7.3 Vector processing functional units

All data processing units like the ALU, the Barrel shifter and the Multiplier/Accumulator can perform vector operations. Each one can perform operations each cycle and the units are interconnected by a flexible bus structure which allows a pipeline to be set up from the reading of the operands, trough various function unit to the writing of the results.



The pipelined processing above is generated by the following assembly code program:

```
repeat, graph (merge_ARGB);;;
merge_ARGB:
genad(A)=>V=input,A=rd4x8(ri)=>M=mult(A,V,nuu)===>genad(B)=>B=rd4x8(ri)=>F=add(M,B)=>V=output;
```

11.7.4 Vector type data storage access

All types of data storage known to Imagine have a vector access mode. There are four types of data storage:

♦The Vector access memory unit	One vector can be read and one can be written simultaneously.
♦The Data access memory. unit	One vector can be read **or** written.
♦The three port register file.	2 vectors can be read and 1 can be written simultaneously
♦The Multiplier/Accumulator register file.	One vector can be read and one can be written simultaneously

All these units can operate simultaneously.

11.8 The multimedia interrupt handler in the Imagine 2

11.8.1 Programmers view:

The status of a pending Interrupt of level 1 or level 2 is visible with the IP1 and IP2 (Interrupt Pending) bit in the SEQ_Status register. The servicing interrupt causes the MI1 or MI2 (Interrupt Mask) bit in the sequencer status register of the Imagine to be set. It prohibits other interrupts of that level and lower priority levels from causing interrupt calls. These status bits can be set and reset by software.

A '1' for IP1 or IP2 means that the interrupt of that level is detected, acknowledged and that the interrupt vector belonging to this interrupt is loaded and visible in bits 3..7 of the interrupt table register.

The two bits are mutually exclusive. It is possible that IP2 belonging to the lower priority level 2 interrupt is temporary overruled by a later arrival of a level 1 interrupt before it gets the chance to be handled (which is just what we want). The IP2 becomes temporary '0' and bits 3..7 of the interrupts now show the interrupt vector of the level 1 interrupt. The state of the suppressed level 2 interrupt will be restored at the moment the level 1 interrupt service call has been made and no other level 1 interrupt has shown up in the meantime.

The Interrupt address is assembled within the interrupt table register. Bits [23:8] of this register contain the base address of the interrupt table. The lowest eight bits of the address for the interrupt call are depending on the Interrupt Vector provided by the served interrupt requesting device. Bit 7 is depending on the level of the interrupt. Bit3..bit6 are replaced with the interrupt vector. Bit 2 depends on the way the interrupt routine is called: by hardware or by software. The lowest two bits are always zero.

SEQ_Interrupt[7]	:	Level 1: 'LEV:=0',	Level 2: 'LEV:=1'
SEQ_Interrupt[6:3]	:	Interrupt Vector:	
SEQ_Interrupt[2]	:	Hardware: 'S/H:=0',	Software: 'S/H:=1'.

When the sequencer detects a pending interrupt which is not masked and not disabled by functions like Vector memory operations and Repeat functions, and the current instruction is not a sequencer instruction, then it executes a call to the interrupt service routine. The Program Counter of the previous instruction is pushed on the tiny internal Stack and the contents of the Interrupt table register is copied to the Program Counter:

SEQ_PrCounter → Top of Internal Micro Stack, and
SEQ_Interrupt → SEQ_PrCounter

The two instructions PC-1 and PC which were already in the instruction pipeline are disabled and discarded.

The opposite process takes place during a Return from interrupt operation. The return address is popped back from the tiny Stack, placed into the Program Counter and the Instruction address output register. This return should reset the MI**n** control flag of it interrupt priority level which was set by the hardware at the start of the interrupt service routine call.
Example:

return, flags (unmask_int1);

11.8.2 Multiple interrupts without repeated state saving and restoring:

The level 1 interrupts will typically come from video fifo, line and raster interrupts for video output and input and DirectX cache line requests. The interrupt allocation table has reserved interrupts for two different simultaneous video formats. Handling of all the interrupts should be possible within 1.0 to 1.5 microseconds at most. The Vector access generator has multiple display pointers with auto increment capabilities to allow interrupt handling times of circa 100 nanoseconds to preserve as much time as possible for graphics processing instead of interrupt handling. These interrupts are fast because there is no need for state saving and restoring.

Most of the other Multi Media I/O interrupts and communication interrupts need software interference and the state of the processor needs to be saved at the start and restored at the end of the interrupt. Multiple interrupts of different sources can clash and can (once in a while) occur all at the same time. Repeated state saving and restoring take to much time and would degrade the real time performance of the Imagine. However the processor is equipped with a special mechanism which allows the handling of multiple interrupts in one go.

The first interrupt is called by the hardware and therefor takes the hardware entry in the interrupt table. This entry saves all the state of the processor which is needed by any of the interrupts. At the end of the routine and before restoring the state of the processor a test is made if another (level 1) interrupt is pending and if so a (software) jump is made to the routine for this particular interrupt. The software entry for the interrupt is now taken which points to exactly the same routine but skips the initial state saving instructions.
This goes on until the last pending interrupt has been served which will restore the processor state and do a return. This procedure is demonstrated with a 'simulator' run which demonstrates a link from an interrupt 6 to an interrupt 7.

```
CONTENTS INTERRUPT TABLE:
.......;
.......;
entry_int5_hardware:jump (int5h);;;;
entry_int5_software:        jump (int5s);;;;
entry_int6_hardware:jump (int6h);;;;
entry_int6_software:        jump (int6s);;;;
entry_int7_hardware:jump (int7h);;;;
entry_int7_software:        jump (int7s);;;;
.......;
.......;
END CONTENTS INTERRUPT TABLE.
```

```
SIMULATOR RUN:
/* hardware interrupt 6 */

entry_int_6_hardware:
            jump (int6h);
            .........;   /* branch delay 1
            .........;   /* branch delay 2
int6h:  /* save state of processor
            .........;   /* save state
            .........;   /* save state
            .........;   /* save state
            .........;   /* save state
int6s:  /* Actual Interrupt service instr.
            .........;   /* interrupt 6 code
            .........;   /* interrupt 6 code
            .........;   /* interrupt 6 code
            .........;   /* interrupt 6 code
            if (interrupt1), jump_int;
            .........;   /* branch delay 1
            .........;   /* branch delay 2
entry_int7_software:
            jump (int7s);
            .........;   /* branch delay 1
            .........;   /* branch delay 2
int7s:  /* Actual Interrupt service instr.
            .........;   /* interrupt 7 code
            .........;   /* interrupt 7 code
            .........;   /* interrupt 7 code
            .........;   /* interrupt 7 code

            if (interrupt1), jump_int;
            .........;   /* branch delay 1
            .........;   /* branch delay 2
int7restore:  /* no other interrupt
            .........;   /* restore state.
            .........;   /* restore state.
            .........;   /* restore state.
            .........;   /* restore state.
exint7: return, flags (unmask_int1)
            .........;   /* branch delay 1
            .........;   /* branch delay 2
xxxxx:      .........;   /* interrupted instr.

END SIMULATION.
```

cr55: The Interrupt service Routine Pointer register

Interrupt Table Address [23:0]																L E V	Interrupt Vector [3:0]				S / H	‘00’	
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

11.9 The status / control register

SEQ_Status

The Status and control Register contain a number of flags which are related to the program flow of the Imagine. The Interrupt handler provides/uses the largest part, but Floating Point (FPE), the Image Mask generator (IMZ) and the Vector Access unit (IAB) and the Sequencer (RRS) also provide flags.

The highest 8 bits reflect the status of the hardware, either the interrupts status or the status of some of the working registers. These bits are read only. The lowest 8 bits are used for control purposes.
These flags can be read and written. The **flags** command which can be combined with a number of sequencer instructions can be used to set selected individual flags with either immediate value in the instruction word or run-time value coming from the corresponding bits in the address register. The instruction contains 8 bits which can be set to enable the modification of individual flags and 8 bits which can contain the values to be given to the flags. Alternatively the last 8 bits can come from the lowest 8 bit of the SEQ_Address register. Writing directly to the SEQ_Status register is not possible.

cr52: Sequencer status / control register

IP2	IP1	‘0’	FPE	IAB	IMZ	RRS	‘0’	MI2	MI1	‘000’			UF2	UF1	UF0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Read Only Status Flags:

These flags which can be consulted by the sequencer for conditional instructions reflect external and internal status information. These flags do not need to be saved during interrupts since they reflect non changeable or indirect (redundant) information (e.g. Image Mask zero).

The Interrupt Pending flags **IP1** and **IP2** are '1' when an interrupt has arrived externally, has been Acknowledged and an Interrupt Vector has been received and placed in bits [6:3] of the Interrupt table register.
The bits are reset when the Interrupt Service Routine is called by the hardware or when either the jump_int or call_int functions are executed by software. The two bits are mutually exclusive because only one vector can be placed in the Interrupt Table register. An interrupt of the higher priority level 1 can temporary overrule the **IP2** flag. The **IP2** flag will be temporary '0' and the contents of bits [6:3] of the Interrupt Table register will be temporary replaced by the level 1 Interrupt vector.
The State of the level 2 interrupt is restored once the level 1 interrupt service routine is called.

IPn: = '1': No Interrupt Waiting.
IPn: = '0': Interrupt Waiting

Interrupts are acknowledged externally but stay pending when:
1 The MI**n** (Mask Interrupt flag) is set.
2 The IMAGINE executes non interruptable code.

The Image Data Access Busy flag: **IAB** indicates that the Image Memory Access Generator is Busy. It functions as an Interrupt Mask to avoid Interrupts in the Middle of an Access.
IAB: = '1': Access Busy.
IAB: = '0': Not Access Busy.

The Image Mask Zero flag: **IMZ** is high if the entire Image Mask is zero: all 4x64 bits are '0'. This implies that any write action to DRAM or VRAM is superfluous since no pixel will be written anyway.
IMZ: = '1': Image Mask Zero.
IMZ: = '0': Image Mask Not Zero.

Repeat Register Smaller: **RRS**
Used for the handling of long vectors (> 64..256). The length-1 of the vector is placed in the Repeat Register. The RRS flag is true ('1') if the contents of the Repeat register is equal to or smaller than the maximum repeat count.

Program example:

```
loop_label
repeat, graph ( graph_label );
image_vector( write, quad_byte,image1, ...);
...............................;
...............................;
graph_label:  <repeated dataflow graph>;
...............................;
branch (loop_label), ifnot(repeat_smaller);
```

The Repeat register is automatically decremented with the maximum repeat count each time a conditional sequencer instruction refers to the RRS bit from the Control/Status register (the register is decremented independent of the value of the condition, true or false).

Control Flags:
These flags are used in various real time control operations.

The Mask Interrupt flags: **MI1** and **MI2** are set to '1' when an interrupt service call is effectuated. They inhibit other interrupt requests from causing an unwanted call to an interrupt service routine. The MI2 flag masks level 2 interrupts while the MI1 flag masks level 1 and level 2 interrupts.

The flags can be set and reset by software.
If MI1 is set then level 1 interrupts are still externally acknowledged but no Interrupt call is made by the sequencer. The level of such an interrupt is placed in the Interrupt table entry register and the IR1 flag in the status register is set to '1' so software can observe pending level 1 interrupts. Level 2 interrupts are not externally acknowledged when MI1 is set.

MIn: = '1': Mask Interrupt.
MIn: = '0': Do Not Mask Interrupt.

UF2, UF1, UF0: User Flags 2, 1 and 0.
All eight control flags can be modified by the (assembly code) programmer. Some of them are designated to special functions while others are reserved for future purposes and should be left '0'. The three User Flags are left to the user. They can be used in highly optimised innerloops to select between various options with minimal overhead (example 1) or they can be used to temporary save the state of other flags and use the saved information later for a conditional control flow instruction (example 2)

Example 1:

```
If ( user_flag2 ), branch ( module2a);
```

Example 2:

```
if ( carry ),  flags ( set_user_flag1);
......;
......;
......;
if  ( user_flag1),  subr ( carry_detect );
```

11.10 Direct read and write accesses to the instruction cache

A secondary activity of the sequencer is to handle low level read and write accesses of 64 bit instruction data directly to the Instruction cache ram. The Data for the instruction memory stems from two 32 bit user registers on the Imagine which can be freely read and written to: ICA_Low (cr60) and ICA_High (cr61). The Data stored in these registers can be stored into the instruction cache ram and Instruction cache data can be read back into these registers. Writing to the instruction cache happens typically after a reset / power up during the booting process when the caches are not yet enabled. The instruction address on which the access takes place is provided the SEQ_Address register (cr54). This address can be auto-incremented during the Instruction memory access.

Two Instructions are reserved for down(up)-loading of instruction cache data:

Set_IC_address_...;
IC_access_...;

These two instructions always have to be applied together, one after the other and unconditional.
Set_IC_address... places the Instruction code address which is stored in the address register in the PC and saves the PC+1 on the internal Stack. The access can either be an Instruction read or an instruction write access:
Set_IC_address_read,
Set_IC_address_write
Using these mnemonics causes the flags IRD and IWR to be set in the instruction word (IC[21], IC[20]).

IC_access_.. outputs the control information which handles the transfer and pops the PC+1 back from the internal register Stack. The last branch delay instruction is disabled because this instruction was loaded with the use of an irrelevant instruction address.

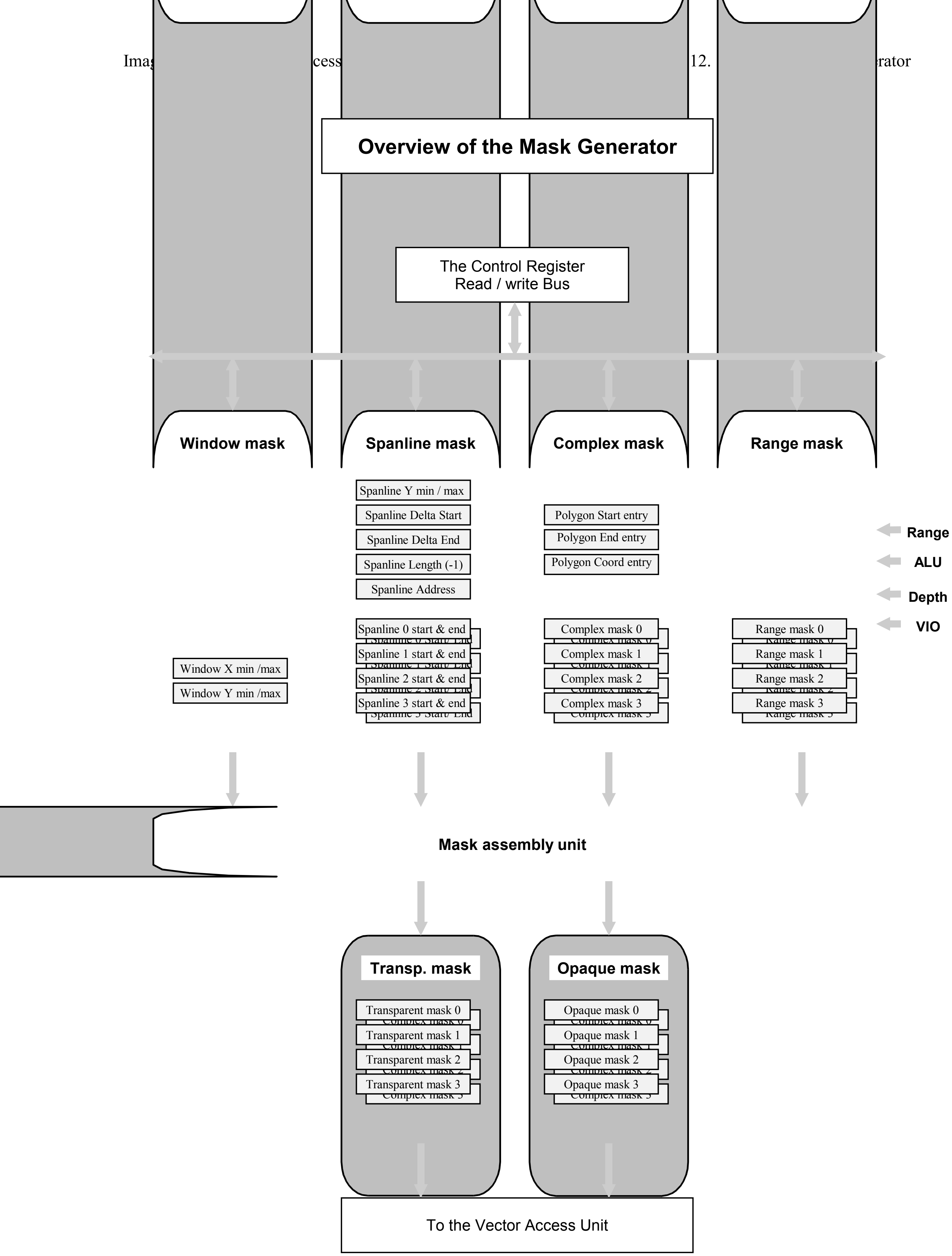
IC_access can be accompanied with ++.
This causes the IAI flag to be set in the instruction word (IC[22]). **IC_access++** copies the incremented PC used for the read/write operation back to the address register where it can be used for the next transfer options:

Chapter

12. THE MASK GENERATOR

The Image Mask Generator
contains a very powerful set of units to deal with a very wide range of graphics primitives. A mask can define the outlines of characters, polygons, arbitrary shapes, window borders etc. A drawing operation will very often need a combination of several mask functions. A simple example shows the amount of code needed to draw a single pixel, e.g.: A Textured triangle which is rendered in OpenGL: A pixel may be drawn only if: It is inside a visible window. If it is inside the area confined by the triangle. If it is in front of previously drawn pixels. If it is between the front and back clipping planes. If the Polygon Stipple pattern for this pixel is '1'. If the Alpha value of the texture is not to low, et cetera . All These conditional operations consume large amounts of instructions, and thus cycles, on standard processors.

The Imagine however can process up to four pixels per clock cycle!
The powerful mask generator plays a central role in this achievement in co-operation with the HISC principles which are the base of the architecture.

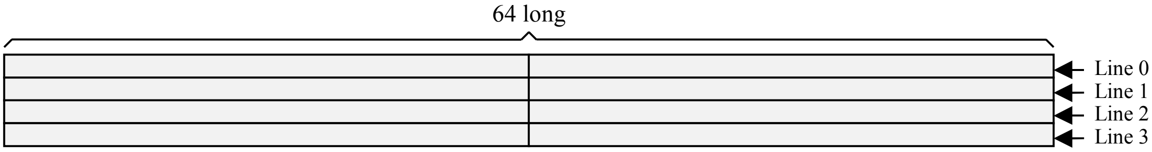


Overview of the Mask Generator

12.1 introduction

12.1.1 The image masks

The Image mask is applied used during write operations of the vector access unit. It corresponds to up to four spanlines of up to 64 pixels. The mask is arranged in eight 32 bit registers with each bit representing a individual pixel. There are several masks like this in the mask generator, two of them are result masks which are connected to the vector access unit. The transparent mask bits are used as write enable for the pixels which are written while the opaque mask bits can be expanded to the 32 bit databus.



12.1.2 The vector access unit

The capability possibility of this unit to access external memory in a vector mode ensures a high basic speed from and to Images. One input word and one output word can be transferred per clock cycle. Internal fifos will read the entire input vector first and then write the buffered output vector. A 200 MHz Imagine 2 with 100 MHz external SGRAM reaches a 1.6 Gigabyte bandwidth for vector I/O accesses. Pixels within stored images can be read or written in the form of vectors. These vectors can have sizes varying from 1 to 64 words. A vector covers a horizontal strip in an image on the screen. The first word is at the left-most position of the strip, while the last word is at the right-most side. Each word in a vector can contain one or several pixels. A 32 bit for example can contain four 8 bit pixels which are on different vertical positions (line 0 ... line 3). The pixel with byte number 0, which contains the eight least significant bits, is at the top position while the pixel with byte number 3 is at the bottom position. This arrangement is consistent with the industry norms where the x-axis is increasing from left to right while the y-axis is increasing from top to bottom. A very important feature of the Imagine is that is can access these 2-dimensional vectors or stripes with the starting point at any X,Y location in the image. This unique feature allows a straight forward implementation of a very wide range of graphics and image processing functions.

12.1.3 The usage of the image mask

A strip drawn by a two-dimensional vector has a rectangular nature and only few graphics primitives are rectangular. Characters, Polygons, Circles or arbitrary shaped objects cannot efficiently be drawn with rectangles. In many cases a graphical primitive can obscure other ones partly or wholly. A classical example are the 3D textured triangles which overlap the triangles behind them. A character can be drawn with its background or its foreground set to 'transparent' so that it shows the image below it. Graphics standards like Microsoft's Direct Draw define transparent colours, pixels which have a transparent colour are not written when the image is copied (source transparency) or can be overwritten by another image (destination transparency).

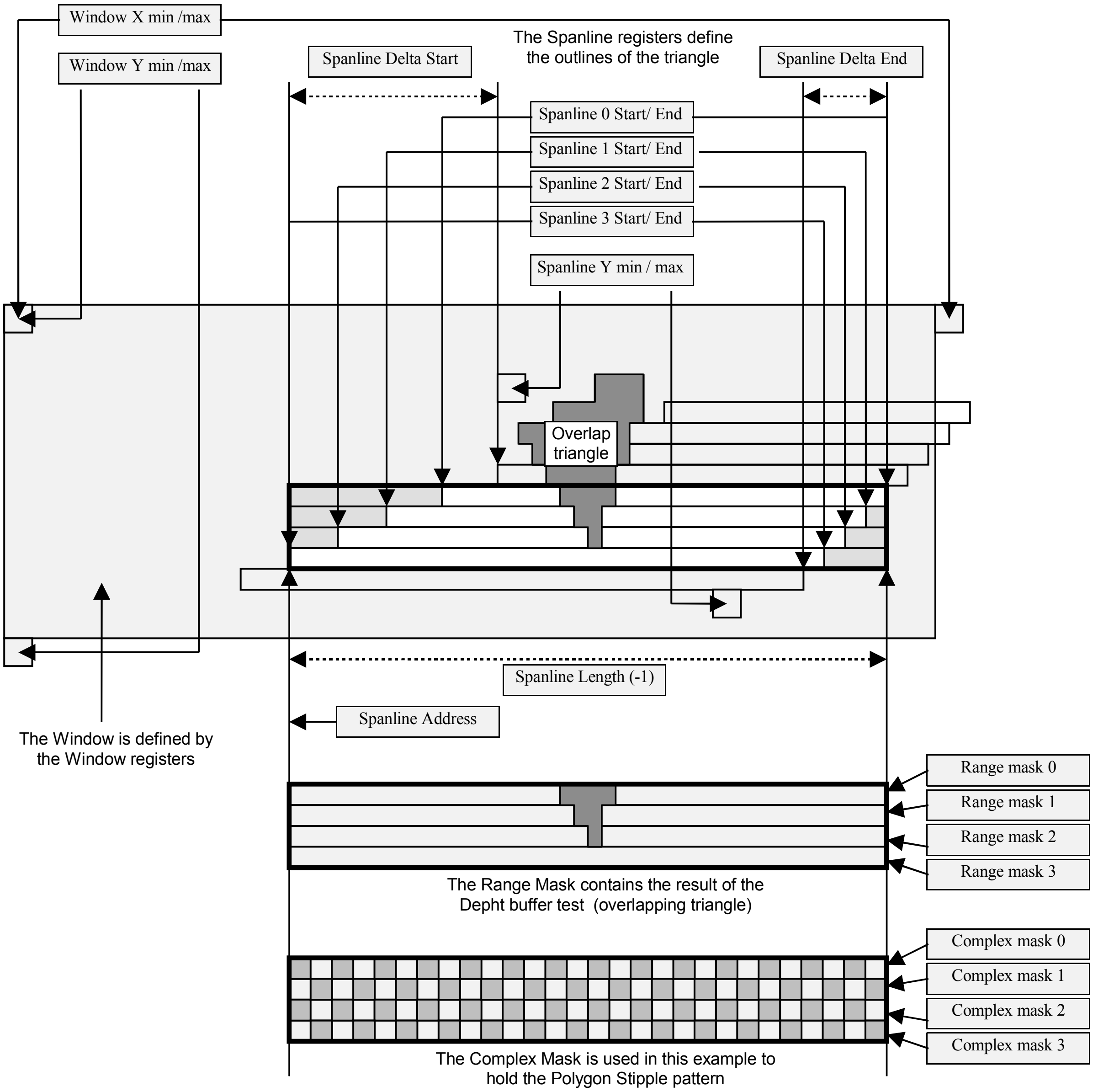
12.1.4 The image mask and its construction

All previously mentioned drawing operations can be realised by introducing a (2-dimensional) Image mask. Objects and graphics primitives are rendered with the aid of this mask which is stored in up to eight 32 bit registers. The actual decision of pixels is

written or not to the destination is hold in the Image mask registers. The registers contain mask bits for a total of 64 times 4 pixels. The mask bits represent a Boolean decision related to many elementary situations listed below. In practice the final image mask used to render a graphics primitive will be the result of several of these Boolean decisions combined together.

- The pixel is falling inside **OR** outside the area occupied by a polygon (which may be self intersecting)
- The pixel is within a certain colour range **OR** falls outside the range.
- The alpha value of a pixel is within a range **OR** falls outside the range.
- The Z buffer value is before the closest pixel and behind the nearest visible point **OR** it is invisible.
- The pixel belongs to the body of a character **OR** to the background of the character.
- The pixel is within the window **OR** it falls outside the window.
- The pixel is inside **OR** outside a scanline defined object.

Example how the mask generator can be used to draw Depth buffered Stippled Triangles



Window X min /max

Window Y min /max

Spanline Delta Start

The Spanline registers define the outlines of the triangle

Spanline Delta End

Spanline 0 Start/ End

Spanline 1 Start/ End

Spanline 2 Start/ End

Spanline 3 Start/ End

Spanline Y min / max

Overlap triangle

Spanline Length (-1)

Spanline Address

The Window is defined by the Window registers

Range mask 0

Range mask 1

Range mask 2

Range mask 3

The Range Mask contains the result of the Depht buffer test (overlapping triangle)

Complex mask 0

Complex mask 1

Complex mask 2

Complex mask 3

The Complex Mask is used in this example to hold the Polygon Stipple pattern

12.2 The image mask control registers

12.2.1 The mask generation control registers:

MSK_Control1, MSK_Control2
These registers determine the operation of the Image Mask generator. They can be written as control registers or alternatively via Image Mask Generator Instructions which define the contents with the lower 32 bit of the Instruction code.

12.2.2 The Window mask control registers

MSK_Window_X and MSK_Window_Y
These two registers contain the definition of the drawing window. They can be used to disable drawing outside a particular rectangle. MSK_Window_X contains the minimum X co-ordinate in its highest 16 bits and the maximum X co-ordinate (-1) in the least significant 16 bits. MSK_Window_Y does the same for the Y co-ordinates. The 16 bit values are interpreted as signed integers. Negative X and Y co-ordinates do exist and are handled correctly when compared with the contents of these registers. The Window registers should not contain negative values. The valid co-ordinate range is defined from -32768,-32768 to +32767,+32767. The minimum value contains the first pixel belonging to the window while the maximum points to the first pixel outside the window.

12.2.3 The Spanline mask control registers

MSK_SpanStart, MSK_SpanEnd, MSK_SpanLines,
Two sets of four spanline registers contain the 32 bit values in signed 16.16 fixed point which represent the startpoints and endpoints for four horizontal spanlines lines. MSK_SpanStart, MSK_SpanEnd provide post incremental access to these registers which are indexed by the SPAN_RW_PTR[1:0] field in control register MSK_Control1 [5:4] The control register MSK_SpanLines reads the 16 bit integer parts of both Start and End value when read and writes the same fields when written. (The fractional parts are set to zero)

MSK_Spanline_Y
This register defines a vertical window for spanline objects . MSK_Spanline_Y contains the minimum Y co-ordinate in its highest 16 bits and the maximum Y co-ordinate (-1) in the least significant 16 bits. This vertical window is combined with the spanline start and end points to define a spanline object

MSK_DeltaStart, MSK_DeltaEnd
define 32 bit values in signed 16.16 fixed point format which can be added to 1, 2 or all 4 spanline Start registers and corresponding spanline End registers. The instruction code spanlines++ will activate this function which takes as many cycles as there are spanline Start / End register pairs which are incremented. The registers to be incremented are selected with the MASK_MOD_MAP[2:0] field in control register MSK_Control1 [22:20] which can select the four individual lines, line pair 0,1 or pair 2,3 or all four lines at once

MSK_SpanLength, MSK_SpanAddr
These two read only registers contain calculated values which can be copied directly into the SEQ_Repeat control register (MSK_SpanLength) and bits [15:0] of the VAU_Image1 control register (MSK_SpanAddr). The first value represents the difference between the lowest spanline Start co-ordinate and the highest spanline End co-ordinate of all the spanlines which are enabled by the MASK_MOD_MAP[2:0] field in control register MSK_Control1 [22:20] This number is equal to the repeat count needed to write 1, 2 or 4 spanlines with a single vector. The second value is equal to the lowest spanline Start co-ordinate and thus represents the start X co-ordinate of the vector mentioned above. Remark: the lowest spanline Start co-ordinate and the highest spanline End co-ordinate are selected by looking at the sign of the Start slope and the End slope which are defined by MSK_DeltaStart, MSK_DeltaEnd This method is designed for (and thus limited to) triangles for 3D graphics applications

cr88: MSK_Control1 Mask generator control register 1

<table>
<tr><td>I
M
E</td><td colspan="8">‘00000000'</td><td colspan="3">MASKMOD MAP [2:0]</td><td colspan="2">MASK SIZE [1:0]</td><td colspan="2">MASK LWPTR [1:0]</td><td colspan="4">MASK ENABLES [3:0]
WM SM RM CM</td><td colspan="2">‘00'</td><td colspan="2">CPLX ASM [1:0]</td><td colspan="2">‘00'</td><td colspan="2">SPAN RW-PTR [1:0]</td><td colspan="2">‘00'</td><td colspan="2">MASK RW-PTR [1:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr89: MSK_Control2 Mask generator control register 2

<table>
<tr><td colspan="8">‘00000000'</td><td colspan="6">RANGE INPUT POINTER [5:0]</td><td colspan="2">RANGE OP [1:0]</td><td colspan="2">‘00'</td><td colspan="3">RANGE INPUT MAP [2:0]</td><td>R
I
O</td><td colspan="2">RANGE SEL [1:0]</td><td>‘0'</td><td colspan="2">CPLX ALPHA [1:0]</td><td>‘0'</td><td colspan="2">SF [1:0]</td><td colspan="2">LS [1:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr90: MSK_Window_X Window X minimum / maximum

<table>
<tr><td colspan="16">WINDOW X CO-ORDINATE MINIMUM VALUE [15:0]</td><td colspan="16">WINDOW X CO-ORDINATE MAXIMUM VALUE [15:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr91: MSK_Window_Y Window Y minimum / maximum

<table>
<tr><td colspan="16">WINDOW Y CO-ORDINATE MINIMUM VALUE [15:0]</td><td colspan="16">WINDOW Y CO-ORDINATE MAXIMUM VALUE [15:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr93: MSK_Spanline_Y Spanline Y minimum / maximum

<table>
<tr><td colspan="16">SPANLINE Y CO-ORDINATE MINIMUM VALUE [15:0]</td><td colspan="16">SPANLINE Y CO-ORDINATE MAXIMUM VALUE [15:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr94: MSK_PolyStart Polygon Start Co-ordinate entry

<table>
<tr><td colspan="32">POLYGON START CO-ORDINATE (16.16 FIXED POINT) [31:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr95: MSK_PolyEnd Polygon End Co-ordinate entry

<table>
<tr><td colspan="32">POLYGON END CO-ORDINATE (16.16 FIXED POINT) [31:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr96: MSK_PolyCoord Polygon Start & End co-ordinates

<table>
<tr><td colspan="16">POLYGON START CO-ORDINATE (16 BIT FIXED POINT) [15:0]</td><td colspan="16">POLYGON END CO-ORDINATE (16 BIT FIXED POINT) [15:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr97: MSK_SpanStart Spanline Start Co-ordinates (4 registers)

<table>
<tr><td colspan="32">SPANLINE START CO-ORDINATE (16.16 FIXED POINT) [31:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr98: MSK_SpanEnd Spanline End Co-ordinates (4 registers)

<table>
<tr><td colspan="32">SPANLINE END CO-ORDINATE (16.16 FIXED POINT) [31:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr99: MSK_SpanLines Spanline Start & End co-ordinates (4 registers)

<table>
<tr><td colspan="16">SPANLINE START CO-ORDINATE (16 BIT FIXED POINT) [15:0]</td><td colspan="16">SPANLINE END CO-ORDINATE (16 BIT FIXED POINT) [15:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

cr100: MSK_DeltaStart Slope of the Spanline Start Co-ordinates

SLOPE OF THE SPANLINE START CO-ORDINATES (16.16 FIXED POINT) [31:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr101: MSK_DeltaEnd Slope of the Spanline End Co-ordinates

SLOPE OF THE SPANLINE END CO-ORDINATES (16.16 FIXED POINT) [31:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr102: MSK_SpanLength Spanline Vector Length (-1)

'0000 0000 0000 0000'																SPANLINE VECTOR LENGTH (-1) +32383..-32384 [15:0]															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr103: MSK_ SpanAddr Spanline Vector X Co-ordinate

SPANLINE VECTOR SPANLINE VECTOR X CO-ORDINATE [15:0] CO-ORDINATE																SPANLINE VECTOR X CO-ORDINATE [15:0]															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr104: MSK_CplxAlpha Complex Mask registers (4 x 2 registers)

COMPLEX MASK REGISTERS 4 times [63:0]																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr105: MSK_RangeClip Range Mask registers (4 x 2 registers)

RANGE MASK REGISTERS 4 times [63:0]																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr106: MSK_Transp Transparent Mask registers (4 x 2 registers)

TRANSPARENT MASK REGISTERS 4 times [63:0]																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr107: MSK_Opaque Opaque Mask registers (4 x 2 registers)

OPAQUE MASK REGISTERS 4 times [63:0]																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

12.2.4 The Range mask control registers

MSK_RangeClip
The four times two range mask registers contain Boolean data for a rectangular block of 64x4 pixels which can be used for the construction of the transparent and opaque image mask registers. This mask can gather status information of the Range Unit in the Multiplier / Accumulator, status information from the ALU, results from the Depth compare test of the 3D graphics pipeline or Alpha test information from the Vector I/O unit. The eight registers are accessible via control register MSK_RangeClip. If the mask is defined as 64 bit then the incremental order is: maskline 0 bits [31:0], then bits [63:32], then maskline 1 bits [31:0], then bits [63:32] et cetera and wrapping back to the first register at the end. The maskline number is given by the MASK_RW_PNT[1:0] field in control register MSK_Control1 [1:0] and the bits are selected by the MASK_LW_PTR field in MSK_Control1 [16]. If the mask is defined as 32 bit then the MASK_LW_PTR field is ignored

12.2.5 The Complex mask control registers

MSK_CplxAlpha
These registers contain typically 64 4-bit values which can be used in more complex mask calculations. Alternatively it contains Boolean data for a rectangular block of 64x4 pixels similar to the range mask registers. In its native operation mode it works with 4 bit
values which can be sums of line crossings for complex polygons according to either the odd/even rule or the winding rule. The eight registers are accessible via control register MSK_CplxAlpha. If the mask is defined as 64 bit then the incremental order is: maskline 0 bits [31:0], then bits [63:32], then maskline 1 bits [31:0], then bits [63:32] et cetera and wrapping back to the first register at the end. The maskline number is given by the MASK_RW_PNT[1:0] field in control register MSK_Control1 [1:0] and the bits are selected by the MASK_LW_PTR field in MSK_Control1 [16]. If the mask is defined as 32 bit then the MASK_LW_PTR field is ignored

MSK_PolyStart, **MSK_PolyEnd**, **MSK_PolyCoord**
These write only registers serve as entry points for calculated coordinate data and are used to calculate the area covered by complex polygons. The co-ordinates are translated to 64 bit coverage masks which are send to the Complex Mask generator. The Polygon Start coordinate and Polygon End coordinate entries (MSK_PolyStart, MSK_PolyEnd) expect significant coordinate data in the 16 most significant bits (signed integer). The lowest 16 bits are considered as the fractional coordinate parts are discarded. Alternatively MSK_PolyCoord can be used to enter coordinate data. This entry expects both a Start and an End coordinate entry in compacted form. The 16 most significant bits should contain the Start point. These registers can not be read back.

12.2.6 The Result mask registers

MSK_Transp: The Transparent Image Mask registers
The transparent Image Mask contains the calculated Image Mask result for a rectangular block of 4x64 pixels. This Mask can be used for write operations in which case the bits can inhibit writing of individual pixels. The bits are sent 4 at a time to the Vector Access Unit which write pixel vectors to external memory.

MSK_Opaque: The Opaque Image Mask registers
The opaque Image Mask contains the calculated Image Mask result for a rectangular block of 4x64 pixels. This Mask can be used for write operations in which case the bits can select the 32 bit colour/mask data which is written to external memory via the Vector Access unit.

The two sets of eight registers are accessible via control registers MSK_Transp and MSK_Opaque. If the mask is defined as 64 bit then the incremental order is: maskline 0 bits [31:0], then bits [63:32], then maskline 1 bits [31:0], then bits [63:32] et cetera and wrapping back to the first register at the end. The maskline number is given by the MASK_RW_PNT[1:0] field in control register MSK_Control1 [1:0] and the bits are selected by the MASK_LW_PTR field in MSK_Control1 [16]. If the mask is defined as 32 bit then the MASK_LW_PTR field is ignored

12.3 The function specific mask generators

The final image mask is constructed by combining four major function specific mask generators:

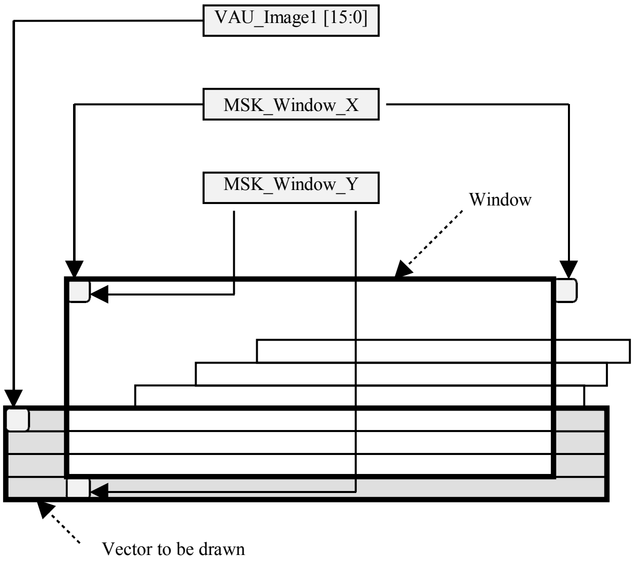
- **The Window Mask**
- **The Spanline Mask**
- **The Range Mask**
- **The Complex Mask**

All these units calculate various masks in parallel, which are then combined together by the mask assembly unit. If any of the input registers for the various mask units is changed this will result in an immediate change of the image mask. However the image mask registers which are used for the rendering itself will only take over the new Mask when it is instructed to do so. This will typically be after the end of a vector draw operation and before the start on the next one which will use the image mask registers. This means that you can construct the new Image mask while the previous one is used in the drawing operation.

12.3.1 The Window mask generator

Window systems in general restrict the locations where a program may render its graphics primitives to rectangles or regions consisting of a list of rectangles. The window mask generator generates a mask with bits set to 1 for pixels inside the window.The mask can then later be used to enable or disable writing. The term *scissoring* is used if this technique is used to simplify 'software only' clipping.

The simplest polygon, a trapezium (two trapezia make one arbitrary triangle) can intersect in 48 different ways with a rectangular window. All these cases have to be taken into account and calculations have to be done with subpixel precision to avoid artefacts. These extra calculations give a lot of overhead in the case of smaller polygons. The window mask generator allows to define a rectangular area to which all rendering primitives are clipped by hardware. The software renders all pixels of the primitive anyway but the actual writing into Image memory is inhibited by the hardware. Handling all polygons by scissoring can have counter effects (imagine a triangle with a surface 100 times that of your visible window) You can use the
following practical method: Do a simple test to give an indication of the size of the polygon. If the size is big then calculating intersection points with the window rectangle does not represent a significant overhead. If it is small, do a test if the polygon is either totally outside the window **or** partly or wholly inside the window. If the latter is the case you can render it directly with the scissoring technique.



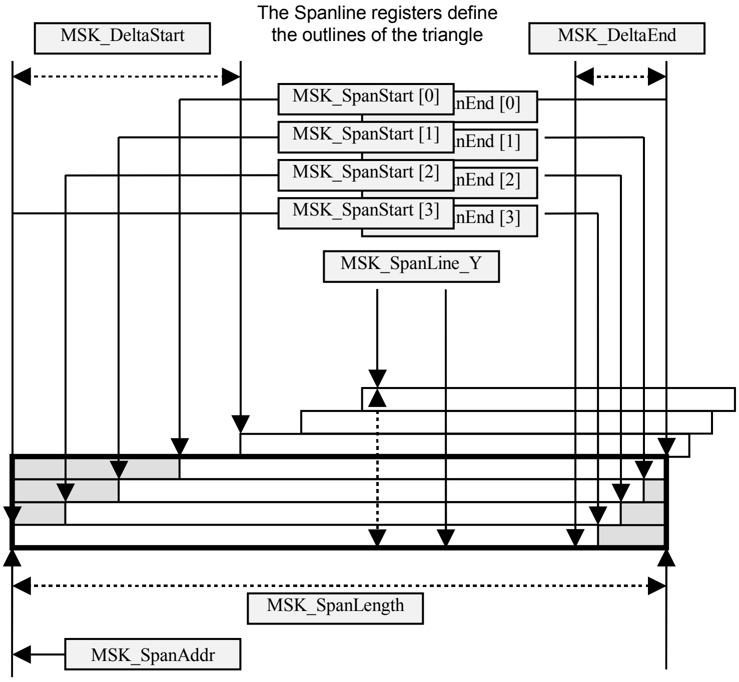
MSK_Window_X: bits [31:16]: X minimum / bits [15:0]: X maximum coordinate. (16 bit signed)
MSK_Window_Y: bits [31:16]: Y minimum / bits [15:0]: Y maximum coordinate. (16 bit signed)

The minimum co-ordinates point to the first location which is *inside* the window while the maximum co-ordinates point to the first location *outside* the window. The window co-ordinates are compared to the Reference X and Y co-ordinates found in control register VAU_Image1: Image address pointer 1. This register contains the top left position of the vector to be drawn or read from the image memory. If the Window Mask generator is enabled, then the pixels outside the window are masked during a vector write operation. The end result needed for the construction of the Transparent and/or Opaque mask needs 4 x 64 bits representing 4 lines of 64 pixels. The 256 bit mask can be used in the Mask assembly unit in which it is combined with the result from the other mask generators.

12.3.2 The Spanline mask generator

The purpose of Spanlines: Many basic graphics primitives including convex polygons, can be defined with a Spanline representation. A spanline is a horizontal line with a start point and an end point. A spanline shape defines a list of start and end points of horizontal lines on top of each other.

An arbitrary shape can be represented with a single spanline shape if no horizontal line contains disjunct areas. If this is the case the shape can be represented by a combination of several spanline shapes. A pixel belongs to a spanline by definition when it lays on top of or after the start point and before the scan line's end point. The Image Mask generation unit contains two sets of 4 registers which each hold these start and end points for the 4 lines of the Image mask. The Spanline Mask generator combines these co-ordinates in combination with the Image address pointer (which points to the four 64 pixel lines in Image memory which are about to be updated and its special purpose hardware which of these 256 pixels fall inside the defined spanline shape. All the programmer has to do is to supply the co-ordinates, and the hardware does the rest!



The Image Mask is automatically updated for the next 4x64 pixels if you update Image Address pointer 1: control register VAU_Image1. This means a significant reduction of overhead in the time critical inner loop of the algorithm. An update of the X,Y reference address which points to the X,Y location from where data will be written causes an immediate update of both the Window Mask and the Span Mask.

Defining the Spans: The Span Mask generator has two sets of four registers, one set for each horizontal line. Each line has a 32 bit start point register and a 32 bit end point register, both expect signed 16.16 fixed point values. The start point is the first point which is included in the span and is located at the left side. The end point is the first point which is excluded from the span and is located at the right side. A top and bottom limit can be dined with the MSK_Spanline_Y control register which contains the minimum and maximum values.

Accessing the Spanline registers: The way to write to the Spanline register is to use the MSK_SpanStart and MSK_SpanEnd control registers together with the SPAN_RW_PTR[1:0] index in MSK_Control1[5:4]. The index select any of the the four spanlines and is auto-incremented after a read or a write. The control registers expect a 32 bit X co-ordinate in signed 16.16 format. (The lower 16 bit are fractional). The control registers MSK_SpanLines can read the 16 significant bit of both SpanStart and SpanEnd simultaneously or write the same fields in a single write access. (zeroes file the fractional parts).

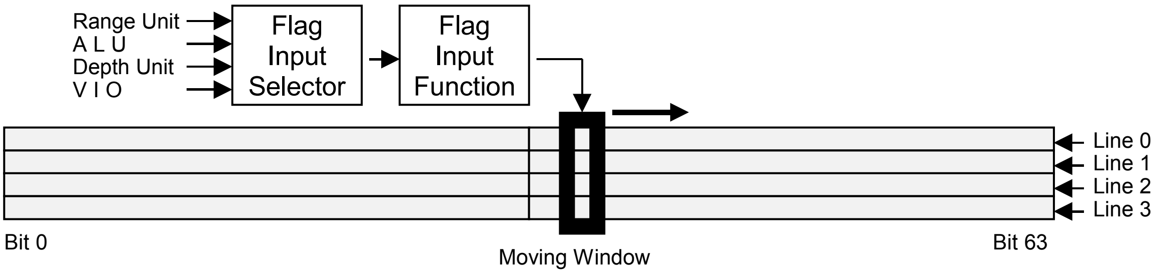
Incrementing the Spanline registers: The four SpanStart and SpanEnd registers can be incremented by the Delta values stored in the MSK_DeltaStart and MSK_DeltaEnd control registers. One or more of the four spanlines can be incremented depending of the contents of the MASK_MOD_MAP[2:0] field in MSK_Control1[22:20]. The operations takes the same number of cycles as the number of spanlines which is incremented.

Obtaining information for Vector processing: Two read only registers provide information for Vector processing: MSK_SpanAddr contains the Start X co-ordinate of a vector which contains one, two or four scan-lines. This value can be moved to VAU_Image1. Control register MSAK_SpanLength contains the value which can be placed in the SEQ_Repeat register and which defines the length of the vector from the start of the left most Spanline start co-ordinate to the end of the right-most Spanline end co-ordinate. A "Bressenham Delta is send to the 3D graphics unit to select between the two Delta value used for the Bressenham interpolation by this unit The MASK_MOD_MAP[2:0] field defines the Spanlines which are used for all these functions.

12.3.3 The Range mask generator

In the cases as described so far, the mask was derived from geometric information. Co-ordinates are translated to individual bits in the mask registers. These masks then determine the shape of a graphics primitive. Another important class of masks is represented by the Range mask generator. Here the contents of the mask is not determined by geometric information but by some properties of the contents of the individual pixel. The most simple case is a character bitmap font. The bits in the bitmap control if the pixel is written or not, or if it is written in either the foreground or the background colour.

The Range mask Generator



The Range Mask generator has the ability to assemble the boolean results flags from vector operations into the Range mask. Multiple results can be Or-ed or AND-ed together. The results from four different units can be selected. The RANGE_SEL[1:0] field in MSK_Control2[9:8] determines which of the four is used.

1) The status flags from the Multiplier/ Accumulator (4x8, 2x16 or 32 bit)
Within the multiplier / accumulator you can select the following options:
Inside, Higher, Lower, Wrong, not Inside, not Higher, not Lower, not Wrong

2) The status flags from the ALU
Within the ALU you can select the following options: (4x8, 2x16 or 32 bit)
Zero, Minus, Carry, Sgncmp, not Zero, not Minus, not Carry, not Sgncmp

3) The results from the 3D graphics Depth buffer compares (2x16 signed/unsigned, 32 signed/unsigned/float)
Within the Depth Compare unit you can select the following 'Open GL' options for the Depth Test:
Never, Less, Equal, Less/Equal, Greater, not Equal, Greater/Equal, Always
(A parallel options is a compare with the front clipping plane)

4) The result from the Alpha test in the VIO (32 bit aRGB values)
Within the VIO unit you can select the following 'Open GL' options for the Alpha Test:
Never, Less, Equal, Less/Equal, Greater, not Equal, Greater/Equal, Always
(Parallel options are a test on Alpha not Zero and a compare of Alpha with a dither value)

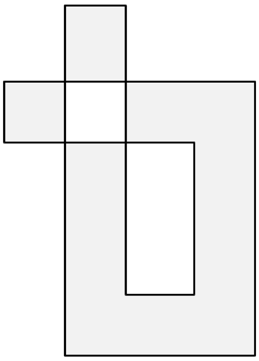
The four bits from the Range Clip unit in the Imagine data processor can be written to a location in the Image Mask register. The second Image mask control register contains a 6 bit counter: RANGE_INPUT_POINTER[5:0] which can be set by writing to the control register or using an image mask instruction to modify the control register. see MSK_Control2[23:18]. The value contained in the counter points to the location where the four comparison result bits will be written. Valid values are in the range from 0 to 63. The pointer should start at 0 according to standard conventions which define the most significant bit in a word as the left-most pixel on the screen. The counter is post-incremented after the bits are inserted. The fields RANGE_OP[1:0], RANGE_INPUT_MAP[2:0] and RIO determine which operations are performed during the construction of the Range mask. These fields can be found in MSK_Control2[17:10]

12.3.4 The Complex mask generator.

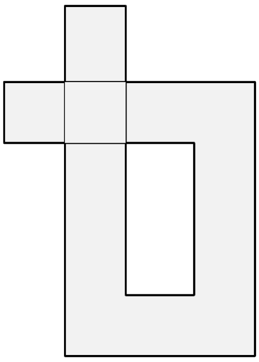
This unit can be used to generate pixel masks for complex, self intersecting polygons These polygons can for instance represent scaleable Latin or Japanese characters, It is more complicated to determine which pixels fall inside and which fall outside the body of the polygon. Two different definitions exist to specify the inside area of a complex polygon: The Odd/Even rule and the Winding rule. The definitions for these two are as follows:

Odd/Even rule: A point belongs to the polygon if an infinitely long half line in any arbitrary direction with its starting point in the tested crosses an odd number of edges.

Winding rule: A point belongs to the polygon if an infinitely long half line in arbitrary direction with its starting point in the tested point crosses an unequal number of left winding and right winding edges.



Odd/Even rule



Winding rule

All graphics standards can select between the two rules demonstrated above.

The Complex Polygon Mask generator resolves the membership question for 64 pixels on a horizontal line. The programmer needs to provide the Generator with the crossing points of the edges and the infinite horizontal line which contains the set of pixels under test. An entire 64 pixel line with 12 crossings can take as few as 6 cycles to resolve.

Both the Odd/Even rule and Winding rule allow an infinite number of crossing points. The Winding rule evaluation has the practical limitation that the sum of left and the sum of right winding edges may not differ more than 15, which will not be the case in any practical situation.

The image mask generator instructions

INSTRUCTION WORD

‘1101'				SET_ CTRL [1:0]		CA MASK [1:0]		RC MASK [1:0]		I R M	I M M	T I M	O I M	I S L	I S P
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48

INSTRUCTION WORD: New control register contents for MSK_Control1

I M E	‘00000000'								MASKMOD MAP [2:0]			MASK SIZE [1:0]		MASK LWPTR [1:0]		MASK ENABLES [3:0] WM	SM	RM	CM	‘00'		CPLX ASM [1:0]		‘00'		SPAN RW-PTR [1:0]		‘00'		MASK RW-PTR [1:0]	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

INSTRUCTION WORD: New control register contents for MSK_Control2

‘00000000'								RANGE INPUT POINTER [5:0]						RANGE OP [1:0]		‘00'		RANGE INPUT MAP [2:0]			R I O	RANGE SEL [1:0]		‘0'	CPLX ALPHA [1:0]		‘0'	SPAN FUNCT [1:0]		LINE SIZE [1:0]	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

An Image Mask Instruction can be a combination of up to 8 instructions separated by commas. The individual instructions correspond with the 8 fields in the Instruction word bit [59:49]. The instructions of the Image Mask Generator define Mask Assembly Instructions for the 2 result masks: Transparent and Opaque masks, Set / Reset instructions for Function Specific Masks, information for the 2 Image Mask control registers, et cetera.

List of assembly mnemonics:

IC[59:58]: **mask_control1(** option list **)** or **mask_control2(** option list **)**
IC[57:56]: **reset_complex_alpha_mask** or **set_complex_alpha_mask** or **invert_complex_alpha_mask**
IC[55:54]: **reset_range_clip_mask** or **set_range_clip_mask** or **invert_range_clip_mask**
IC[53]:mask_modification_map++
IC[52]:range_input_map++
IC[51]:make_transparent_mask
IC[50]:make_opaque_mask
IC[49:48]: **spanlines++**

<u>**SET_CTRL:** Set contents of Control Registers:</u>

Control register information can be written directly into one of the two control registers of the Image Mask generator. The definition of the parameters which can be supplied with these functions is given in the paragraphs which describe the mnemonics of the control registers itself. The control registers are modified in the same cycle in which the instruction is executed.

SET_CTRL[1:0] = IC[59:58]

00: no op (default)
01: **mask_control1** (option list)
10: **mask_control2** (option list)

RC MASK and CA MASK

These fields enable some simple operations to be performed on the Range Clip Mask and the Complex/ Alpha Mask The Range/Clip Mask and the Complex/Alpha mask can be initialised to 0 or 1 or alternatively they can be inverted (all 128 bits at once). This action takes place at the same cycle in which the instruction is issued.

RC_MASK[1:0] = IC[55:54]

00:	no op (default)
001	**reset_range_clip_mask**
10	**set_range_clip_mask**
11	**invert_range_clip_mask**

CA_MASK[1:0] = IC[57:56]

00:	no op (default)
01:	**reset_complex_alpha_mask**
10:	**set_range_clip_mask**
11:	**invert_complex_alpha_mask**

IRM and IMM: Post Increment Fields

The control registers contain two fields which can be incremented with the instruction word: The Range Input Mapping and the Mask modification mapping. The increment takes place one cycle after the instruction is issued. IRM post increments the value of the RangeInMap field (reg #09, bits 11..13) in the next cycle. IMM post increments the value of the Mask Modification Mapping field (reg #08, bits 20..22) in the next cycle.

The post-increment operations do not influence all the bits in the field:

001:	Do not modify any bit
01x:	Modify only bit x
1xx:	Modify only bits xx

IMM = IC[53]

0:	no op (default)
1:	**mask_modification_map++**

IRM = IC[52]

0:	no op (default)
1:	**range_input_map++**

TIM and OIM Assemble the result masks

Both the Final Masks (the Transparent and Opaque mask) can be assembled by the a Mask instruction. They are assembled one cycle after the instruction is issued. This means that a single instruction can set a control register, initialise or invert any of the two input masks and then generate the mask with the newly provided data. TIM assembles the Transparent Image Mask in the next cycle. OIM assembles the Opaque Image Mask in the next cycle.

ISL (post) Increment the Span lines

The four Spanlines each have a Start X co-ordinate and an End X co-ordinate. These values are defined as 32 bit fixed point values with the binary point in the middle (16.16) The increment function controlled with the ISL flag will add a Start Delta value and an End Delta value to all Spanlines which are selected with the MASK Modification MAP (**MSK_Control1[22:20]**). This post-operation does not influence the assembly of the Transparent or the Opaque mask caused by the current instruction

Detailed description of Image Mask control register 1. (cr88)

cr88: MSK_Control1

I M E	'00000000'								MASKMOD MAP [2:0]			MASK SIZE [1:0]		MASK LWPTR [1:0]		MASK ENABLES [3:0] WM	SM	RM	CM	'00'		CPLX ASM [1:0]		'00'		SPAN RW-PTR [1:0]		'00'		MASK RW-PTR [1:0]	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The instruction **mask_control1()** needs a number of parameter, any of the following parameters may be given, separated by commas, when this instruction is used. They parameters are optionally. The default value is used when a parameter is omitted.

List of assembly mnemonics for MSK_Control1:

cr88 [31]:	**little_endian** (default) or **big_endian**
cr88 [22:20]:	**make_lines_0123** or **make_lines_01** or **make_lines_23** or **make_line_0** or **make_line_1** or **make_line_2** or **make_line_3**
cr88 [19:18]:	**masksize_64** (default) or **masksize_32**
cr88[15]:	**window**
cr88[14]:	**spanline**
cr88[13]:	**range** or **clip**
cr88[12]:	**complex** or **alpha**
cr88 [9:8]:	**straight** (default) or **odd_even** or **winding**
cr88 [5:4]:	**spanline(0)** (default) or **spanline(1)** or **spanline(2)** or **spanline(3)**
cr88 [1:0]:	**maskline(0)** (default) or **maskline(1)** or **maskline(2)** or **maskline(3)**
(cr88 [16]==0)	**maskline(0, 31:0)** or **maskline(1, 31:0)** or **maskline(2, 31:0)** or **maskline(3, 31:0)**
(cr88 [16]==1)	**maskline(0, 63:32)** or **maskline(1, 63:32)** or **maskline(2, 63:32)** or **maskline(3, 63:32)**

Some mask register addresses have multiple registers for multiple spanlines. The masks for different lines are accessed with the use of two entry pointers:

Maskline_pointer Mask Read/Write Pointer
This field is used as a selector when any of the four mask register sets. are accessed. (Range mask, Complex mask, Transparent mask, Opaque mask). The field represents a two bit auto increment pointer which is used as a reference for read and write accesses to the Complex registers, The Range registers, the Transparent image mask registers and the Opaque image mask registers. The pointer is post-incremented after an access to any of these registers.

MASK_RW_PTR() = MSK_Control1[1:0]
MASK_LW_PTR() = MSK_Control1[16]

00 0	**maskline(0)**	line 0 bits[31:0]
01 0	**maskline(1)**	line 1 bits[31:0]
10 0	**maskline(2)**	line 2 bits[31:0]
11 0	**maskline(3)**	line 3 bits[31:0]
00 0	**maskline(0, 31:0)**	line 0 bits[31:0]
01 0	**maskline(1, 31:0)**	line 1 bits[31:0]
10 0	**maskline(2, 31:0)**	line 2 bits[31:0]
11 0	**maskline(3, 31:0)**	line 3 bits[31:0]
00 1	**maskline(0, 63:32)**	line 0 bits[63:32]
01 1	**maskline(1, 63:32)**	line 1 bits[63:32]
10 1	**maskline(2, 63:32)**	line 2 bits[63:32]
11 1	**maskline(3, 63:32)**	line 3 bits[63:32]

Spanline_pointer Span line Read/Write Pointer
This field is used as a selector when the four spanline registers are accessed. The field represents a two bit auto increment pointer which is used during read and write actions of the Spanline Coordinate registers. The pointer is post-incremented after a spanline register access or a Polygon End Coordinate Entry access if the *Entry function* (cr41) has defined the Entry points as inputs for the Spanline register.

SPAN_RW_PTR () = MSK_Control1[5:4]

00	**spanline(0)**	access line 0
01	**spanline(1)**	access line 1
10	**spanline(2)**	access line 2
11	**spanline(3)**	access line 3

The Final Image Mask Assembly functions:

The final Image Mask which is represented by the Transparent Image Mask and/or the Opaque Image Mask is assembled from four function specific Mask assembly Units. The Image Mask assembly function defines which of these units are used and how they are used: Four bits, one for each of the four function specific units defines if the particular unit is used or not.

WM:	Window Mask Enable.
SM:	Span Line Mask Enable.
RM:	Range Mask Enable.
CM:	Complex Mask Enable.

'1': Use Mask, '0' do not use Mask.

Mnemonics
A mask is selected by adding its mnemonic to the parameter list of the function **mask_control1()**. The mask is deselected if it is omitted from the parameter list.

WM = MSK_Control1[15]

0:	do not use the Window Mask
1:	**window**

SM = MSK_Control1[14]

0:	do not use the Spanline Mask
1:	**spanline**

RM = MSK_Control1[13]

0:	do not use the Range Mask
1:	**range** or **clip**

CM = MSK_Control1[12]

0:	do not use the Complex Mask
1:	**complex** or **alpha**

Complex Mask registers Usage

This parameter defines the use of the contents of the complex/ alpha mask for the assembly of the Transparent and/or Opaque masks. (The default value is 00) The Complex Mask register contents can undergo an extra processing step before it is used in the construction of the final Image Mask. Four bits on equal bit positions in the four registers are considered as a four bit data word. The function defines how these four bits are mapped to the four bits used for the final mask assembly. The straight function uses the four Complex bits directly. This option should be used if the mask is used for alpha plane calculations or as a simple clip mask which is written straight into the complex alpha mask. The **odd_even** and **winding** functions are typically used during the rendering of complex polygons. The odd_even function sets all four bits to logical '1' if the four bit input value is 'odd', otherwise it resets all four bits to '0'. The winding function operates in a similar way: all four bits are set to '1' if the four bit input value is 'not zero', otherwise it resets all four bits to '0'.

CPLX_ASM[1:0] = MSK_Control1[9:8]

00	**straight**	straight mask usage
01	**odd_even**	complex odd/even rule
10	**winding**	complex winding rule

Mask Modification Mapping

This field determines which register from either the Transparent Image Mask and the Opaque image mask are updated. This field can be post-incremented with a bit in the Image Mask Generation Instruction word. The post-increment operation does not influence all the bits in the field

MASK_MOD_MAP[2:0]
= MSK_Control2 [22:20]

000:	none (default)
001:	**make_lines_0123**
010:	**make_lines_01**
011:	**make_lines_23**
100:	**make_line_0**
101:	**make_line_1**
110:	**make_line_2**
111:	**make_line_3**

Detailed description of Image Mask control register 2. (cr89)

cr89: MSK_Control2

<table>
<tr><td colspan="8">‘00000000'</td><td colspan="6">RANGE INPUT POINTER [5:0]</td><td colspan="2">RANGE OP [1:0]</td><td colspan="2">‘00'</td><td colspan="3">RANGE INPUT MAP [2:0]</td><td>R I O</td><td colspan="2">RANGE SEL [1:0]</td><td>‘0'</td><td colspan="2">CPLX ALPHA [1:0]</td><td>‘0'</td><td colspan="2">SF [1:0]</td><td colspan="2">LS [1:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

The instruction **mask_control2()** needs a number of parameter, any of the following parameters may be given, separated by commas, when this instruction is used. They parameters are optionally. The default value is used when a parameter is omitted.

List of assembly mnemonics for MSK_Control2

cr89 [23:18]:	**range_pointer(0)** (default) or **range_pointer (1)** or ... or **range_pointer (63)**
cr89 [17:16]:	**and_range_flags** or **or_range_flags** or **copy_range_flags**
cr89 [13:10]:	**copy_to_0123** or **and2_to_01** or **and2_to_23** or **or2_to_01** or **or2_to_23** or **and4_to_0** or **and4_to_1** or **and4_to_2** or **and4_to_3** or **or4_to_0** or **or4_to_1** or **or4_to_2** or **or4_to_3**
cr89 [9:8]:	**range_flags** (default) or **alu_flags** or **depth_flags** or **vio_flags**
cr89[6:5]:	**data_inc** (default) or **data_dec** or **mask_incinc** or **mask_incdec**
cr89[3:0]:	do not use for new code (see Imagine 1 manual)

Complex/ Alpha Mask generation Function

This parameter defines what operation is executed in the Complex/Alpha mask when data is written to the Polygon Entries. **MSK_PolyStart** (cr94), **MSK_PolyEnd** (cr95) and **MSK_PolyCoord** (cr96). These entries start functions in the Complex / Alpha Mask generator. The Complex/Alpha mask ALU works on 64 nibbles of 4 bit each. These nibbles can be set and reset with the instruction word and incremented and/or decremented with data input. Two 64 bit masks can be generated via these entries. **MSK_PolyStart** entry will generate the 64 bit Start mask when written to. It expects a spanline start X co-ordinate with a 16.16 fixed point format.

The upper 16 bit are compared with the 16 X co-ordinate bits from **VAU_Image1** (cr116). Bit 0 of the mask corresponds with the Start co-ordinate written to **MSK_PolyStart** (cr94). Bit 1 corresponds to the next pixel (at the right side) et-cetera. The corresponding Mask bits are set to '0' if the are before (left) of the reference value from **VAU_Image1** and are set to '1' if the are behind (right) of the reference value. This array of 64 single bits is added to the array of the 64 nibbles of the Complex / Alpha mask when either **mask_incinc** or **mask_incdec** are selected. A similar second 64 bit mask is generated by writing to **MSK_PolyEnd** (cr95). This 64 bit mask is added to the array of the 64 nibbles of the Complex / Alpha mask when **mask_incinc** is selected and subtracted when **mask_incdec** is selected. A write operation to **MSK_PolyCoord** (cr96) will generate both 64 bit masks. This entry expects the 16 bits Start X co-ordinate in bits [31:16] and the End X co-ordinate in bits [15:0]. An alternative usage is to supply the mask directly to **MSK_PolyCoord** (cr96). The 32 bit input data is now added (option **data_inc**) or subtracted (option **data_dec**) as an array from 32 single bits. It is the MASK_LWPTR[1:0] field **MSK_Control1 [17:16]** which decides which 32 nibbles are modified. A '0' modifies nibbles [31:0] while a '1' modifies nibbles [63:32]

CPLX_ALPHA[1:0] =	MSK_Control2 [6:5]	
00	**data_inc**	increment. with data
01	**data_dec**	decrement with data
10	**mask_incinc**	inc / inc with mask(s)
11	**mask_incdec**	inc / dec with mask(s)

The Range mask flags Select function

RANGE_SEL = MSK_Control2 [9:8]

00	**range_flags**	from multiplier
01	**alu_flags**	from ALU
10	**depth_flags**	from the 3D pipeline
11	**vio_flags**	from the vector IO unit

The Range Mask Input Functions

The control fields of the Range Clip Mask define how incoming status flags from the Range Clip unit are inserted into the Range Mask. The four flags undergo a transformation before being applied to the Range Mask. The mnemonics defines two fields with one mnemonic. (RANGE_INPUT_MAP[2:0] and RIO)

RANGE_INPUT_MAP[2:0]:
The four Range flags selected from the Range Clip Unit, The ALU, The Depth Test unit or the Vector IO unit can be mapped in various ways to the four bits which are stored in the four Range mask registers. The four flags can be and-ed / or-ed to modify one mask register bit in one mask-register, two flags can be and-ed / or-ed to modify one bit in one mask register while the other two flags are also and-ed /or-ed to modify the same bit in another mask register or the four flags are independently applied to bits in all four mask registers. The RANGE_INPUT_MAP field in the control register can be post-incremented with a Mask instruction. This post-increment operation only influences certain bits in the field depending on the MSB bits:

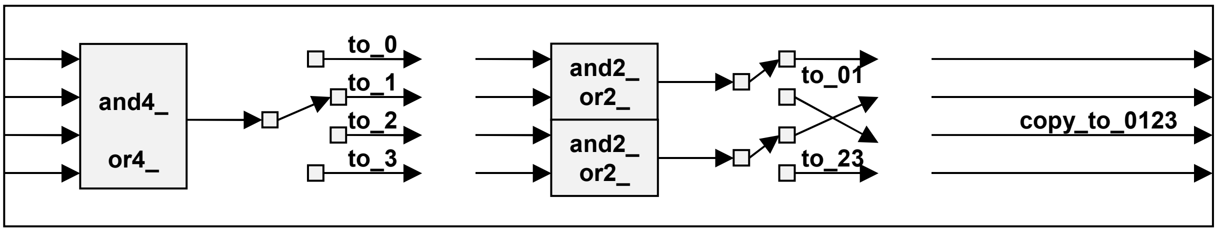
RIO: Range Input Operation:
This is the operation used to combine 2 or 4 flags when used for 1 line. A '0' **AND**s flags and a '1' **OR**s flags.

RANGE_INPUT_MAP[2:0] and RIO
= MSK_Control2 [13:10]

000.0	no-op (default)		
001.x	**copy_to_0123**		
010.0	**and2_to_01**	010.1	**or2_to_01**
011.0	**and2_to_23**	011.1	**or2_to_23**
100.0	**and4_to_0**	100.1	**or4_to_0**
101.0	**and4_to_1**	101.1	**or4_to_1**
110.0	**and4_to_2**	110.1	**or4_to_2**
111.0	**and4_to_3**	111.1	**or4_to_3**

Increment the RANGE_INPUT_MAP[2:0]:

001	Do not Increment
01X	Increment only bit 0
1XX	Increment bits [1:0]



RangeOp: Range Mask Operation:
The newly arriving flags can be combined with the current values in the Range/Clip registers. The status flags can overwrite the old information of the Range Mask (copy) but can also be combined with it (and, or):

RANGE_OP = MSK_Control2 [17:16]

00:	no op	*<default>*
01:	**and_range_flags**	New = Old&Input
10:	**or_range_flags**	New = Old|Input
11:	**copy_range_flags**	New = Input

Range Input Pointer

This parameter defines the bit position in the Range/Clip masks where the status flags from the Range unit, the ALU, the Depth buffer compare unit or the Vector I/O unit are inserted. The default value is 0. The input information will be stored or merged with existing information. This pointer is post-incremented after each time that flags are received by any of the mentioned units.

Mnemonics: **range_pointer (0) ... range_pointer (63)** (MSK_Control2 [23:18])

AN EXAMPLE OF A GENERATED MASK:

Chapter

13. VECTOR ACCESS UNIT

The Vector Access Unit
Can access external memory in vector mode. Simultaneous input and output operations are possible via the internal bi-directional fifo. Quad byte and Double short accesses can be with bytes / shorts after each other or above each other. Access can be non-aligned without speed penalty. Accesses can be in horizontal direction or vertical direction.

Introduction to the Vector Access functions.

Firstly the basic read and write instructions and how to utilise the two masking sources will be described:

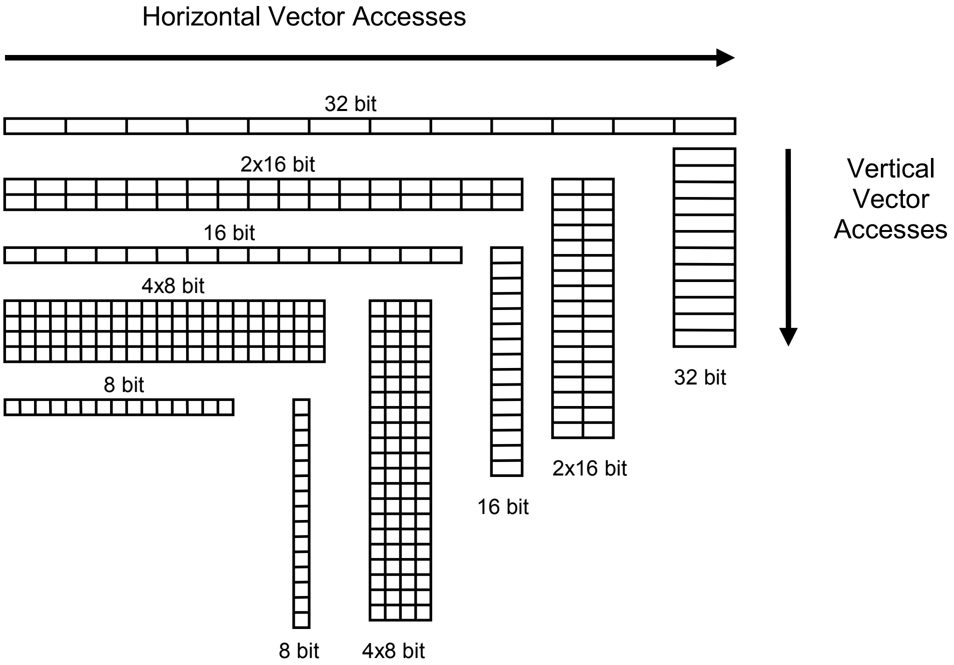
'Quasi' Simultaneous read and write vector operations:

The vector access operations transfer vector data between the SDRAM or SGRAM memory and the Imagine Processor core. The Imagine vector access unit supports peak data rates of 1.600 Gigabyte per second with a 200 MHz clock speed. The Processor can read and write (simultaneously) vectors from 1 to 64 (32-bit) words on a horizontal line at a speed of one read word plus one write word per cycle.

Supported Memory Access Types

Vector accesses to memory can be both horizontal and vertical, The pixel types can be single 32 bit pixels, single or double 16 bit pixels and single or quadruple 8 bit pixels.

The individual pixels in a word are on top of each for horizontal accesses and after each other for vertical accesses with the least significant one at the top (hor) or left-most (ver) location. Accesses for multi pixel words can be non-aligned without speed penalty



So a Vector access can access a single line of 32 bit pixels, it can access two lines of 16 bit pixels or four lines of 8 bit pixels in parallel with the lines on top or after of each other. A special and very important feature is that the 32 bit word can be accessed in a non aligned fashion. The pixels (which are byte addressed) are not bound to 32 bit borders. Any 8 or 16 bit pixel can be accessed as being the top left pixel of the Vector (the addresses used by the Imagine are the Top-Left co-ordinates of the Vector).

Mnemonics of the Vector Access Generator

vector_load (	origin	**origin1** *or* **origin2** *or* **origin3**
	,coordinate	*D***coord1** *or D***coord2** *or D***coord3** *with optional* **++** *where D is either blank,* **1D***,* **2D** *or* **3D**
	[,direction]	**horizontal** *(default) or* **vertical**
	[,pixel_type]	**8** *or* **16** *or* **32** *or* **4x8** *or* **2x16** *or* **1x32**
	[,fixed_length]	**length(** *X* **)** *where X is 1...128*
	[,byte_enables]	**bytes_***enables*
	[,state_select]	**select_state***X or* **clear_state***X where X is 0...3*
	[,continue] **);**	**continue**

vector_store (	origin	**origin1** *or* **origin2** *or* **origin3**
	,coordinate	*D***coord1** *or D***coord2** *or D***coord3** *with optional* **++** *where D is either blank,* **1D***,* **2D** *or* **3D**
	[,direction]	**horizontal** *(default) or* **vertical**
	[,pixel_type]	**8** *or* **16** *or* **32** *or* **4x8** *or* **2x16** *or* **1x32**
	[,fixed_length]	**length(** *X* **)** *where X is 1...128*
	[,pixel_mask]	**pixelmask** *or* **new_pixelmask**
	[,data_source]	**bicolor***, or* **new_bicolor**
	[,byte_enables]	**bytes_***enables*
	[,mask_pointer]	**mask_pointer(***X***)** or **mask_reference(***X***)** *with optional* **++** *and X is 0...63 or blank*
	[,state_select]	**select_state***X or* **clear_state***X where X is 0...1*
	[,line_mapping]	**);line_***map with optional* **++**

vector_control (	*[reset flag]*	**reset_on** *or* **reset_off**
	[coordinate_dim]	**1Dcoord** *or* **2Dcoord** *or* **3Dcoord**
	[,pixel_type]	**8** *or* **16** *or* **32** *or* **4x8** *or* **2x16** *or* **1x32**
	[,byte_enables]	**bytes_***enables*
	[,mask_pointer]	**mask_pointer(** *X* **)** or **mask_reference(***X***)** *with optional* **++** *and X is 0...63 or blank*
	[,state_select]	**select_state***X or* **clear_state***X where X is* **0...3**
	[,line_mapping]	**);line_***map*

enables:	**0** *or* **1** *or* **2** *or* **3** *or* **01** *or* **02** *or* **03** *or* **12** *or* **13** *or* **23** *or* **012** *or* **013** *or* **023** *or* **123** *or* **0123**
map :	**none** *or* **8** *or* **16_01** *or* **16_23** *or* **32_0** *or* **32_1** *or* **32_2** *or* **32_3** *or* **map**

Instruction word definition of the Vector Access Unit

INSTRUCTION WORD

‘1110'				AC	RW	HV	ORIG		COORD		IAD	TIM	OIM	TIE	OIE	CD	PT	BE	MP	ST	OM	IMP	IOM	FV	FIXED LENGTH [6:0]						
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32

INSTRUCTION WORD

Vau RST	‘0'	COORD DIM [1:0]		‘00'		PIX TYPE [1:0]		‘0000'				BYTE_ENABLES [3:0]				‘00'		MASK_POINTER [5:0]						MR	SCL	STATE [1:0]		‘0'	MOM [2:0]		
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AC	vector access	1 = vector access, 0 = update control register only
RW	read or write	0 = read, 1 = write, (1 = update VAU Reset Flag)
HV	horizontal or vertical	0 = horizontal, 1 = vertical
ORIG	select address origin	0 = origin1, 1 = origin2, 2 = origin3
COORD	select coordinate reg.	0 = coordinate1, 1 = coordinate2, 2 = coordinate3
IAD	increment Address.	1 = (post) Increment selected coordinate in control register
TIM	make Transparent mask	1 = Enable Transparent mask for write enables
OIM	make Opaque mask	1 = Enable Opaque mask for Bi Color expansion
TIE	use Transparent mask	1 = Enable Transparent mask for write enables
OIE	use Opaque mask	1 = Enable Opaque mask for Bi Color expansion
CD	coordinate dimension	1 = Load coordinate dimension in control register
PT	pixel type	1 = Load pixel type in control register
BE	byte enables	1 = Load byte enables in control register
MP	mask pointer	1 = Load mask pointer enables in control register
ST	state select	1 = Load state select/clear in control register
OM	mask output mapping	1 = Load mask output mapping in control register
IMP	increment map pointer	1 = (post) Increment mask pointer in control register
IOM	increment output map.	1 = (post) Increment mask output mapping in control register
FV	fixed/variable length	0 = fixed length, 1 = variable length
FIXED_LENGTH	fixed length	0..127 represents a length of 1 to 128
VAU_RST	VAU reset flag contents	0 = Reset OFF, 1 = Reset ON
COORD_DIM	coordinate dimension	0 = 1D, 1 = 2D, 2 = 3D, 3 = old XY coordinate
PIXEL_TYPE	pixel type	0 = bytes, 1 = shorts, 2 = words,
BYTE_ENABLES	byte enables	0 = mask, 1 = write, bit[3] → [31:24],..,.., bit[0] → [7:0]
MASK_POINTER	mask pointer	value range is 0 ..63 for the Imagine 2
MR	mask reference	The reference X coordinate points to the mask pointer position.
SCL	state clear	clear / initialise the selected history state
STATE	state select	selected history state (0..3 for reading, 0..1 for writing)
MOM	mask output mapping	0=no_op, 1=line_8, 2=line16_01, 3=line16_23, 4=line_32_0, 5=line_32_1, 6=line_32_2, 7=line_32_3,

The Transparent Pixel Mask

Individual 8 or 16 bit pixels can be masked during a vector write with a the use of the Transparent Image Mask of the Mask generator. Individual components of 32 bit true color pixels can also be masked. The chapter on the Image Mask generator explains the assembly of this mask. The Transparent Mask is contained in a 4x64 bit register set. The programmer can invoke the usage of the Transparent mask with the keyword **pixelmask** for the corresponding parameter. If the mask needs to be assembled before you start the vector write, use the keyword **new_pixelmask**; if you do not specify anything the pixels will not be masked.

Example: **image_vector** (*write, quad_byte, surface2, coord1, pixelmask*);

This instruction causes a vector write of 4x8 bit pixel words (quad_byte) starting at the pixel at the XY location defined in coordinate register 1 on the 2D surface pointed to by the Surface register 2 and masks pixels by applying the Transparent Pixel mask.

The Opaque Pixel Mask

The Opaque Pixel Mask ♦ The **Bicolor** option uses the four mask bits to select between the foreground and background colours contained in the registers with these names. The bits from Opaque Mask register 0 select between the bytes 0 of the foreground and background registers (bit 0..7) which are then placed on bit 0..7 of the external Image databus. These eight bits correspond to the highest of the four lines in case of 8 bit colours when four lines are written in parallel. The four mask bits define in a one-to-one fashion.

♦ **bicolor, new_bicolor**

The bicolor option supports very high speed colour expansion;

Example: **image_access** (*write, quad_byte, image2, bicolor*);

The vector access control register

Cr112: VAU_Control, Vector Access Unit Control register

<table>
<tr><td>Vau
RST</td><td>‘0'</td><td colspan="2">COORD
DIM
[1:0]</td><td colspan="2">‘00'</td><td colspan="2">PIX
TYPE
[1:0]</td><td colspan="4">‘0000'</td><td colspan="4">BYTE_ENABLES
[3:0]</td><td colspan="2">‘00'</td><td colspan="6">MASK_POINTER
[5:0]</td><td>‘0'</td><td>SCL</td><td colspan="2">STATE
[1:0]</td><td>‘0'</td><td colspan="3">MOM
[2:0]</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

The fields of the Image Memory Access control register.

Mask Output Pointer

The Mask information stored in the Transparent and Opaque Mask is accessed during a vector write with use of the Mask output pointer. This pointer walks through the mask during the vector write. It is decremented for each new horizontal address (the left-most pixel has the highest bit address, compatible with industry standards). A typical 32xn vector starts with the mask at bit location 31 and ends with bit location 0.

The Mask Output mapping

Both Transparent and Opaque masks are line oriented masks. Each of the four registers in both masks has information for another line which may be 8, 16 or 32 bit/pixel. The output bus is 32 bit wide for the first version of the Imagine. This means that four 8 bit, or two 16 bit pixel lines can be drawn in parallel. The mapping of the internal mask lines to the 32 bit databus is handled with the Mask Output mapping field.

Mask output mapping

		MSK0	MSK1	MSK2	MSK3
		Byte0	Byte1	Byte2	Byte3
000	<default>	'1'	'1'	'1'	'1'
001	**line_8**	line 0	line 1	line 2	line 3
010	**line_16_01**	line 0	line 0	line 1	line 1
011	**line_16_23**	line 2	line 2	line 3	line 3
100	**line_32_0**	line 0	line 0	line 0	line 0
101	**line_32_1**	line 1	line 1	line 1	line 1
110	**line_32_2**	line 2	line 2	line 2	line 2
111	**line_32_3**	line 3	line 3	line 3	line 3

VECTOR ACCESS UNIT CONTROL REGISTERS

cr112: VAU_control Vector Access unit control register

Vau RST	‘0'	COORD DIM [1:0]		‘00'		PIX TYPE [1:0]		‘0000'				BYTE_ENABLES [3:0]				‘00'		MASK_POINTER [5:0]						MR	SCL	STATE [1:0]		‘0'	MOM [2:0]		
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr113: VAU_status Vector Access unit status register

Int Dis	Vau Out wait	Vau Inp wait	Vio Inp wait	Bi Col	‘000'			‘00'		WRITE FIFO LEVEL [5:0] 128 BIT WORDS						‘00'		READ FIFO LEVEL [4:0] 128 BIT WORDS						‘00'		TASK BUFFER LEVEL [4:0]					
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr114: VAU_Foreground Foreground Color register

Foreground Color value [31:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr115: VAU_Background Background Color register

Background Color value [31:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr116: VAU_Coord1 1,2 or 3 dimensional Coordinate 1 register

X coordinate [31:0]																															
Y coordinate [15:0]																**X** coordinate [15:0]															
								Z coordinate [15:0]								**Y** coordinate [15:0]								**X** coordinate [15:0]							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr117: VAU_Coord2 1,2 or 3 dimensional Coordinate 2 register

X coordinate [31:0]																															
Y coordinate [15:0]																**X** coordinate [15:0]															
								Z coordinate [15:0]								**Y** coordinate [15:0]								**X** coordinate [15:0]							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr118: VAU_Coord3 1,2 or 3 dimensional Coordinate 3 register

X coordinate [31:0]																															
Y coordinate [15:0]																**X** coordinate [15:0]															
								Z coordinate [15:0]								**Y** coordinate [15:0]								**X** coordinate [15:0]							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr120: VAU_Surface1 Surface 1 offset pointer

Linear offset address for 2D surface number1 [31:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr121: VAU_Surface2 Surface 2 offset pointer

Linear offset address for 2D surface number2 [31:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr122: VAU_Surface3 Surface 3 offset pointer

Linear offset address for 2D surface number3 [31:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

'**image_access**' executes a single access to memory opposed to an '**image_vector**' which accesses from 1 to 64 consecutive image memory locations in the X direction. The image access is a '*write*' access of a 32 bit '*word*' and the address is taken from '*image2*': the image address pointer 2. '*bitplane*' is used to present the bitplane mask register to the Image databus during the leading RAS edge.

The Foreground & Background colour registers

The Foreground and Background colours are standard features of all window systems and many graphics standards. They are basically used in combination with character fonts where binary information is expanded into for- and background colours. The Foreground and Back-ground colour registers are intended as 'semi-permanent' locations for these colours which can stay there until software needs other ones.

The contents of the 32 bit register can be placed on the databus during the leading RAS and CAS edges. The colour registers are 32 bit wide. In 8 or 16 bit operations the 8 or 16 bit colours should be repeated over the entire 32 bit width of the registers. The colours can be used in these modes to handle 2 or 4 pixels in parallel.

All these modes work in both scalar and vector mode.

1♦ The Opaque Mask Expansion is invoked by using the **bicolor** option for the data during the CAS edges. The 32x4 opaque mask is used for a maximum of 32 writes in vector mode.

The binary information in the mask will be expanded to the foreground and background colours which are then placed on the CAS edges during writing.

Example of an opaque mask expansion vector write operation:

image_vector (*write,quad_byte,image1,bit_plane,bicolor*)

2♦ In the Transparent Mask mode only one of the two colours is used and should be specified with either **foreground** or **background** as the data source during the CAS edges. The binary information in the 32x4 Transparent Mask is then used to write/not write this colour into Image memory. Example of Transparent Mask operations which use a colour register:

image_vector (*write,word,image1,foreground,pixel_mask*);

The foreground colour is written to the non masked pixels.

The Image Memory Address Registers.

These registers contain 2 dimensional Image memory addresses (X,Y) which can be selected by the Image Access function. They can be post-incremented when they are referred by the image access function.
These X,Y addresses are byte oriented and represented by two sixteen bit integer numbers in the range from -32768 to +32767. The X address is supplied in the highest 16 bit (16..31) while the Y address is supplied in the lowest 16 bit (0..15).

The Image Address Pointers

Three registers can be used during graphics operations. A typical usage is one destination address pointer (IA1) and two source address pointers (IA2,IA3). Image Address Pointer 1 is per definition the one which is referenced by the Image Mask generator to generate a new pixel mask. All three registers can be post-incremented in horizontal direction in a way that allows operations on arbitrary length vectors. The length (-1) of such a vector is stored in the repeat register and can range from 1 to 32768 (the values stored range from 0 to 32767). The maximum vector we can read or write contains 32 words; the actual maximum that will be used can be smaller and is defined by the maximum repeat register which can define any value from 1 to 32.

The length of both vectors which is written in the example and the number of times the vector instruction is repeated is defined by either the repeat register or the maximum repeat register if the contents of the repeat register is larger than the maximum.

Say we want to write a vector with a scan length of 117 pixels and the maximum vector length has the default value 32. This vector is then split up into 3 vectors of 32 and a tail vector of 21 during four loops by the following sample program which demonstrates a basic programming mode of the Imagine:

```
lab:  repeat after (4);
    image_vector (write,word,image1++,data,pixel_mask);
    ....................;
    ....................;
    <repeated vector instruction>, V = output;
    ....................;
    branch (lab), ifnot (repeat_smaller);
    ....................;
    ....................;
```

The **repeat** instruction refers to the repeat register (var) for its repeat count. It will get three times the value 32 and the last value will be 21. It will wait every time for 4 cycles until the target function has arrived which will be repeated by the specified number of times.

During the execution of the **image_vector** function quad_byte data will be written to the address specified by Image address pointer 1. This function will write 3 times a 32 word vector and a final 21 word vector. The address register is referred to by Image++ which causes an post-increment by 32 in the first 3 loop passes and 21 during the last loop pass.

Note that the <repeated vector instruction contains an output instruction which places calculated data (from the Imagine's data processing units) on the Image databus while the image_vector instruction has the parameter *data* which means the data should come from this source indeed.

The output instruction is repeated 3 x 32 times and 1 x 21 times. The data can for instance be read from the internal register file or the data memory with an auto increment mode.

The **branch** causes the loop to be executed four times. The status flag being tested is found in the control/status register and is the result of a comparison of the repeat register and the maximum repeat count register. If the first one is larger then we are not ready yet.
If it is smaller or equal then we need one or more extra passes. The repeat register is post-decremented with the maximum repeat count each time an arbitrary sequencer instruction refers to the flag mentioned above.

TRANSLATION FROM MNEMONICS INTO INSTRUCTION WORD

Mnemonics of the Image Memory Access Generator:

image_function (*function, pixel_type, address_source [,data_source][,plane_mask][,pixel_mask]*)

Image Function (Ic59..58)

Fn	image_function	description
00	**image_access**	Single image memory access
01	**image_vector**	Vector image memory access (From 1 to 32 reads or writes in burst mode)
11	**image_control**	Set control register cr52 (copy bits 0..23 from the instruction word.

The Parameters for the Image Access Functions

function (Ic17..13) and (Ic10..0)

A large number of functions can be selected from the two large tables on the following pages. These functions include all existing functions for Dual and Triple ported DRAM and will extended (in software) as new functions become available. The function determines the values which are send to the control inputs of the various memory chips.

Pixel Type/Size (Ic57..55, Ic54)

This parameter defines the size and type of the pixel which is used during the image memory access. The types byte and short modify only 8 or 16 bit in image memory during write operations while all others write 32 bit
The types double_short and single_word causes
automatic alignment on 16 and 32 bit Y addresses (bit 0 and/or bit 1 of the Address is cleared during the Address selection which chooses between 5 control registers for the access address. The size determines also the Y-increment value for the two Display registers.

The Output mapping (cr52[2:0] can be incremented by adding ++ to the pixel type (no space in between). This causes bit IC[54] of the instruction word to be set.

T/Sz	pixel_type		size
0 00	**quad_byte**	(++)	4x8 bit
0 01	**double_short**	(++)	2x16 bit
0 10	**single_word**	(++)	1x32 bit
1 00	**byte**	(++)	8 bit
1 01	**short**	(++)	16 bit
1 10	**word**	(++)	32 bit

Coordinate
source (IC[5554]

This parameter selects between five different control registers. Three of the are Image addresses which are used typically for graphics operations. E.g.: They can point to two source areas and one destination area. The other two registers contain display addresses which are typically used during the line refresh interrupt to supply the next line address.
Both types of registers can be auto incremented by attaching ++ to the mnemonic. An auto-increment for an Image Address results in an X-address increment with 1 for single accesses and N = 1..32 for vector access, where N is the length of the Vector. An autoincrement of the Display pointers results in an Y-address increment with 1,2,4 or 8 depending on the size of the pixel (Ic55..56): byte, short, word or double.

IC[22,21,53]		selected address source:
00 0	**image1**	Image Addr. Pointer 1.
00 1	**image1++**	
01 0	**image2**	Image Addr. Pointer 2.
01 1	**image2++**	
10 0	**image3**	Image Addr. Pointer 3.
10 1	**image3++**	

data source (Ic20..18)

This parameter defines the origin of the data send to Image memory during write operations
♦ The default option 'data' uses data from the Image I/O unit which can select Data from any data processing unit of the Imagine.
♦ The color registers can be used to download a color externally.
♦ The bicolor mode uses 4 bit at a time from the opaque mask register to decide between either the foreground or the back-ground color for each of the four bytes of the Image Databus

CD2..0	data_source	selected CAS data source:
00 0	**data**	Image I/O unit (default)
11 0	**bicolor**	Selected for/background
11 1	**new_bicolor**	Selected for/background

Transparent Mask assembly usage (IC(50..51)

This parameter selects the transparent mask to be used when writing to image memory. Four bits at a time can be send to image memory via the four external MSK output pins. the option new_pixel_mask generates a new transparent mask just before the actual write actions to image memory start.

E/A	pixel_mask	Transp. Mask operation
00	<default>	do not use the mask.
10	**pixel_mask**	use the Transp. mask.
11	**new_pixel_mask**	generate & use mask.

Chapter

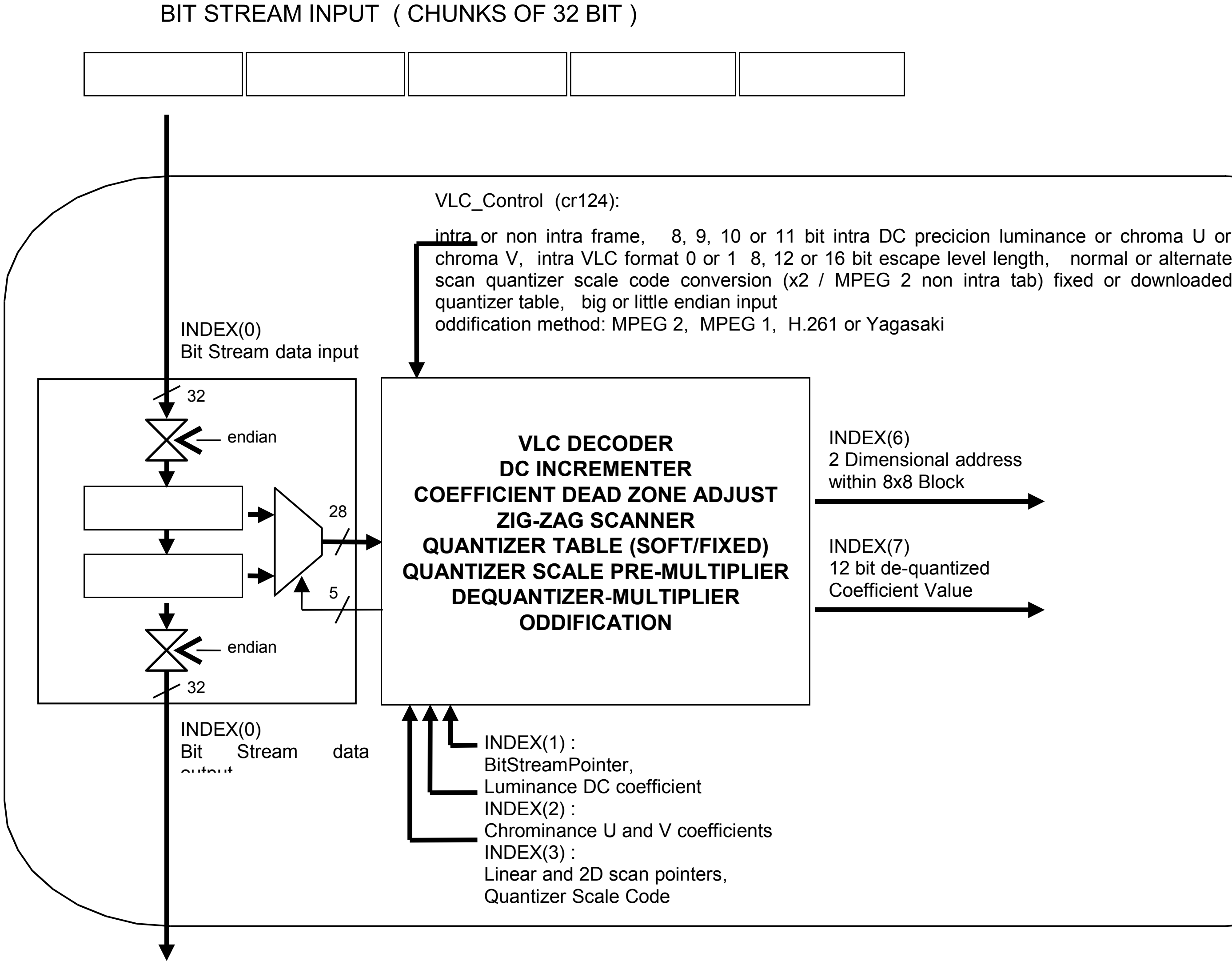
VLC DECODER / DEQUANTIZER

The block level VLC decoder/dequantizer supports MPEG 2, MPEG 1, and H.261 encoded video bit-streams. The input to the unit are 32 bit chunks of serial data in either big or little endian format. The VLC decoder takes five cycles to generate the 12 bit decoded and dequantized coefficient and the 2D sub-address within the 8x8 block after scan conversion. The sub-address is added to a 2D address to obtain the destination address in external memory. Only the non-zero coefficients are generated and written. The external memory should be cleared previously with for instance the SGRAM block fill mode (12..16 Gigabyte/second) to account for the zero coefficients.

A separate register contains the value for coefficient [7][7] which may not have been encoded but has been made non-zero by either the MPEG 2 or Yagasaki oddification method used to correct the IDCT rounding mismatch. This value may always be written to memory after the EOB occurs without the need for any test.

The values in memory are completely processed and the next step in the decoding process is the Inverse DCT.

BLOCK LEVEL VLC DECODER :


BIT STREAM INPUT (CHUNKS OF 32 BIT)
VLC_Control (cr124):
intra or non intra frame, 8, 9, 10 or 11 bit intra DC precicion luminance or chroma U or chroma V, intra VLC format 0 or 1 8, 12 or 16 bit escape level length, normal or alternate scan quantizer scale code conversion (x2 / MPEG 2 non intra tab) fixed or downloaded quantizer table, big or little endian input
oddification method: MPEG 2, MPEG 1, H.261 or Yagasaki
INDEX(0)
Bit Stream data input
32
endian
28
5
endian
32
INDEX(0)
Bit Stream data output
VLC DECODER
DC INCREMENTER
COEFFICIENT DEAD ZONE ADJUST
ZIG-ZAG SCANNER
QUANTIZER TABLE (SOFT/FIXED)
QUANTIZER SCALE PRE-MULTIPLIER
DEQUANTIZER-MULTIPLIER
ODDIFICATION
INDEX(6)
2 Dimensional address within 8x8 Block
INDEX(7)
12 bit de-quantized Coefficient Value
INDEX(1) :
BitStreamPointer,
Luminance DC coefficient
INDEX(2) :
Chrominance U and V coefficients
INDEX(3) :
Linear and 2D scan pointers,
Quantizer Scale Code

EXAMPLE BLOCK LEVEL VLC DECODER SUB-ROUTINE:

```
vlc_block_level_subroutine:
B = rd(VLC_Control)   =>    F = copy(B);       // prepare for "Data Request" test
;
// The block inner loop is executed once for each non-zero coefficient in the block
// The VLC_decode has no effect if new data is needed, we may therefor perform the
// test after the VLC_decode instruction is given.

vlc_block_loop:                                 // start of the block inner-loop
VLC_decode( ReadStart(6),   WriteIndex(0));     // VLC decode instruction
if (minus), call (load_new_stream_data);        // call to load new bit stream chunk
B  = rd2x16(VLC_Data);                          // read sub address for 8x8 block
A  = rd2x16(BlockAddr),     F = copy(B);        // sub address to ALU for zero test
if (zero), return;                              // return if End Of Block detected
B  = rd(    VLC_Data),      F = add(A,B);       // read coefficient, calculate address
DA = wrAd(F),               D = short(B);       // write calculated coefficient
jump (vlc_block_loop);                          // jump to start of the loop
B  = rd( VLC_Control);                          // branch delay:
                            F = copy(B);        // prepare for "Data Request" test
```

INSTRUCTION WORD

‘1010'				‘1100'				Dec	BT	L/C [1:0]		RD inc	READ INDEX [4:0]					WR inc	WRITE INDEX [4:0]				
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40

VLC_decode (Luminance, ReadStart(4), WriteIndex(0));

VLC_control (Chrominance_U, ReadStart(3), WriteIndex(0));

cr124: VLC_control Variable Length Code unit Control register

REQ = 1 Request for new Serial Data Word (read only). All evaluations are disabled as long as this bit is '1'. The bit is cleared by writing a new 32 bit word to the Index 0 reg.

Intra / Non Intra

INT = 0 non Intra Frame
INT = 1 Intra Frame table

Quantiser Scale Table Select

QS = 0 Quantiser Scale Code x2
QS = 1 MPEG 2 non intra table

EOB = 1 End Of Block detect (read only)
All results and scan pointers (linear and 2D) are reset to zero when an EOB is detected. A 2D scan address = '0' test can be used as an alternative End Of Block detect.

Luminance/Chrominance

L/C = 1x Luminance
L/C = 00 Chrominance U
L/C = 01 Chrominance V

ESC Escape Level Length

0 = 8,16 bit: MPEG 1, H.261
1 = 12 bit: MPEG 2

Req	Eob	RD inc	READ INDEX [4:0]					‘00'		WR inc	WRITE INDEX [4:0]					L/C [1:0]		‘0'	INT	IDP [1:0]		‘0'	VlC	‘0'	Esc	ODD [1:0]		AS	QS	QT	EM
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Read and Write Indices

0 = Bit Stream I/O register
1 = Bit Stream pointer + DC Luminance
2 = DC Chrominance coefficients
3 = Scan & 2D pointer, Quant scale code
4 = Decoded coefficient register
5 = Quantizer value register
6 = 2 Dim address output register
7 = De-quantized Coefficient output register
8 = De-quantized Coefficient [7][7] output reg.
16:31 Downloadable Quantizer Table
Bit 29 / 21 = 1: Post Read / Write Index Incr.

Intra DC Precision

IDP = 0 8 bit DC coef
IDP = 1 9 bit DC coef
IDP = 2 10 bit DC coef
IDP = 3 11 bit DC coef

Intra VLC format

VLC = 0 use table 0
VLC = 1 use table 1

Oddification Method

0 = MPEG 1 **1** = MPEG 2
2 = H.261 **3** = Yagasaki

AS = 0 standard ZigZag
AS = 1 alternate ZigZag

QV = 1 Use Quant Table

EM = 0 big, **1** = little endian

cr125: VLC_Data Variable Length Code unit Data

INDEX 0

INDEX 0 : BIT STREAM I/O [31:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

INDEX 1

'0000'				LUMINANCE INTRA DC COEFFICIENT [11:0]												'0000 0000 00'										BIT-STREAM POINTER [5:0]					
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

INDEX 2

'0000'				CHROMINANCE INTRA DC COEFFICIENT FOR U [11:0]												'0000'				CHROMINANCE INTRA DC COEFFICIENT FOR V [11:0]											
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

INDEX 3

'0000 0000 000'											QUANTIZER SCALE CODE [4:0]					'00'		LINEAR SCAN POINTER [5:0]						'00'		2D SCAN ADDRESS [5:0]					
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr125: **VLC_Data** Variable Length Code unit Data

INDEX 4

EXTENDED SIGN BIT																			DECODED COEFFICIENT OUTPUT REGISTER [12:0] (2 X LEVEL [+ SIGN])												
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

INDEX5

'0000 0000 0000 0000 0'																	QUANTISER SCALE: [14:0] QSCALE_TYPE [QSCALE_CODE] X QUANT_TABLE [N][M]														
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

INDEX 6

'0000 0000 0000 0'													Y Address [2:0]			'0000 0000 0000 0'													X Address [2:0]		
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

INDEX 7

'0000 0000 0000 0000 0000'																				FINAL DE-QUANTIZED COEFFICIENT [11:0]											
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

INDEX 8

'0000 0000 0000 0000 0000'																				FINAL DE-QUANTIZED COEFFICIENT FOR POSITION 77 [11:0]											
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

INDICES 16 through 31

Down loadable Quantizer Table Value Row N, Column 3 or 7, bits [7:0]								Down loadable Quantizer Table Value Row N, Column 2 or 6, bits [7:0]								Down loadable Quantizer Table Value Row N, Column 1 or 5, bits [7:0]								Down loadable Quantizer Table Value Row N, Column 0 or 4, bits [7:0]							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Chapter

15. MOTION ESTIMATOR.

The Motion Estimator performs 200 operations per cycle sum of difference operations needed for MPEG1 and MPEG 2 encoding. It supports arbitrary MxN kernel sizes up to 256x256 and arbitrary search space sizes up to 4096 by 4069

Overview

The motion estimator pipeline is designed to find the motion vector which is used for various type of motion picture compression algorithms including MPEG-1, MPEG-2, and H.261. It is flexible and programmable enough to support all possible motion estimation parameters. The search area can range from 1x1 to 4,096x4,096 pixels, and the search kernel can range from 1x1 to 128x128 pixels without any restriction like the alignment of some integer. The minimum value detection hardware is also included in the pipeline so that the pipeline can continue to work at the highest efficiency.
Description

The motion estimator pipeline is the autonomous unit that does not always require clock-by-clock instruction feeding. The unit starts processing when the motion estimator instruction is given. The motion estimator instruction is the set of instructions designed only for motion estimator pipeline. No other instruction can be executed in the same clock as the motion estimator instruction is specified.

Fig.2 Block diagram of motion estimator pipeline
Fig.2 shows the block diagram of the motion estimator pipeline. It consists of four parts, the Kernel register array (notated as KRA), the Search register array (SRA), the Calculation unit (CU), and the Minimum detection unit (MDU).
The KRA holds the pixel data of the motion estimation kernel, and the SRA holds the pixel data of the motion estimation search area (Fig.2). Each array consists of 16 registers of 32bit width. Every 32bit register represents the 4 pixels of 8bit data, so two register array hold 64 pixels per each. The CU calculates the sum of the absolute differences of the corresponding 64 pixel-pairs. The summation of this unit is affected by the Columns Mask and the Row Mask described later. The CU can also add the optional subtotal input data. The MDU is optionally used, and records the minimum value of the CU output and its corresponding XY position.
The motion estimator instruction initiates the particular type of processing, and also specifies the number of clocks of the processing. During the processing, the motion estimator pipeline read the data from V-bus and A-bus at each clock cycle. V-bus input is used to fill the search register array. This means the V-bus value is written in the leftmost register of the array, and other contents are shifted to the right. Depending on the motion estimator instruction that started the processing, the motion estimator pipeline executes in the following way.
When the motion estimator instruction is motion_estimator_load, A-bus input is used to fill the KRA in the same way as the SRA. In this case, the CU and the MDU do not work nor change. The cr81 doesn't change neither.
When the motion estimator instruction is motion_estimator_calc, A-bus input is used as the subtotal input value. The subtotal input value is optionally used to be added to the output of the CU, depending on the add_subtotal flag in the cr80. The resulting value is written to cr81.
When minimum_test flag in cr80 is set, the MDU is enabled. The MDU compares the cr81 with the minimum value register in the cr82. If the minimum value register is larger, the minimum value register is updated and the current X and Y positions held in the MDU are written to the minimum X and Y position registers in the cr83. The current X position is incremented when the MDU is enabled.
The Column mask and the Row mask specify the valid pixel pairs that should be joined to the summation of the CU. These masks enhance the flexibility of the motion estimator pipeline in the two meanings. First, they enable the motion estimator pipeline to support arbitrary size of the motion estimation kernel by hiding the unneeded pixel pairs. Second, they enable the pipeline to support not only 8bit/pixel data but also 16bit/pixel and 32bit/pixel data by incorporating the pragmatic approximation. The examples of the usage are shown in the Section 5.
When the motion estimator instruction is given, the new_search flag and the new_row flag affect the current XY position held in the MDU in the following way. If the new_search is set, both current X and current Y are set to zero. If the new_row is set, current X is set to zero and current Y is incremented.
Control registers
Flags and the values obtained from calculation are kept in the four 32-bit control registers, cr80, cr81, cr82, and cr83. The location of the flags and values are shown in Fig.3. The result of the sum of absolute differences is stored in the lower 24 bits of cr81, and the minimum value of the sum of absolute differences is stored in the lower 24 bits of cr82. When the resulting value of the sum of absolute differences is smaller than the one in cr82, the current values of X and Y positions are written to cr83 as indicated below. The Column mask, the Row mask, and the flags, new_search, new_row, minimum_test, and add_subtotal, are located in cr80.

Fig. 3 Control registers, cr80, cr81, cr82, and cr83

Pipeline description

The process in the motion estimator pipeline has six stages:

Stage 1: Motion estimator instruction is given (only once for each processing).
Stage 2: Motion estimator reads V-bus and A-bus value, shift the KRA and the SRA.
Stage 3: CU reads the KRA and the SRA, calculates the difference for 64 pixel pairs, reduces 64 values to 32 values using Wallace tree.
Stage 4: CU reduces 32 values to 2 values.
Stage 5: CU adds 2 values and optional subtotal input, and writes the result to the cr81.
Stage 6: MDU updates the cr82 and cr83, according to the result of the previous stage. The current X is incremented.

Examples

8bit/pixel, kernel:16x4, search area:47x4(32x1position)

```
/* Set new_search, minimum_test.  Reset new_row, add_subtotal */
/* Set the Column Mask and the Row Mask to all 1's */
A = rd(ri), V = input;
motion_estimator_load(16);
A = rd(ri), V = input;
/* repeat 14 times */

V = input;
motion_estimator_calc(32);
V = input;
/* repeat 30 times */
;;;;;

B = rd(cr82); /* Read the minimum value */
B = rd(cr83); /* Read the XY position */
```

8bit/pixel, kernel:16x8, search area:47x8(32x1position)

```
/* Set new_search.  Reset minimum_test, new_row, add_subtotal */
/* Set the Column Mask and the Row Mask to all 1's */
A = rd(ri), V = input;
motion_estimator_load(16);
A = rd(ri), V = input;
/* repeat 14 times */

V = input;
motion_estimator_calc(32);
V = input;
V = input;
V = input;
V = input;
V = input, B = rd(cr81);
V = input, B = rd(cr81), wr(ri, B);
/* repeat 31 times */

/* Set minimum_test, add_subtotal.  Reset new_search, new_row. */

A = rd(ri), V = input;
motion_estimator_load(16);
A = rd(ri), V = input;
/* repeat 14 times */

A = rd(ri), V = input;
motion_estimator_calc(32);
A = rd(ri), V = input;
/* repeat 30 times */
```

```
;;;;;

B = rd(cr82); /* Read the minimum value */
B = rd(cr83); /* Read the XY position */
```

Mnemonics of the Motion Estimator

```
motion_estimator_load;
motion_estimator_load ( coefficients, load_data )
motion_estimator_calc;
motion_estimator_calc ( calc_data );
motion_estimator_calc ( repeat );
```

Program Example:

```
motion_estimator_load ( 16,15 );
A = rd (ri++),  V = input;
A = rd (ri++),  V = input;
. . . . . .
A = rd (ri++),  V = input;
A = rd (ri++);


motion_estimator_calc ( 33 );
A = rd (ri++),  V = input;
A = rd (ri++),  V = input;
A = rd (ri++),  V = input;
```

cr80: MES_Control: Control register of the Motion Estimator (read / write)

Column Enables [15:0]																Soft Count Parameters								Row Enables [3:0]				NS en	NR en	MTe n	AS en
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr81: MES_SumOfDiff (Sub)-Total Output (read only)

'00000000'								(Sub)-Total Output [23:0]																							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr82: MES Minimum: Minimum Sum of Differences found (read only)

'00000000'								Minimum Sum of Differences found [23:0]																							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

cr83: MES_Position: Position of the Minimum Sum of Differences found (read only)

'0000'				X position of the Minimum [11:0]												'0000'				Y position of the Minimum [11:0]											
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Chapter

24. VIDEO TIMING GENERATORS

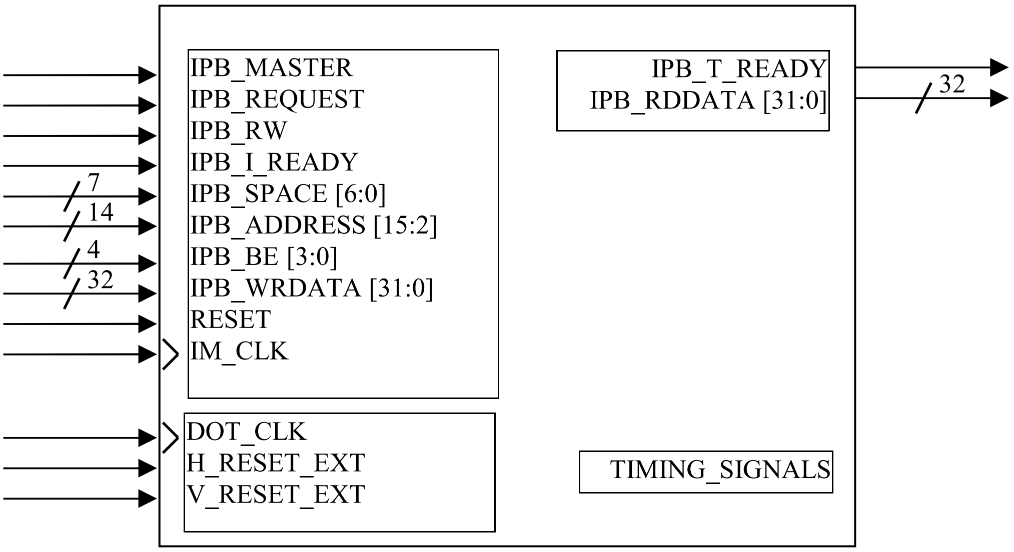
The Video Timing Generators execute timing instructions from their own Timing instruction RAM and are capable op generating arbitrary video timing signals up to a resolution of 4096 by 4096 pixels, including CCIR601, NTSC and PAL-M formats. There are 2 timing generators, one for output and one for input.

The video output timing generator sends its timing signals to the video output unit. It can be to the external Vreset pin or to the CCIR 656 video input. It can choose between the clock from the internal dot clock generator, the CCIR 656 video input clock divided by 1, 2 or 4 or the Imagine clock for testing. The video output unit operates on the same clock as the video output timing generator.*

The video input timing generator can be synchronised to the CCIR 656 video input. It runs on the CCIR 656 video input clock divided by 1, 2 or 4 or the Imagine clock for testing. The CCIR 656 video input unit operates on the same clock as the video input timing generator.

24.1 The I/O signals of the Video Timing Generator

24.1.1 schematic overview



24.1.2 signal definitions

IPB_MASTER IPB_REQUEST IPB_I_READY IPB_SPACE [6:0] IPB_ADDRESS [15:2] IPB_BE [3:0] IPB_WRDATA [31:0] RESET IM_CLK IPB_T_READY IPB_RDDATA	input input input input input input input input input output output	See : “The protocol of the Internal Peripheral Bus”
DOT_CLK H_RESET_EXT V_RESET_EXT TIMING_SIGNALS [7:0]	input input input output	External DOT clock. (May not be present) External horizontal sync signal (Only for SLAVE mode) External vertical sync signal (Only for SLAVE mode) 8 timing signals progammed as low 8 bits in the program memory. Bit definitions are : 7 Vsync 6 Hsync 5 Blank 4 reserved 3 reserved 2 reserved 1 Vertical interrupt (VInt) 0 Horozontal interrupt (Hint)

24.2 Module overview of the Video Timing Generator (VTG)

The Video Timing Generator consists of seven modules.

1. IPB-interface.
2. Horizontal counter (12-bit).
3. Vertical counter (12-bit).
4. Decoder (Contains the EOB and EOS state flip flops).
5. Program Counter (PC).
6. Instruction RAM.
7. Read multiplexer.

24.2.1 The IPB_interface

The IPB-interface connects the VTG to the Internal Peripheral Bus. It decodes the IPB request and determines the appropriate action for the request. All accesses through the IPB to this unit require multiple cycles because the VTG can operate asynchronously to the IMAGINE on the DOT_CLK.

24.2.2 The counters

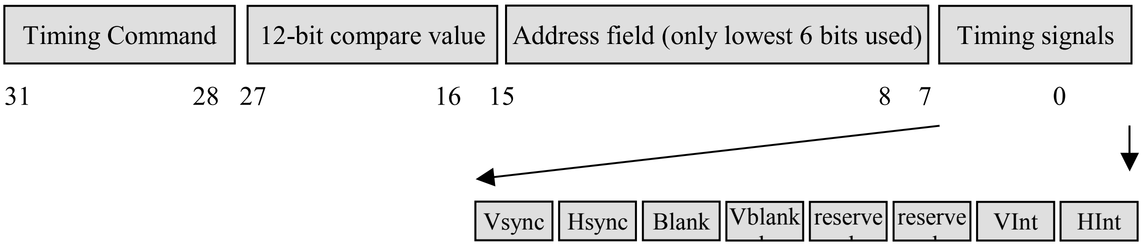
There are two 12-bit counters (Horizontal and Vertical) to allow for a 4096x4096 pixel display. The 6-bit program counter (PC) points to the current instruction address in the VTG instruction RAM, allowing for 64 instruction addresses.

24.2.3 The Decoder

The decoder controls the functioning of the VTG. The current mode (RESET, HOLD, MASTER/SLAVE) and current state of the counters are used to determine the control of the other units (all except the IPB-interface).

24.2.4 The Instruction RAM

The Instruction RAM is a three port RAM with two read ports and one write port. The one read port is used exclusively for the decoder. The other tow ports are used exclusively for the IPB interface. The RAM contains 64 words of 32-bits.



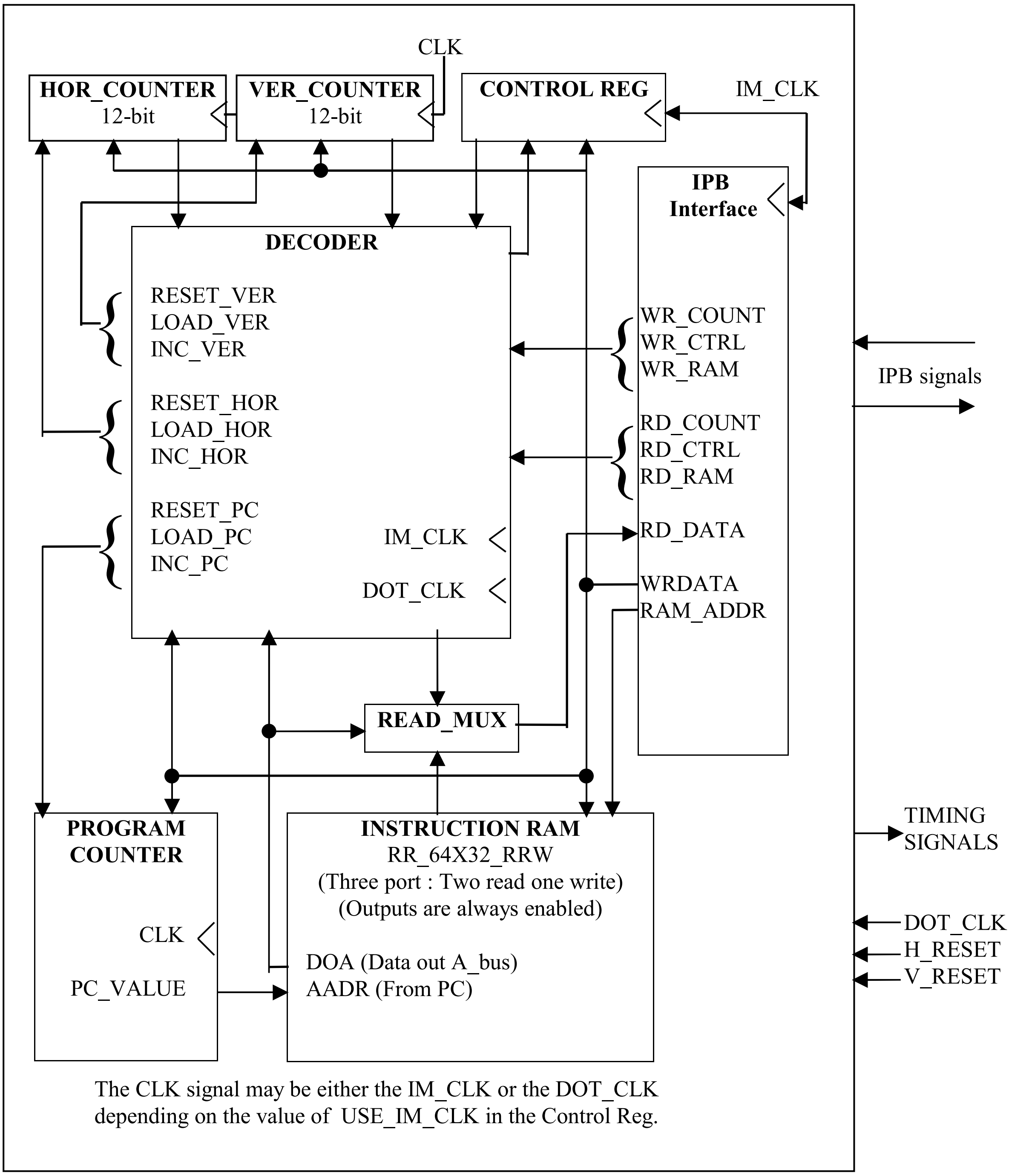
The 32-bit instruction word contains four fields

- The instruction (2-bits, 2-reserved).
- The 12-bit compare value for both horizontal and vertical compares, depending on the instruction.
- The address field for loading the program counter. This is only used for instruction 1 (only if EOB and EOS are false). For all the other instructions the address field is don't care.
- The video timing signals (8-bits). The timing signals include horizontal sync, vertical sync, blank and interrupts. Importantly the current instruction always specifies the value of the output timing signals. Care should therefore be taken in programming the video timing generator to achieve the desired output signals. On start-up the program memory is uninitialized and should always be programmed before enabling the unit.

24.2.5 The Read multiplexer

The read multiplexer selects the required read data from the IPB request. Read requests can always be done from the IPB even if the unit is functioning on the DOT_CLK.

Block diagram of the Video Timing Generator



The CLK signal may be either the IM_CLK or the DOT_CLK depending on the value of USE_IM_CLK in the Control Reg.

24.3 Functional description of the Video Timing Generator

The VTG executes the timing instructions in the Instruction RAM and is capable op generating arbitrary timing signals upto a resolution of 4096 by 4096 pixels, including CCIR601, NTSC and PAL-M formats.

The VTG has two main modes of operation. The first mode is used to control the unit from the IPB. In this mode the unit can be tested and the instruction RAM programmed for the required video display format. On initialisation the instruction RAM contains random values and has to be programmed before activation.

The second mode of operation is used to generate the timing signals for the display format. Usually the VTG runs asynchronously to the IMAGINE clock from the externally applied DOT clock. It can however be programmed to run on the IMAGINE clock if for instance the DOT clock is not available. The VTG has only four instructions, but this is sufficient for an arbitrary complex display format.

24.3.1 Video Timing Generator instruction description

The four allowed instructions are as follows:

Instruction 0 : Wait for line segment end.

This instruction is repeated until the horizontal counter is equal to the compare value given in the instruction. The address counter is incremented to the next instruction in this case. (The address load value is don't care for this instruction).

Instruction 1 : Wait for line end.

This instruction is repeated until the horizontal counter is equal to the instruction compare value. It the values are equal :

- If EOB and EOS are both false (End of screen, End of band), load the address counter with the value given in the instruction. This is intended for repeating the same line.
- If EOB is true and EOS is false, increment the address counter to the next instruction. This is intended to start the next screen band.
- If EOS is true, reset the address register and all registers and flags.

In all three cases reset the horizontal counter and EOB and EOS flags.

Instruction 2 : Test if the current line is the is the last of a Band of the screen.

Set the EOB flag if the vertical counter is equal to the compare value of the instruction, else reset EOB. Always reset EOS. Always increment the program counter and horizontal counter. (The address load value is don't care for this instruction).

Instruction 3 : Test if the current line is the last of the screen.

Set the EOS flag if the vertical counter is equal to the compare value of the instruction, else reset EOS. Always reset EOB. Always increment the program counter and horizontal counter. EOS indicates the end of the entire frame in case of an interlaced format.

24.4 Sample program for the Video Timing Generator

The following is an example format of 14 lines by 18 pixels. The visible display area is 6 lines by 9 pixels (lines 5-10, pixels 6-14). Each pixel shows by which instruction it is generated.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0	0	0	1	1	1	2	3	3	3	3	3	3	3	3	4	4	4
1	0	0	0	1	1	1	2	3	3	3	3	3	3	3	3	4	4	4
2	5	5	5	6	6	6	7	8	8	8	8	8	8	8	8	9	9	9
3	5	5	5	6	6	6	7	8	8	8	8	8	8	8	8	9	9	9
4	5	5	5	6	6	6	7	8	8	8	8	8	8	8	8	9	9	9
5	10	10	10	11	11	11	12	13	13	13	13	13	13	13	13	14	14	14
6	10	10	10	11	11	11	12	13	13	13	13	13	13	13	13	14	14	14
7	10	10	10	11	11	11	12	13	13	13	13	13	13	13	13	14	14	14
8	10	10	10	11	11	11	12	13	13	13	13	13	13	13	13	14	14	14
9	10	10	10	11	11	11	12	13	13	13	13	13	13	13	13	14	14	14
10	10	10	10	11	11	11	12	13	13	13	13	13	13	13	13	14	14	14
11	15	15	15	16	16	16	17	18	18	18	18	18	18	18	18	19	19	19
12	15	15	15	16	16	16	17	18	18	18	18	18	18	18	18	19	19	19
13	15	15	15	16	16	16	17	18	18	18	18	18	18	18	18	19	19	19

The screen format above is generated with the following program:

Addr.	**video timing command**	**compare value**	**load. address**	**video timing signals**
0	Wait for segment end	comp=2	addr=x	hsync=0, vsync=0, blank=0
1	Wait for segment end	comp=5	addr=x	hsync=1, vsync=0, blank=0
2	Test if line=end of band	comp=1	addr=x	hsync=1, vsync=0, blank=0
3	Wait for segment	comp=14	addr=x	hsync=1, vsync=0, blank=0
4	Wait for line end	comp=17	addr=0	hsync=1, vsync=0, blank=0
5	Wait for segment end	comp=2	addr=x	hsync=0, vsync=1, blank=0
6	Wait for segment end	comp=5	addr=x	hsync=1, vsync=1, blank=0
7	Test if line=end of band	comp=4	addr=x	hsync=1, vsync=1, blank=0
8	Wait for segment end	comp=14	addr=x	hsync=1, vsync=1, blank=0
9	Wait for line end	comp=17	addr=5	hsync=1, vsync=1, blank=0
10	Wait for segment end	comp=2	addr=x	hsync=0, vsync=1, blank=0
11	Wait for segment end	comp=5	addr=x	hsync=1, vsync=1, blank=0
12	Test if line=end of band	comp=10	addr=x	hsync=1, vsync=1, blank=1
13	Wait for segment end	comp=14	addr=x	hsync=1, vsync=1, blank=1
14	Wait for line end	comp=17	addr=10	hsync=1, vsync=1, blank=0
15	Wait for segment end	comp=2	addr=x	hsync=0, vsync=1, blank=0
16	Wait for segment end	comp=5	addr=x	hsync=1, vsync=1, blank=0
17	Test if line=end of screen	comp=13	addr=x	hsync=1, vsync=1, blank=0
18	Wait for segment end	comp=14	addr=x	hsync=1, vsync=1, blank=0
19	Wait for line end	comp=17	addr=15	hsync=1, vsync=1, blank=0

24.5 Function Table of the Video Timing Generator

	Res	Hold	M/S	V_res	H_res	Instr	Match	EOS	EOB	PC	VER cntr	HOR cntr	EOS	EOB
RESET	1	x	x	x	x	x	x	x	x	Res	Res	Res	Res	Res
HOLD	0	1	x	x	x	x	x	x	x	Hold	Hold	Hold	Hold	Hold
V_RESET_EXT	0	0	S	1	x	x	x	x	x	Res	Res	Res	Res	Res
H_RESET_EXT	0	0	S	0	1	x	x	0	0	Load	Incr	Res	Res	Res
			S	0	1	x	x	0	1	Incr	Incr	Res	Res	Res
			S	0	1	x	x	1	x	Res	Res	Res	Res	Res
Instr type 0	0	0	M/S	x/0	x/0	0	0	x	x	Hold	Hold	Incr	Hold	Hold
Segment end			M/S	x/0	x/0	0	1	x	x	Incr	Hold	Incr	Hold	Hold
Instr type 1	0	0	M/S	x/0	x/0	1	0	x	x	Hold	Hold	Incr	Hold	Hold
Line end			M/S	x/0	x/0	1	1	0	0	Load	Incr	Res	Res	Res
			M/S	x/0	x/0	1	1	0	1	Incr	Incr	Res	Res	Res
			M/S	x/0	x/0	1	1	1	x	Res	Res	Res	Res	Res
Instr type 2	0	0	M/S	x/0	x/0	2	0	x	x	Incr	Hold	Incr	Res	Res
Test EOB			M/S	x/0	x/0	2	1	x	x	Incr	Hold	Incr	Res	Set
Instr type 3	0	0	M/S	x/0	x/0	3	0	x	x	Incr	Hold	Incr	Res	Res
Test EOS			M/S	x/0	x/0	3	1	x	x	Incr	Hold	Incr	Set	Res

24.6 Interfacing with the Video Timing Generator through the IP

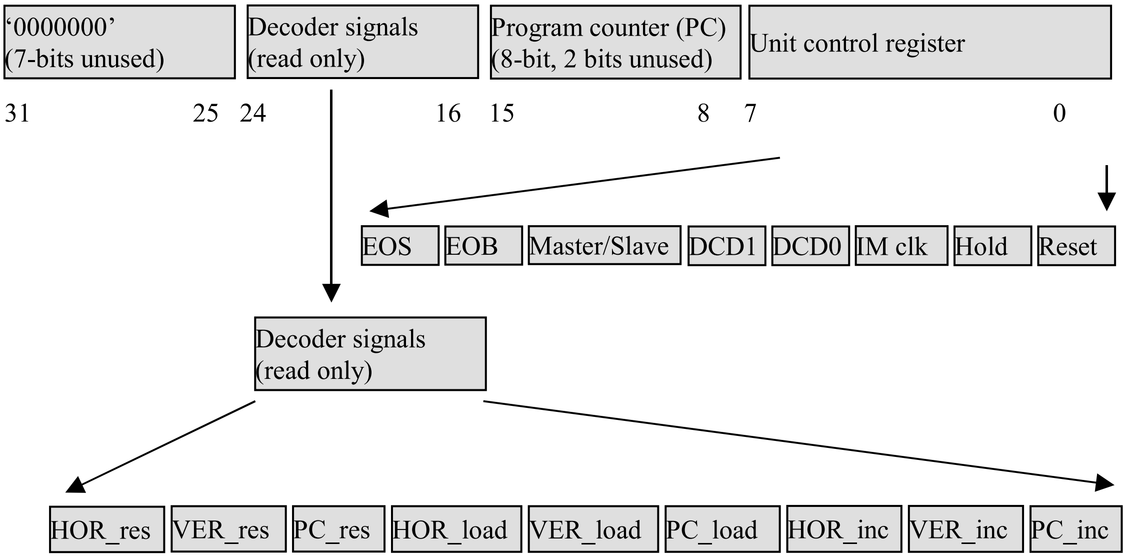
All read and write accesses to and from the Video Timing Generator require multiple cycles for completion because the VTG operates asynchronously to the IMAGINE during normal operation. There are basically three different accesses depending on the selected address (IPB_ADDRESS).

1. Control register (VTG_BASE + 128)
2. Counter register (Horizontal and Vertical) (VTG_BASE + 129)
3. Instruction RAM (VTG_BASE -> VTG_BASE + 63)

Read accesses can always be performed to any of these areas, even if the VTG is using the DOT_CLK. This allows for determining the state of the VTG while it is operational (generating timing signals). Writing to the counter register and part of the control register (PC bits 15-8) can only be done while the VTG is in HOLD mode (Hold bit in control register = 1). This avoids interference with the functioning of the unit during normal operation.

24.6.1 The Control register

The control register has the following fields :



The control register (lowest 8 bits) determines the general state of the VTG. Although the EOS and EOB bits form part of the control register from a programmers perspective, they are actually state flip-flops within the decoder and usually function asynchronously to the IMAGINE on the DOT clock. The lower 6 bits are however always synchronous to the IMAGINE to allow control of the unit without the DOT clock. The low 6- bits can therefore always be accessed from the IPB, but the EOS and EOB can only be changed if the unit is in a HOLD mode (Hold bit set). This avoids interference with the unit during normal operation. The same applies to the Program counter which can also only be change while in a Hold mode.

24.6.1.1 The Unit control register

Reset This is a reset bit for the VTG without the IPB interface. This bit is intended to be used just before starting normal operation of the VTG after testing and programming of the instruction RAM. If this bit is set, the VTG will reset its internal state. All counters will be zeroed (Horizontal pixel counter, Vertical pixel counter and Program counter) as well as the EOB and EOS flags.

Hold This bit is intended to place the VTG in a hold mode for general access from the IPB. In this mode all registers are accessible for reading and writing. The value of the IM_clk bit may also be changed, but for general access it should be set to ‘1’ (use IMAGINE clock) and only be cleared after all testing and programming is complete.

IM_clk If this bit is set the unit will operate using the IMAGINE clock instead of the DOT clock.

DCD1, DCD0 These bits determine the clock pre-scale value. During testing they should be cleared.

00	Divide clock by 1
01	Divide clock by 2
10	Divide clock by 4
11	Divide clock by 8

Master/Slave The VTG can function in a Master or a slave mode. In Master mode the external signals HOR_RESET_EXT and VER_RESET_EXT are ignored and the VTG will just execute the timing program as defined in the instruction RAM. The Slave mode is intended to synchronise the unit to an external source. In Slave mode the unit will function normally while HOR_RESET_EXT and VER_RESET_EXT are low (‘0’) and is only affected if one or both of them are high (‘1’). If the VER_RESET_EXT is high (HOR_RESET_EXT ignored), the internal state will be reset (synchronous to the clock) to start a new screen (Pixel counters, PC, EOB and EOS all reset). If the HOR_RESET_EXT is high a horizontal reset will be executed (synchronous to the clock). The exact operation depends on the current state of the VTG but corresponds to reaching a line end during normal operation (executing Instruction type 1 with the Horizontal counter equal to the 12-bit compare value of the instruction) (See section 3.1 Instruction type 1 when the Horizontal counter equals the compare value).

24.6.1.2 The Program counter

The program counter can only be changed while in the Hold mode. Eight bits are defined although only 6 are implemented (2^6 = 64). The two remaining bits will return ‘00’ when read.

24.6.1.3 The Decoder signals

These signals are read only and writing to them has no effect. They reflect most of the current decoder control signals to facilitate testing or possible state determination during normal operation. These values will rarely be required.

24.6.2 The Counter register

The counter register has the following fields. Writing from the IPB can only be performed while the unit is in a Hold mode. Byte enables are also used during writing to allow 8-bit accesses.

‘0000’ (4-bits unused)	Horizontal pixel counter (12-bit)	‘0000’ (4-bits unused)	Vertical pixel counter (12-bit)
31 28	27 16	15 12	11 0

24.6.3 The Instruction RAM

Any 32-bit value can be written to the instruction RAM, but in general the values should conform to the instruction specification as described in section 3.1. Due to the fact that a three port RAM is used, reading and writing to the instruction RAM can always be done, even while the unit is operational (generating timing signals). This allows for on the fly changing of the VTG program by accessing areas which are not currently required by the unit. This creates a possibility to extend the capabilities of the VTG if the 64 word RAM is not sufficient. Extreme caution should however be taken not to write to the same area of RAM currently being accessed by the unit, as this will probably result in indeterminate operation. On the fly alteration of the VTG instruction RAM is possible, but will very rarely be required as the current 64 word address space is sufficient for all the standard display formats.

24.7 Programmers Notes

The following should be kept in mind with regards to the VTG

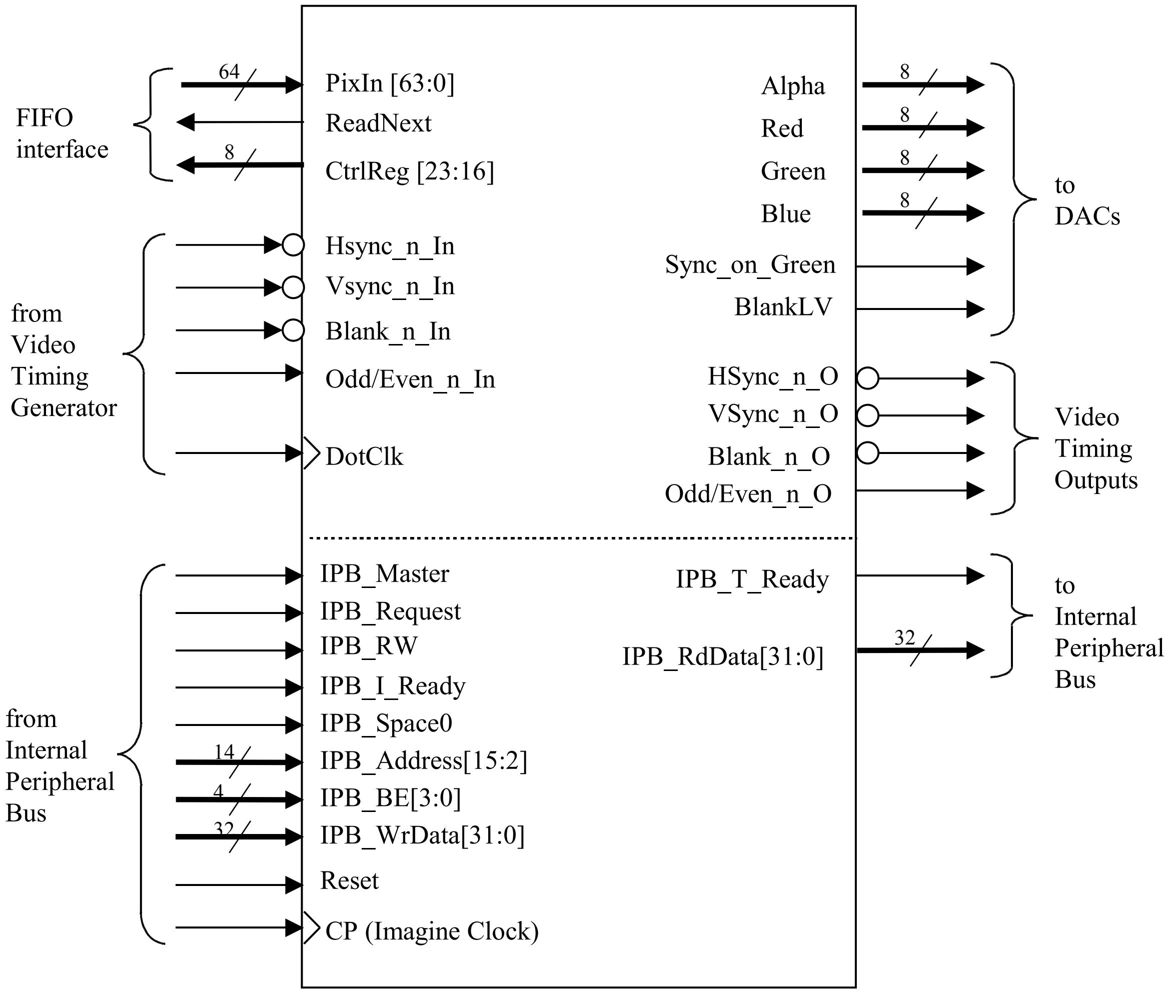
- The IPB RESET signal overrides all other signals. When this signal is active, no operation or access is possible.
- When writing to the internal state registers (Horizontal and Vertical counters, Program counter, EOS and EOB flags) can only be done when in a Hold mode.
- When the unit is in a Hold mode, the IM_clk bit should be set (use IMAGINE clock) and DCD1 and DCD0 should be cleared to avoid problems.
- In Slave mode the unit functions normally if the external reset signals are low (‘0’). If these signals never become active, there is no change between Master and Slave mode. This means that a valid program is still required. Importantly a program induced horizontal or vertical reset will be executed if the program reaches that state before the externally applied signals. This would probably not be the required “slave” operation. In order to avoid this, the horizontal compare value for the “Wait for line end” instructions (Instruction type 1) should be increased (even set to 4095) to insure that the external reset signals be acknowledged.
- Care should be taken to supply the unit with a valid program. In general the compare values in the instructions should always be greater than or equal to the relevant Horizontal or Vertical counter. If the instruction compare value is less than the current counter value, the counter will have to overflow through zero before the reaching the compare value.
- Various programs may produce the same results. Instruction types 2 and 3 usually have different placement possibilities.

Chapter

25. VIDEO OUTPUT UNIT

The Video Output Unit receives video timing information from the video output timing generator and pixel data from the video output fifo. It translates 32 bit, 16 bit hi-color or 8 bit pseudo color pixel information into 32 bit alpha, red, green, blue information. The three color components go to the video DAC and the alpha value can be re-directed to the 8 bits of the digital video port. A 32x32x2 bit VGA compatible hardware cursor is also provided. 3x256x8 bit color look up tables can be used for pseudo color to true color conversion or to used for true color matching as required by the PC98 standart

25.1 The Input / Output Signals of RAMDAC (digital circuit)



25.1.1 Input/ Output signals definitions

Fifo Interface signals		
PixIn [63:0]	input	64 bit pixel data from FIFO. Data formats are: 1) 8bit x 8 words (pseudo color) 2) 16 bit x 4 words (direct color) 3) 32 bit x 2 words (direct color)
ReadNext	output	Data read request to FIFO (active high)
CtrlReg [23:16]	output	FIFO control register outputs. bit 16: Fifo Enable bit 17: Almost Empty Interrupt Enable bit 19, 18: Reserved (reset to ‘0’) bit 23..20: Watermark of Almost Empty Interrupt

Video Timing signals (from Video Timing Generator)		
HSync_n_In	input	Horizontal sync input from Video Timing Generator (active low) . This signal is synchronized with DotClk.
Vsync_n_In	input	Vertical sync input from Video Timing Generator (active low). This signal is synchronized with DotClk.
Blank_n_In	input	Blank input from Video Timing Generator (active low). This signal is synchronized with DotClk.
Odd_Even_n_In	input	Odd or even field input from Video Timing Generator. Odd_Even_n_In indicates odd or even field during interlaced display. When *Odd/Even polarity* register (*Cursor Control Register* bit 13) set to 1, a low signal indicates the even field and a high signal indicates the odd field. The polarity can be inverted by value of *Odd/Even polarity* register. This signal is synchronized with DotClk.
DotClk	input	Pixel clock input.
Color output signals (To DACs)		
Alpha [7:0]	output	DotClk synchronized alpha color outputs. These signals are connected to digital input bus of DAC cell.
Red [7:0]	output	DotClk synchronized red color outputs. These signals are connected to digital input bus of DAC cell.
Green [7:0]	output	DotClk synchronized green color outputs. These signals are connected to digital input bus of DAC cell.
Blue [7:0]	output	DotClk synchronized blue color outputs. These signals are connected to digital input bus of DAC cell.
Sync_on_Green	output	When this signal is high, sync signal is added to Green Analog Output of DAC cell.
BlankLV	output	Blank level select signal. When set to high, blank signal is added to analog outputs of DAC cell (black level > blank level). When set to zero, black level equals to blank level..
Video Timing output signals		
HSync_n_O	output	Pipeline delayed HSync_n_In signal (active low). This signal is synchronized with DotClk.
Vsync_n_O	output	Pipeline delayed Vsync_n_In signal (active low). This signal is synchronized with DotClk.
Blank_n_O	output	Pipeline delayed Blank_n_In signal (active low). This signal is synchronized with DotClk.
Odd_Even_n_O	output	Pipeline delayed Odd_Even_n_In signal. Value of *Odd_Even Polarity* register (*Cursor Control Register* bit 13) is no effect to this signal. This signal is synchronized with DotClk.

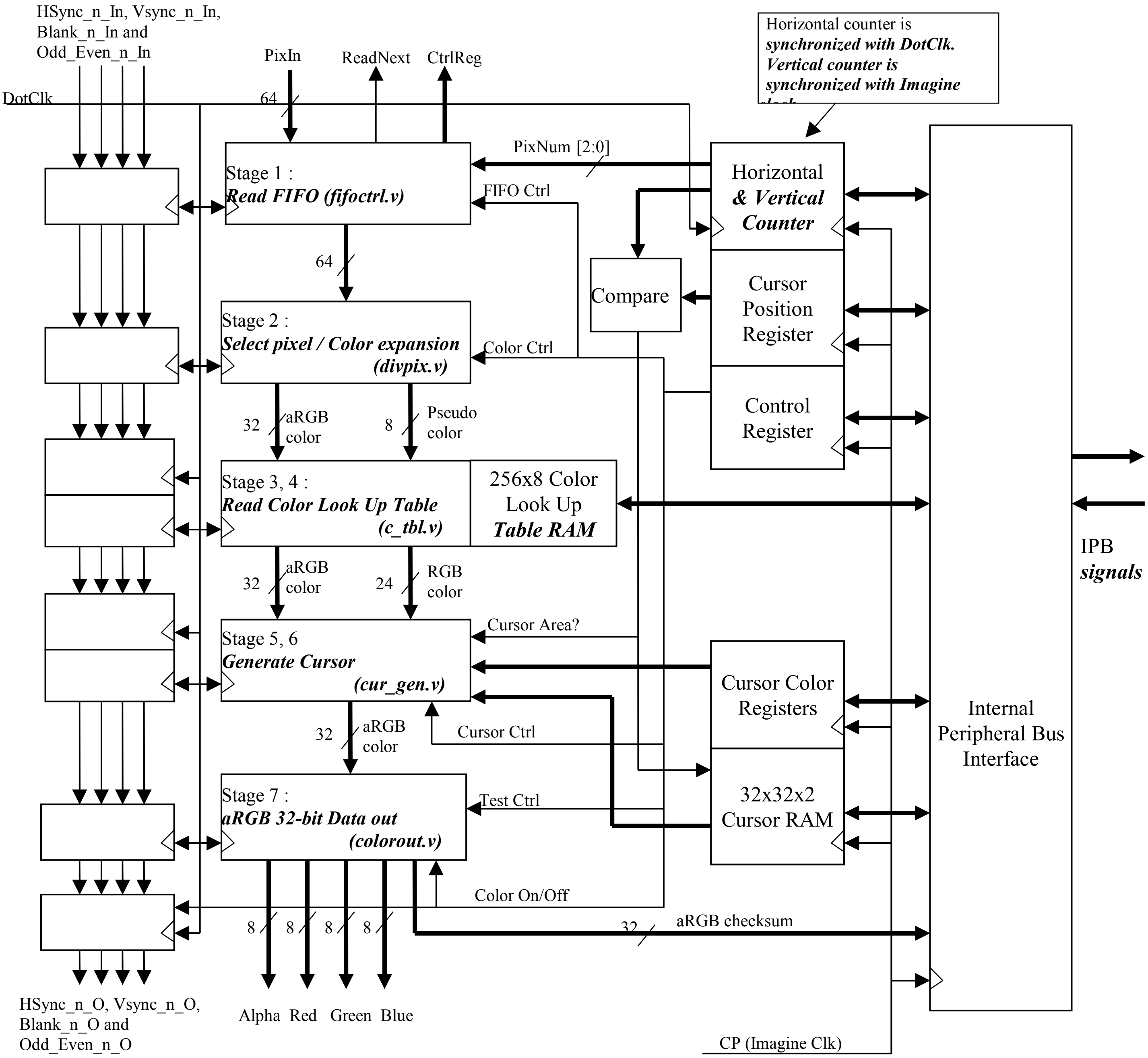
Internal Peripheral Bus I/F signals		
CP (Imagine Clock)	input	See document *"The Protocol of the INTERNAL PERIPHERAL BUS"*
Reset	input	
IPB_Master	input	
IPB_Request	input	
IPB_RW	input	
IPB_T_Ready	output	
IPB_I_Ready	input	
IPB_Space0	input	
IPB_Address[15:2]	input	
IPB_BE [3:0]	input	
IPB_RdData [31:0]	output	
IPB_WrData [31:0]	input	

25.2 RAMDAC module overview

RAMDAC module has two blocks. One is pixel data streams which is pipelined and DotClk synchronized. There are 7 stages to convert each pixels and to operate cursors.

1) Read 64 bit Pixels from FIFO
2) Select one Pixel (8-bit, 16-bit or 32-bit) from 64-bit pixels and 16- bit to 32 bit color expansion
3, 4) Pseudo color read from *Color Look Up RAM* (8-bit data only)
5, 6) Cursor operations
7) Output to DAC cells (select color on/ off)

Video timing signals from *Video Timing Generator* are pipeline delayed same as pixel data.
Another is control block. It has some control registers and interface logic connected to Internal Peripheral Bus. This block is synchronized with Imagine Clock.

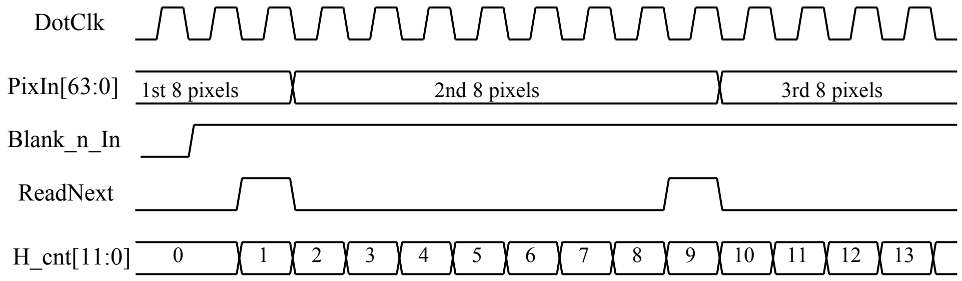

HSync_n_In, Vsync_n_In, Blank_n_In and Odd_Even_n_In
PixIn
ReadNext
CtrlReg
Horizontal counter is synchronized with DotClk. Vertical counter is synchronized with Imagine
DotClk
64
PixNum [2:0]
Stage 1 : Read FIFO (fifoctrl.v)
FIFO Ctrl
Horizontal & Vertical Counter
64
Compare
Cursor Position Register
Stage 2 : Select pixel / Color expansion (divpix.v)
Color Ctrl
Control Register
32
aRGB color
8
Pseudo color
Stage 3, 4 : Read Color Look Up Table (c_tbl.v)
256x8 Color Look Up Table RAM
32
aRGB color
24
RGB color
IPB signals
Stage 5, 6 Generate Cursor (cur_gen.v)
Cursor Area?
Cursor Color Registers
Internal Peripheral Bus Interface
32
aRGB color
Cursor Ctrl
Stage 7 : aRGB 32-bit Data out (colorout.v)
Test Ctrl
32x32x2 Cursor RAM
Color On/Off
8
8
8
8
32
aRGB checksum
HSync_n_O, Vsync_n_O, Blank_n_O and Odd_Even_n_O
Alpha Red Green Blue
CP (Imagine Clk)

25.3 Read FIFO (fifoctrl.v)

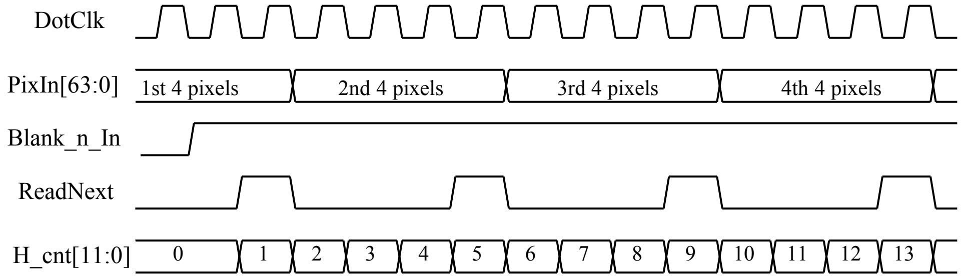
This block is first stage of pipelines. While Blank_n_In signal is high, this block generates ReadNext signal to FIFO, Read 64-bit pixel data from FIFO. The cycle of ReadNext is depend on the value of *Pixel Size Register* (*Color Control Register* bit 3, 2). This register selects incoming data size.

25.3.1 The timing of read from FIFO and ReadNext signal

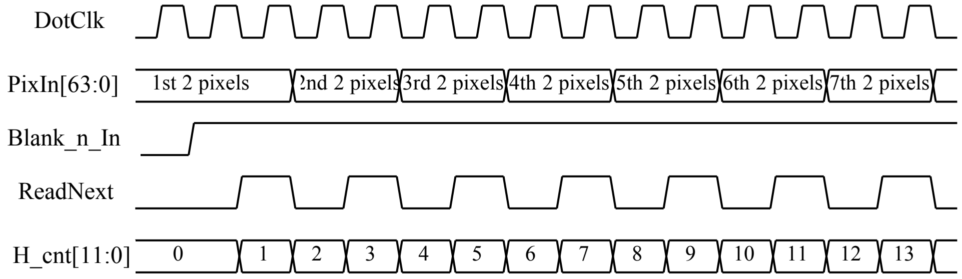
Case 1: Select 8 bit pixels



Case 2: Select 16 bit pixels



Case 3: Select 32 bit pixels



25.3.2 Input Data Format

8 bit Pixels

63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
Pixel 7 [7:0]	Pixel 6 [7:0]	Pixel 5 [7:0]	Pixel 4 [7:0]	Pixel 3 [7:0]	Pixel 2 [7:0]	Pixel 1 [7:0]	Pixel 0 [7:0]

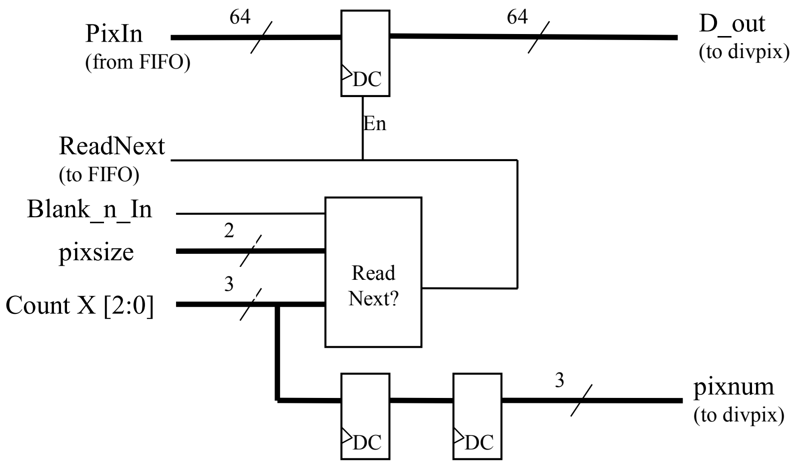
16 bit Pixels

63 48	47 32	31 16	15 0
Pixel 3 [15:0]	Pixel 2 [15:0]	Pixel 1 [15:0]	Pixel 0 [15:0]

32 bit Pixels

63 32	31 0
Pixel 1 [31:0]	Pixel 0 [31:0]

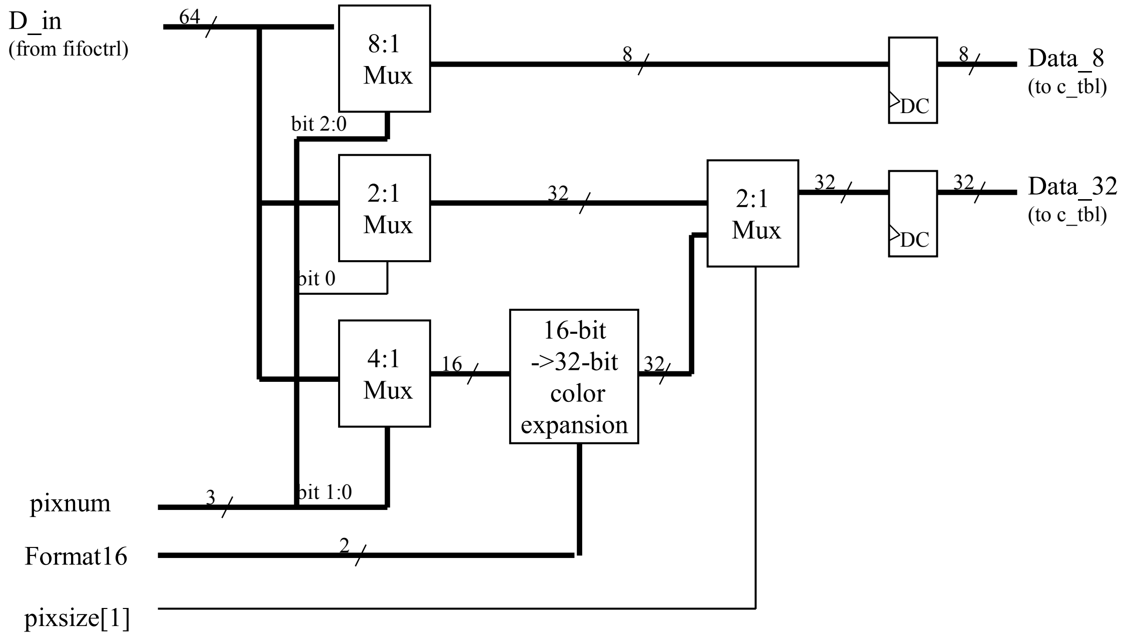
25.3.3 Block diagrams



25.4 Pixel select and 16 bit to 32 bit color expansion (divpix.v)

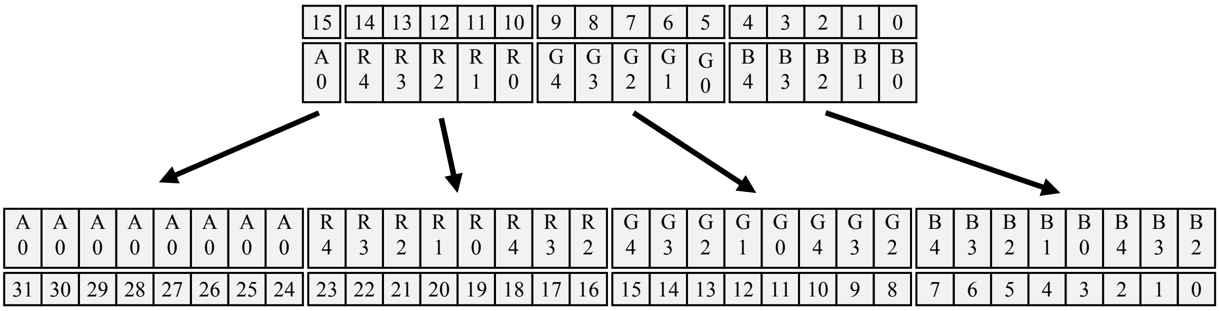
This block select one 32-bit, 16-bit or 8-bit pixel from 64-bit pixels. Pixel size is defined by *Pixel Size register* (*Color Control Register* bit 3, 2). Pixel select signal (pixnum[2:0]) is generated from Fifoctrl block. In case of 16-bit pixels, 16-bit pixels are expanded to 32-bit direct color data. Data format for 16-bit pixels is given by *Format 16 register* (*Color Control Register* bit 5, 4). See subsection *2.2.2 16-bit to 32-bit color expansion*, for more details. One dot clock cycle is needed to select pixel and color expansion. The block diagram of this module is as follows.

25.4.1 Block diagrams



25.4.2 16-bit to 32-bit color expansion

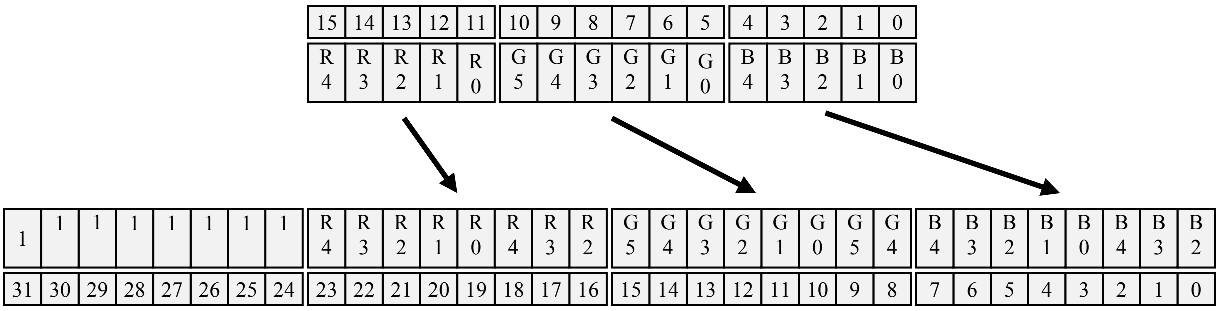
1555 -> 8888



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A0	R4	R3	R2	R1	R0	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0

A0	A0	A0	A0	A0	A0	A0	A0	R4	R3	R2	R1	R0	R4	R3	R2	G4	G3	G2	G1	G0	G4	G3	G2	B4	B3	B2	B1	B0	B4	B3	B2
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

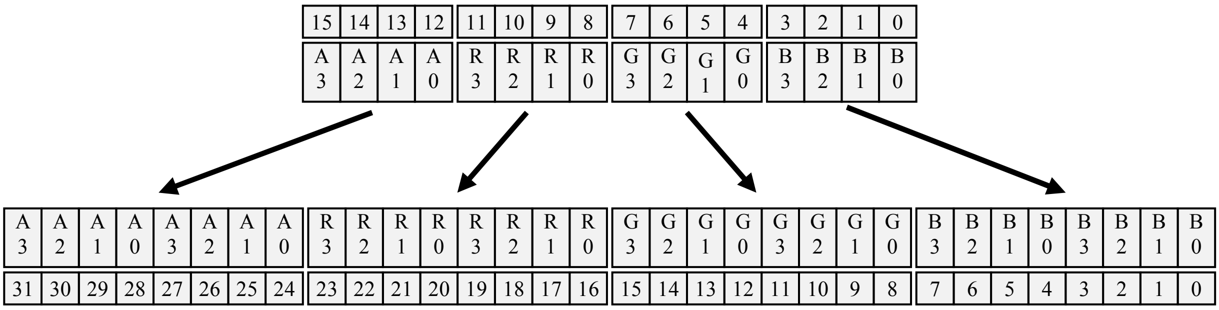
565 -> 8888



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0

1	1	1	1	1	1	1	1	R4	R3	R2	R1	R0	R4	R3	R2	G5	G4	G3	G2	G1	G0	G5	G4	B4	B3	B2	B1	B0	B4	B3	B2
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

4444 -> 8888

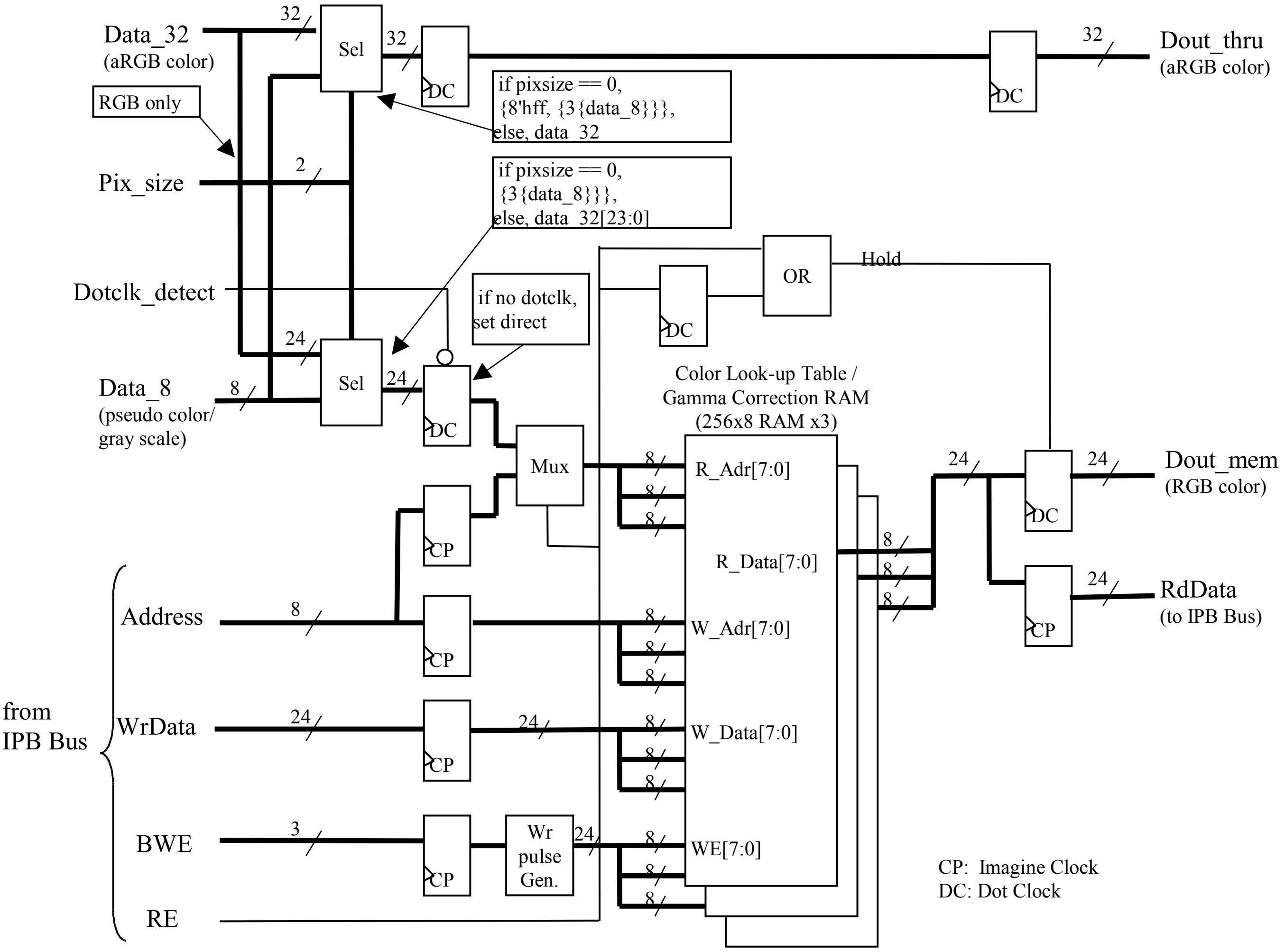


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A3	A2	A1	A0	R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0

A3	A2	A1	A0	A3	A2	A1	A0	R3	R2	R1	R0	R3	R2	R1	R0	G3	G2	G1	G0	G3	G2	G1	G0	B3	B2	B1	B0	B3	B2	B1	B0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

25.5 Read Look-up Table RAM (c_tbl.v)

In pseudo color mode, 8-bit pixel data is used as address the color look-up table RAM. Color look-up table RAM consists of 256x24 bit 2-port asynchronous RAM. 2 dot clock cycles is needed to read color look-up table. In case of 32-bit pixels, data is only 2 clock pipelined. Color look-up table RAM is not initialized and may be written or read from IPB at any time. When read color table by IPB bus, first hold output register of pipeline with valid data (current data), then switch multiplexer to IPB_Address. Output register is held 2 imagine clock cycles. The block diagram of this block is as follows.



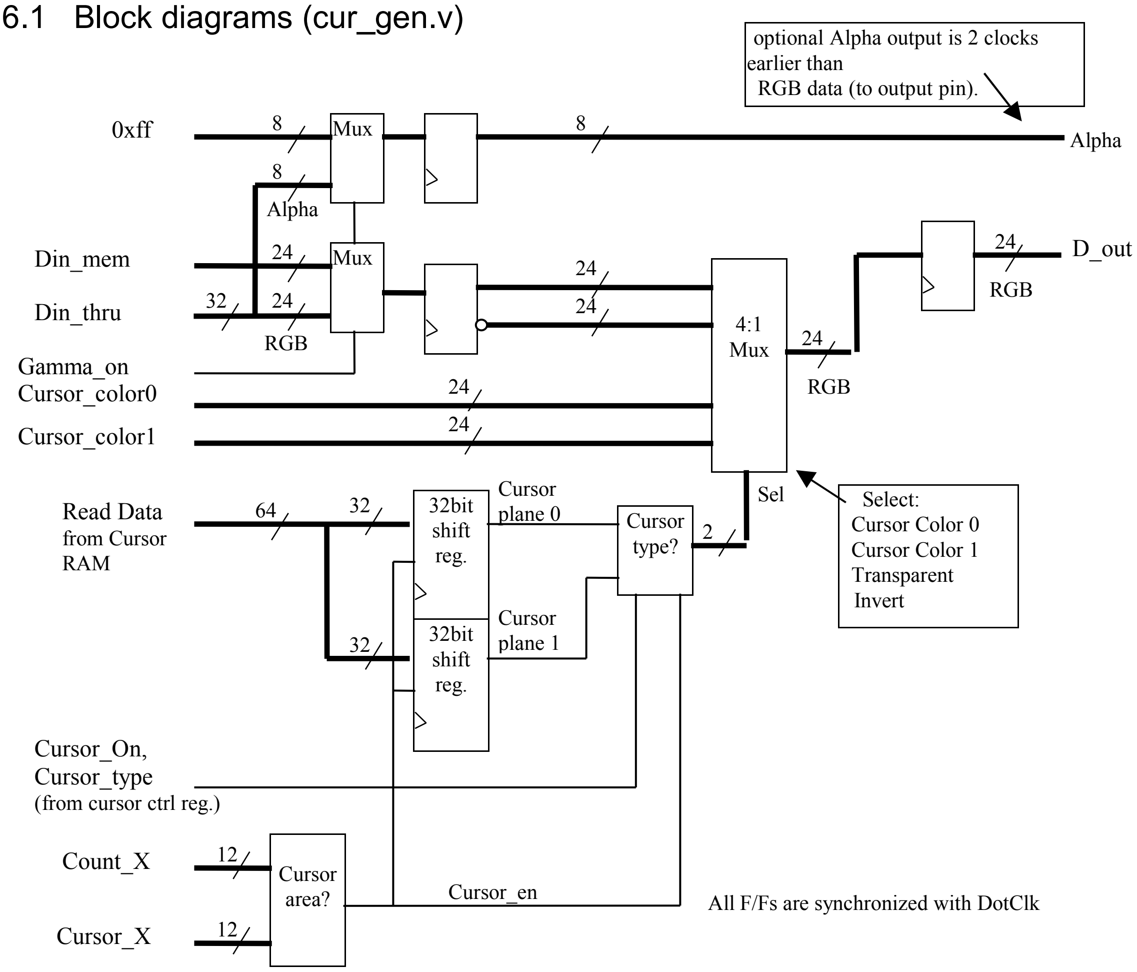
25.6 Cursor Generation (cur_gen.v)

This block generate two color 32x32 pixel cursor. Two dot clock cycles is needed to generate cursor. XGA cursor and X-Windows cursor modes are available. Cursor mode is defined by *Cursor Type register(Cursor Control Register* bit 9). The cursor operates in both non-interlaced and interlaced modes. It is defined by *Cursor interlace register(Cursor Control Register* bit 12*).* When *Cursor interlace register* set to 1, polarity of Even_Odd_n_In can be changed by *Cursor Polarity register*(*Cursor Control Register* bit 13).

The pattern for the 32x32 cursor is provided by the cursor RAM, which may be access from IPB at any time. Cursor positioning is performed using the *Cursor_x, Cursor_y* registers. Positions x and y are defined increasing from left to right and from top to bottom.

Block diagram of this blocks is as follows.

25.6.1 Block diagrams (cur_gen.v)



25.6.2 Cursor modes definitions

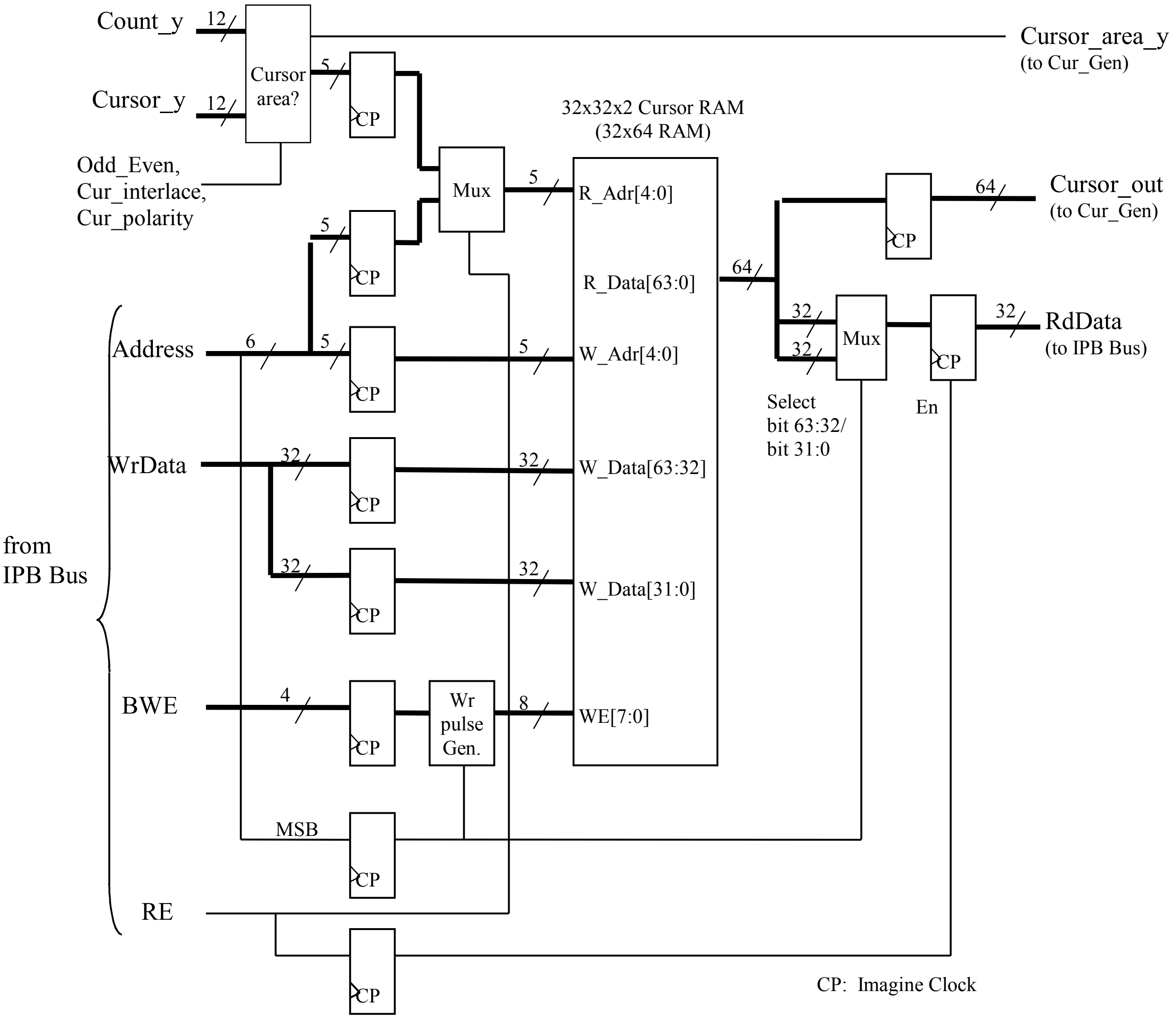
The 32x32x2 cursor RAM provides two bits of cursor information on every dot clock during the 32x32 cursor window. *Cursor Type register* (*Cursor Control Register* bit 9) specify XGA mode or X-Window mode. When *Cursor On register*(*Cursor Control Register* bit 8) is 0, the cursor is disabled. The two bits of cursor pixel data determine the cursor appearance as follows.

RAM		COLOR SELECTION	
PLANE1	PLANE0	XGA mode	X-Window mode
0	0	Cursor color 0	Transparent
0	1	Cursor color 1	Transparent
1	0	Transparent	Cursor color 0
1	1	Complement	Cursor color 1

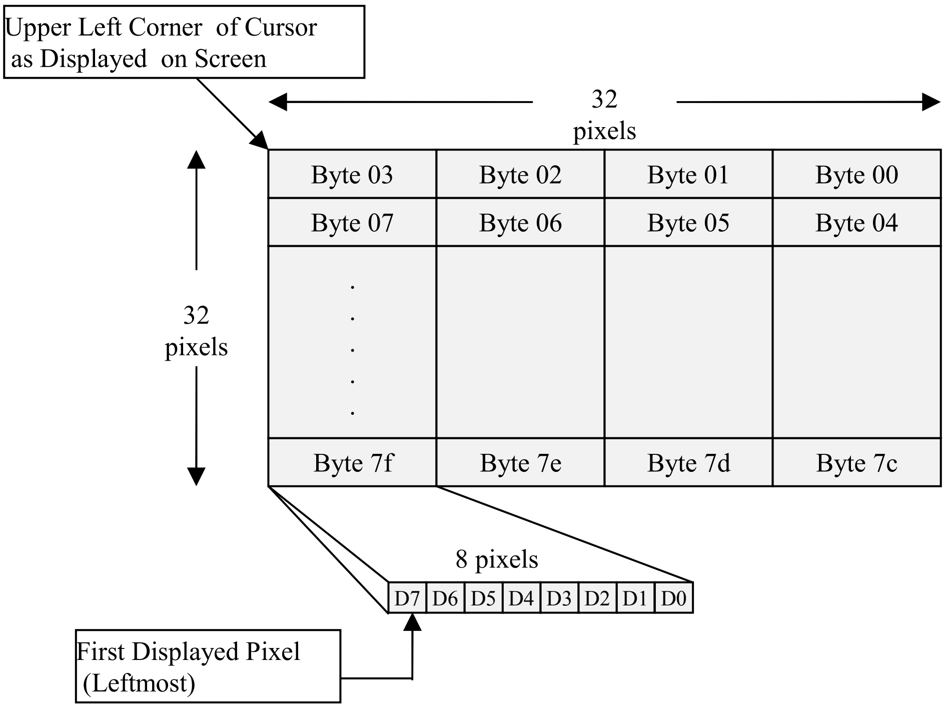
25.6.3 Cursor RAM

The 32x32x2 cursor RAM defines the pixel pattern within the 32x32 pixel cursor window. It is not initialized and may be written or read from Internal Peripheral Bus (offset 0x100 to 0x1ff) at any time.

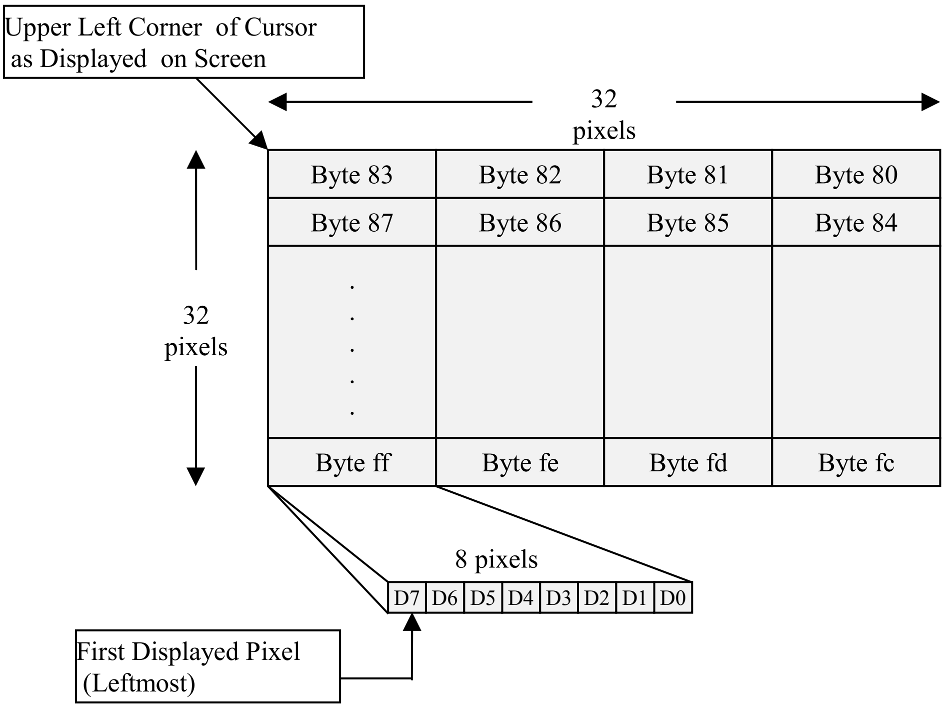
The cursor plane 0 bits for the entire cursor array are stored in the first 128 bytes of the RAM, and the cursor plane 1 bits are stored in the last 128 bytes of the RAM. Information for eight cursor pixels is stored in each byte. Each four bytes of pixels makes one line of the cursor. This 32 bit x 2 plane data is read from cursor generation block each vertical lines and stored to shift register while value of vertical counter in cursor window. Then while value of horizontal counter is in cursor window, the shift register is shift to left every dot clock.



,b,t,q,r,n,q ,o,k,`,m,d ,O



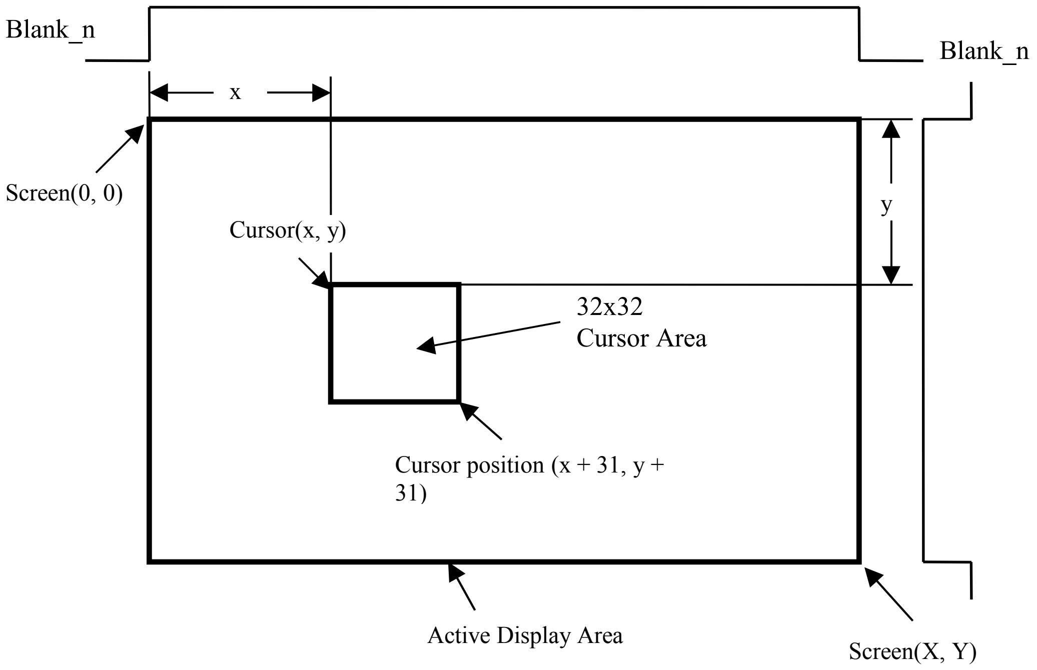
,b,t,q,r,n,q ,o,k,`,m,d 1



25.6.4 Cursor positioning

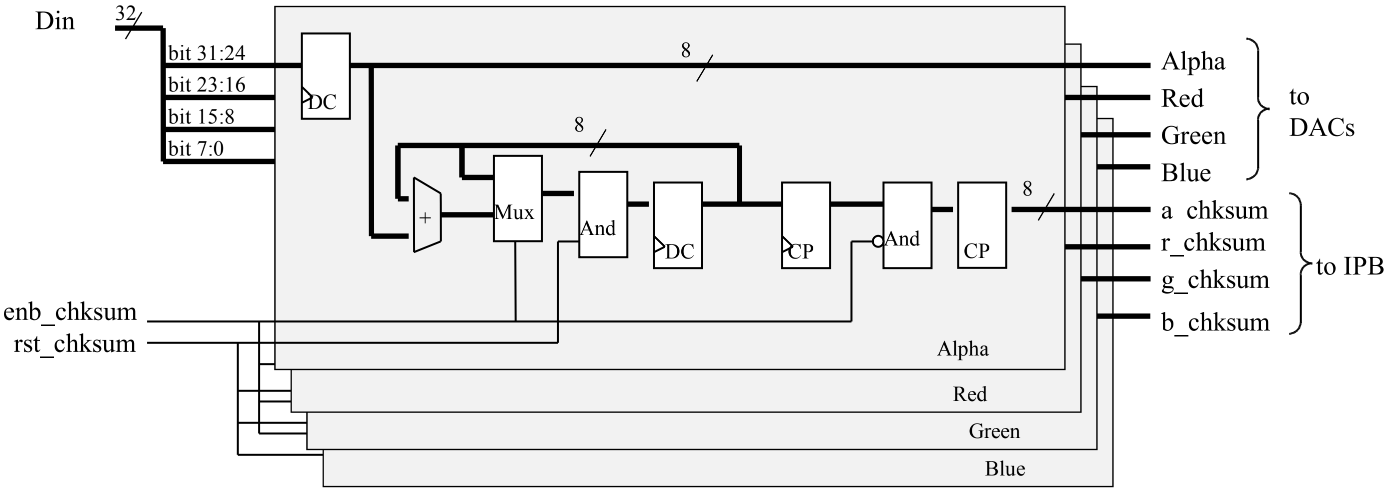
The cursor position x, y registers position the 32x32 cursor on the display screen. The cursor position x, y registers specify the location of the cursor top left corner on the display screen relative to the end of the Blank_n_In signal. Figure shows the orientation of the x, y coordinates for positioning the cursor.

The values written to the cursor position registers represent the position of the top left corner of the cursor. When value X is written to the cursor position x or value Y is written to cursor position y registers, the cursor is off the screen. When the cursor position x, y is (X - 1, Y - 1), only a single pixel of the cursor is displayed and it appears at the lower right corner of the screen.



25.7 Color Data out (colorout.v)

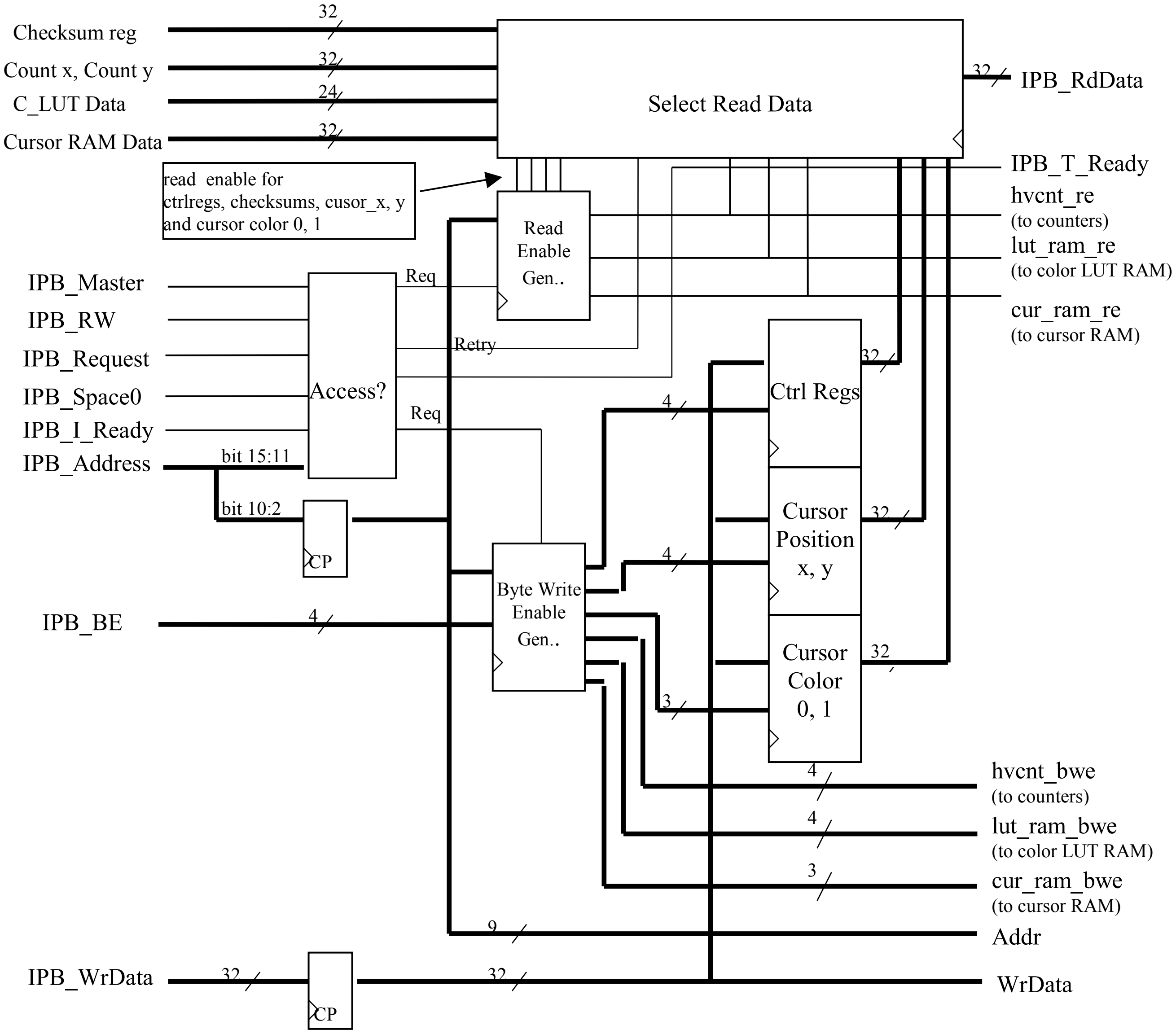
This block has the 32-bit alpha RGB registers for the digital color outputs and checksum registers for each 8-bit color. When *Color On register(Color Control Register* bit 0) is 1, color outputs are disabled (all '0'). The checksum registers is connected to 32-bit color output registers (at the end of the pipeline) and checksum values are calculated each 8-bit colors. When *Test Control Register* bit 24 set to 1, checksum registers are enabled (calculate checksum). When set to 0, checksum registers are held its values. These values are synchronized with Imagine clock using two 32-bit registers. It can be read from Internal Peripheral Bus (offset 0x008). While calculate checksum read values are all '0'. When bit 25 of *Test Control Register* set to 1, checksum registers reset to '0'.The block diagram of this module is as follows.



CP: Imagine clock
DC: Dot Clock

25.8 Internal Peripheral Bus I/F

RAMDAC module has Internal Peripheral Bus(IPB) interface giving direct access to the registers and memories of this module. These are mapped on IPB address space 0 (IPB_Space0). 2048 byte ranges are needed. These registers and memories are addressed directly by IPB_Address[10:2] from IPB and can be written or read at any time. When write access, write transfer always successfully terminate at first try. IPB_Address[10:2] and IPB_WrData[31:0] are stored, then write to registers and memories at next cycle. In case of read access, it takes two cycles to read from registers and four cycles to read from memories (color look-up table or cursor RAM).



25.8.1 RAMDAC base address

IPB_Address

<table>
<tr><td>‘0’</td><td colspan="5">0 0 0 1 0</td><td colspan="10">RAMDAC Register Space</td></tr>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

25.8.2 RAMDAC registers memory map

<table>
<tr><td></td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td colspan="8">Test Control
Register</td><td colspan="8">FIFO Control
Register</td><td colspan="8">Cursor Control
Register</td><td colspan="8">Color Control
Register</td></tr>
<tr><td>4</td><td colspan="32">Reserved</td></tr>
<tr><td>8</td><td colspan="8">Alpha Checksum [7:0]
[read only]</td><td colspan="8">RED Checksum [7:0]
[read only]</td><td colspan="8">GREEN Checksum [7:0]
[read only]</td><td colspan="8">BLUE Checksum[7:0]
[read only]</td></tr>
<tr><td>12</td><td colspan="32">Reserved</td></tr>
<tr><td>16</td><td colspan="4">‘0’</td><td colspan="12">Cursor Position X [11:0]</td><td colspan="4">‘0’</td><td colspan="12">Cursor Position Y[11:0]</td></tr>
<tr><td>20</td><td colspan="4">‘0’</td><td colspan="12">Horizontal Count [11:0]</td><td colspan="4">‘0’</td><td colspan="12">Vertical Count [11:0]</td></tr>
<tr><td>24</td><td colspan="8">‘0’</td><td colspan="8">Cursor Color 0
RED [7:0]</td><td colspan="8">Cursor Color 0
GREEN [7:0]</td><td colspan="8">Cursor Color 0
BLUE [7:0]</td></tr>
<tr><td>28</td><td colspan="8">‘0’</td><td colspan="8">Cursor Color 1
RED [7:0]</td><td colspan="8">Cursor Color 1
GREEN [7:0]</td><td colspan="8">Cursor Color 1
BLUE [7:0]</td></tr>
<tr><td>32</td><td colspan="32">Reserved</td></tr>
<tr><td>256</td><td colspan="32">Cursor Plane 0 Line 0</td></tr>
<tr><td>380</td><td colspan="32">Cursor Plane 0 Line 31</td></tr>
<tr><td>384</td><td colspan="32">Cursor Plane 1 Line 0</td></tr>
<tr><td>508</td><td colspan="32">Cursor Plane 1 Line 31</td></tr>
<tr><td>512</td><td colspan="32">Reserved</td></tr>
<tr><td>1024</td><td colspan="8">‘0’</td><td colspan="24">Color Look Up Table RAM Entry 0
(R[7:0] G[7:0] B[7:0])</td></tr>
<tr><td>2024</td><td colspan="8">‘0’</td><td colspan="24">Color Look Up Table RAM Entry 255
(R[7:0] G[7:0] B[7:0])</td></tr>
</table>

25.9 Control Registers

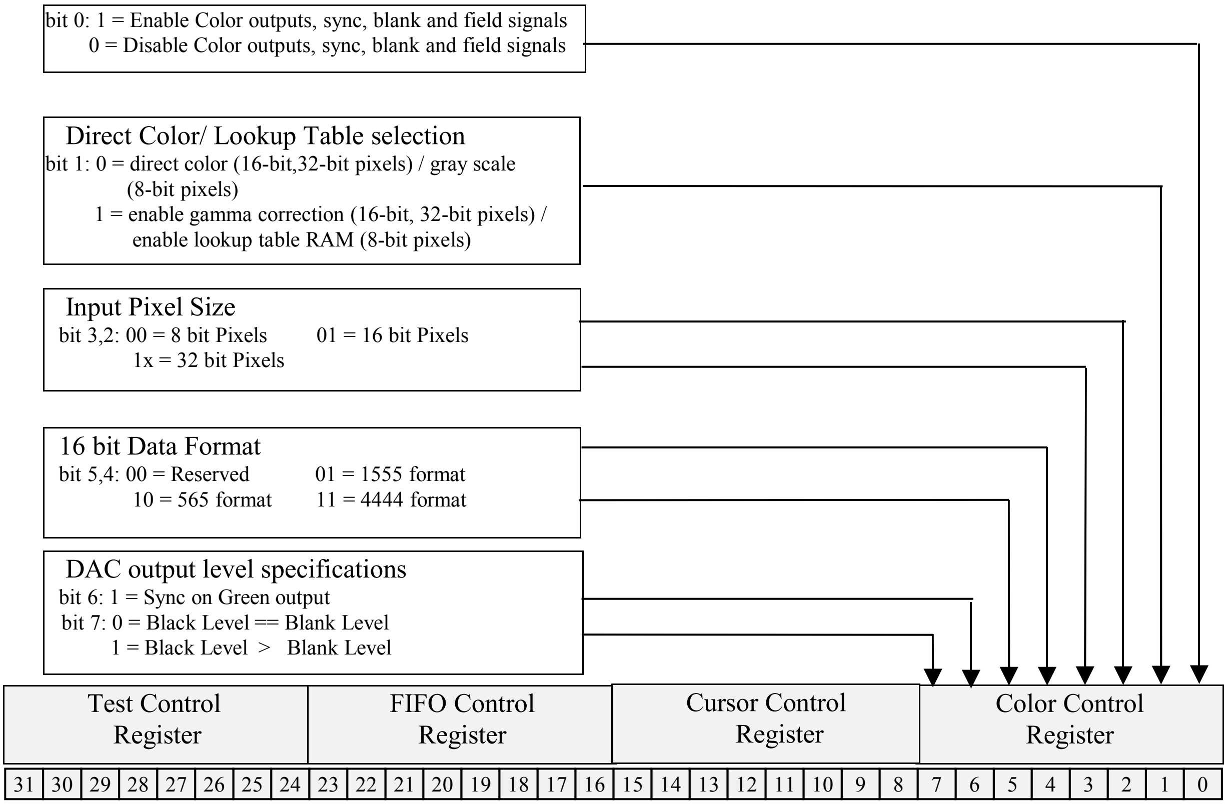
25.9.1 Color Control Registers

This registers controls input data selections and color output signals. Input data types are selected from 16-, 32-bit direct color and 8-bit pseudo color mode. In 16-bit direct color mode, 1555 format, 565 format and 4444 format can be selected.

Bit7:6 controls analog output levels. When bit 6 is set to '1', 442.5 LSBs of current are added to the Green output. When bit 7 is set to '1', 83.5 LSBs of current are added to the outputs.

When Bit 0 is set to '1', color outputs, sync, blank and fields signals is inactive. When IPB Reset is asserted to '1', this bit is set to '1'.

This registers can be write or read from Internal Peripheral Bus. It is not initialized except bit 0.



bit 0: 1 = Enable Color outputs, sync, blank and field signals
0 = Disable Color outputs, sync, blank and field signals

Direct Color/ Lookup Table selection
bit 1: 0 = direct color (16-bit,32-bit pixels) / gray scale (8-bit pixels)
1 = enable gamma correction (16-bit, 32-bit pixels) / enable lookup table RAM (8-bit pixels)

Input Pixel Size
bit 3,2: 00 = 8 bit Pixels 01 = 16 bit Pixels
1x = 32 bit Pixels

16 bit Data Format
bit 5,4: 00 = Reserved 01 = 1555 format
10 = 565 format 11 = 4444 format

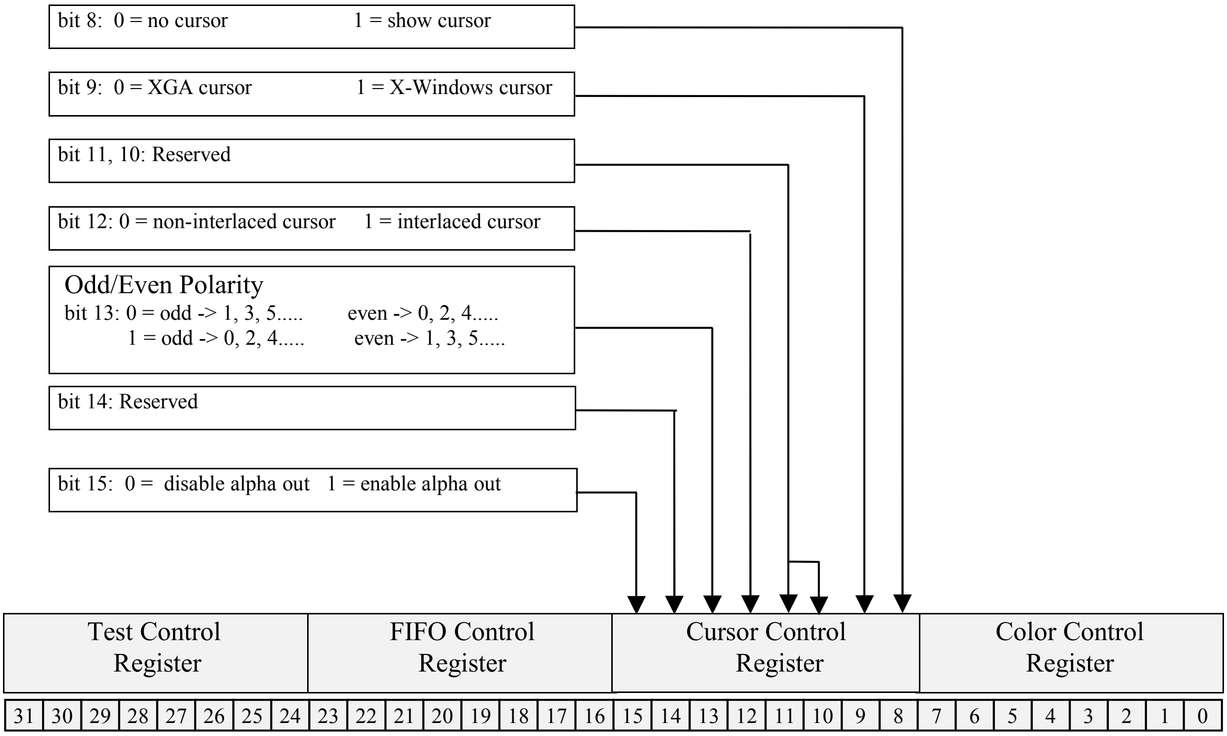
DAC output level specifications
bit 6: 1 = Sync on Green output
bit 7: 0 = Black Level == Blank Level
1 = Black Level > Blank Level

Test Control Register	FIFO Control Register	Cursor Control Register	Color Control Register
31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0

25.9.2 Cursor Control Register

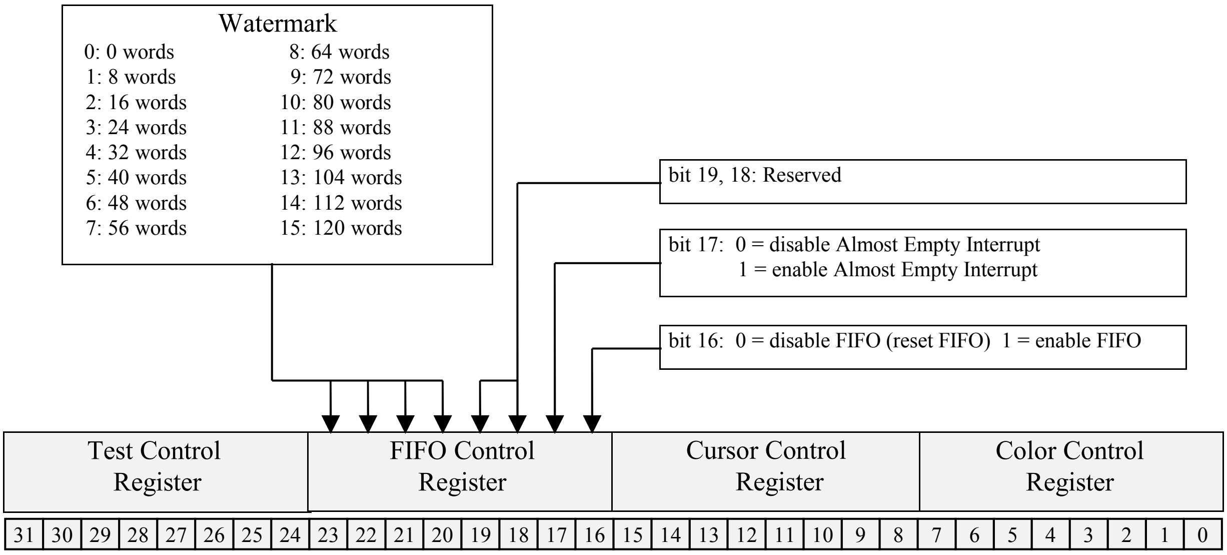
This register controls 32x32 pixel cursor. X-windows and XGA modes are available(bit 9). The cursor operates in both non-interlaced and interlaced modes (bit 12). In non-interlaced mode, bit 7 of this register allows the polarity of Odd/Even_n signal to be inverted when set to ‘1’(It has no influence to Odd/Even_n_Out signal). Bit 8 of this register controls display cursor or not.

This registers is not initialized and can be write or read from Internal Peripheral Bus at any time.


bit 8: 0 = no cursor 1 = show cursor
bit 9: 0 = XGA cursor 1 = X-Windows cursor
bit 11, 10: Reserved
bit 12: 0 = non-interlaced cursor 1 = interlaced cursor
Odd/Even Polarity
bit 13: 0 = odd -> 1, 3, 5..... even -> 0, 2, 4.....
1 = odd -> 0, 2, 4..... even -> 1, 3, 5.....
bit 14: Reserved
bit 15: 0 = disable alpha out 1 = enable alpha out
Test Control Register
FIFO Control Register
Cursor Control Register
Color Control Register
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

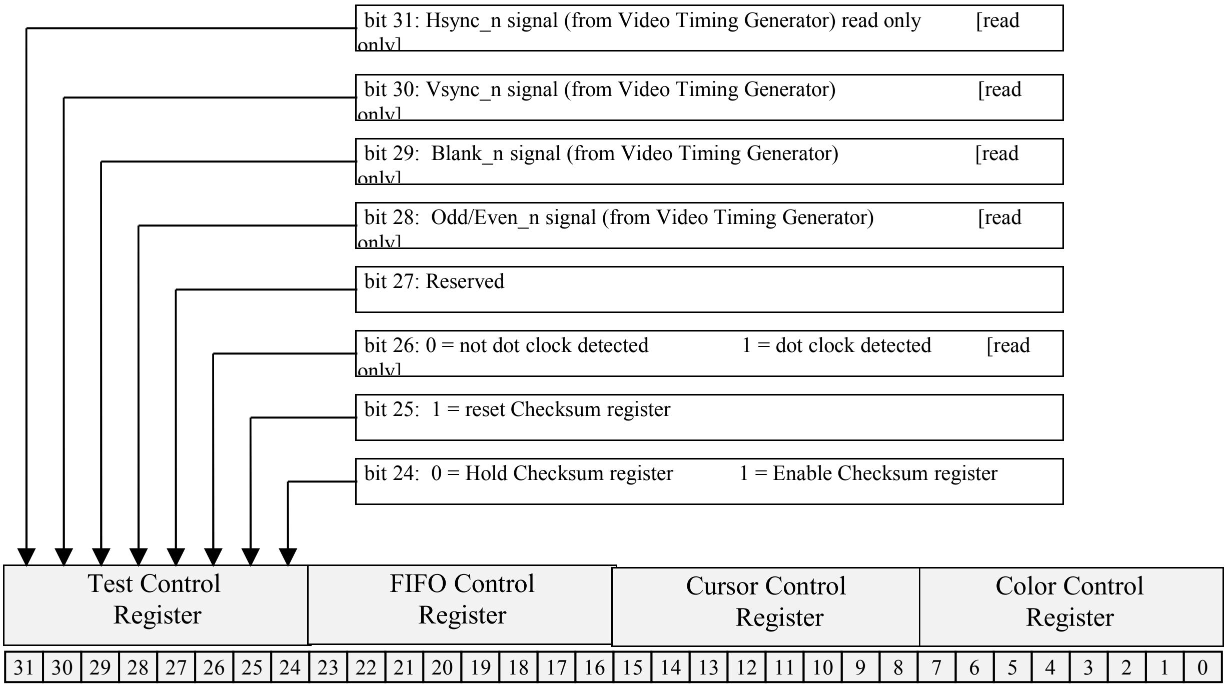
25.9.3 FIFO Control Register

This registers controls the FIFO module. The outputs of this registers are directly connected with the FIFO module. Bit 16 of this register controls the FIFO operation. When set to ‘0’, disable FIFO operation (FIFO read address counter reset to ‘0’). Bit 17 of this register enables FIFO Almost Empty Interrupt. Bit 23..20 is watermark for FIFO Almost Empty Interrupt. When the data stored in FIFO is lower than values of bit 23..20 times 8 words, FIFO Almost Empty Interrupt is generated.
This registers is not initialized and can be write or read from Internal Peripheral Bus at any time.



25.9.4 Test Control Register

This registers is used for test of RAMDAC module. Bit 25, 24 is controls the color out checksum registers operations. When bit 24 is '1', checksum registers are enabled. While enabled, read values of checksum registers are always '0'. When bit 24 is '0', checksum registers are held, these values can be read. When bit 25 set to '1', checksum registers reset to '0'. Bit 25, 24 of this registers is not initialized and can be write or read from Internal Peripheral Bus at any time.
Bit 26 is used for dot clock detection. If set to '0', it means "no dot clock detection".
Bit 31..28 is monitoring registers of timing signals generated by Video Timing Generator. These signals are synchronized to Imagine clock. Bit 31..28 and 26 can be read from Internal Peripheral Bus.


bit 31: Hsync_n signal (from Video Timing Generator) read only [read only]
bit 30: Vsync_n signal (from Video Timing Generator) [read only]
bit 29: Blank_n signal (from Video Timing Generator) [read only]
bit 28: Odd/Even_n signal (from Video Timing Generator) [read only]
bit 27: Reserved
bit 26: 0 = not dot clock detected 1 = dot clock detected [read only]
bit 25: 1 = reset Checksum register
bit 24: 0 = Hold Checksum register 1 = Enable Checksum register
Test Control Register
FIFO Control Register
Cursor Control Register
Color Control Register
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

25.9.5 Test Registers

This registers are checksum of color output data. This registers can be only read from Internal Peripheral Bus at any time. See subsection *2.5 Color Data out*, for more details.

25.9.6 Cursor Position x, y Registers

This registers indicates the location of the cursor top left corner on the display screen. This registers can be write or read from Internal Peripheral Bus at any time. See subsection *2.4.4 Cursor positioning*, for more details.

25.9.7 Count x, Count y Registers

These registers are output of the 12-bit horizontal and vertical counters to display pixels. Horizontal counter count up on every dot clock cycle during Blank_n_In signal is inactive (asserted to 1). When Blank_n_In is 0, horizontal counter reset to 0.

Vertical counter is synchronized by Imagine clock. Vertical counter count up when Blank_n_In signal falls to 0. When Vsync_n_In falls to 0, vertical counter reset to 0. Blank_n_In and Vsync_n_In are synchronized by Imagine clock, then used. The values of these registers can be load from IPB for the purpose of test. When read access from IPB, the values of horizontal counter is connected with IPB via Imagine clock synchronized registers. The outputs of vertical counter is directly connected with IPB.

25.9.8 Cursor Color 0, 1 Registers

These registers store the 24-bit RGB color data for 2 color 32x32 cursor window. This registers can be write or read from Internal Peripheral Bus at any time See subsection *2.4.2 Cursor modes definitions*, for more details.

25.9.9 Cursor Plane 0, 1 entries

These are the entries to the 32x32x2 cursor RAM defines the pixel pattern within the 32x32 pixel cursor window. It is not initialized and may be written or read from Internal Peripheral Bus (offset 0x100 to 0x1fc) at any time. See subsection *2.4.3 Cursor RAM*, for more details.

25.9.10 Color Look-up Table RAM entries

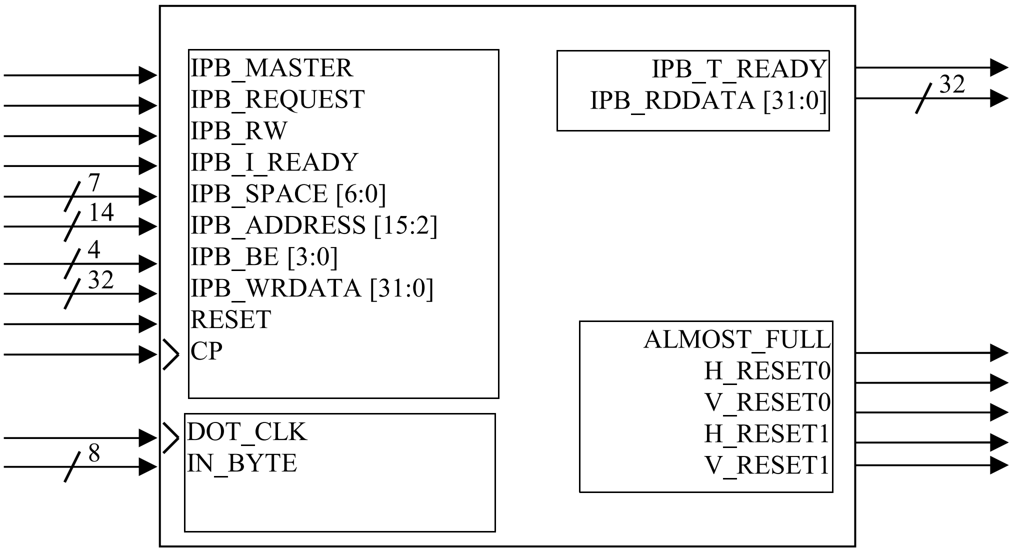
These are the entries to the 256x24 bit color look-up table RAM for pseudo color mode. Color look-up table RAM is not initialized and may be written or read from Internal Peripheral Bus (offset 0x400 to 0x7fc) at any time. See subsection *2.3 Read Look-up Table RAM*, for more details.

Chapter

26. VIDEO INPUT UNIT

The Video Input Unit can load digital video data from the 8 bit video input bus. This bus xecute timing instructions from their own Timing instruction RAM and are capable op generating arbitrary video timing signals up to a resolution of 4096 by 4096 pixels, including CCIR601, NTSC and PAL-M formats. The Video output timing generator sends its timing signals to the video output unit. Both video input and output timing generators can be independently synchronised to the CCIR 656 video input or to the external Vreset pin. Both can independently choose between the clock from the internal dot clock generator or the CCIR 656 video input clock.*

26.1 The Input/Output Signals of the Video Input Unit



26.1.1 Signal definitions

IPB_MASTER IPB_REQUEST IPB_I_READY IPB_SPACE [6:0] IPB_ADDRESS [15:2] IPB_BE [3:0] IPB_WRDATA [31:0] RESET CP IPB_T_READY IPB_RDDATA	input input input input input input input input input output output	See : “The protocol of the Internal Peripheral Bus”
DOT_CLK	input	External DOT clock. (May not be present)
IN_BYTE	input	8-bit digital input data. (Usually YUV data as per CCIR recomendation)
ALMOST_FULL	output	FIFO almost full flag. Signals Imagine that a FIFO read is required to avoid data loss. The FIFO should never be allowed to get completely full and it cannot stall the input.
H_RESET0	output	Horizontal reset pulse for Video Timing Generator 0
V_RESET0	output	Vertical reset pulse for Video Timing Generator 0
H_RESET1	output	Horizontal reset pulse for Video Timing Generator 1
V_RESET1	output	Vertical reset pulse for Video Timing Generator 1

26.2 *Module overview of the Video Input Unit (VIN)*

The Video Input Unit consists of six modules.

1. IPB-interface.
2. Stage0 (Input stage).
3. Stage1 (optional 4:2:2 to Alpha:4:4:4 conversion with possible downsampling).
4. Stage2 (optional downsampling).
5. Stage3 (optional colour conversion YUV -> RGB).
6. FIFO.

26.2.1 The IPB_interface

The IPB-interface connects the VIN to the Internal Peripheral Bus. It decodes the IPB request and determines the appropriate action for the request. All accesses through the IPB to this unit require multiple cycles, except reading from the FIFO which is completed within a single cycle. This allows a fast burst read when the FIFO is almost full. This unit also contains the control registers for the rest of the unit.

26.2.2 Stage0 (Input stage)

The input stage is the most complex of all the stages. It has to detect global synchronisation of the input data stream, proper word grouping (4-bytes), horizontal and vertical sync signals, line and/or field skipping, synchronisation errors etc. This stage basically determines which data to pass to the next stage. Once the input stage outputs data to the next stage, the data passes through the other stages and into the FIFO.

26.2.3 Stage1 (4:2:2 to A:4:4:4 conversion)

This stage optionally converts a 4:2:2 input word into a A:4:4:4 value with A taken from a programmable register. This stage has three modes:

0 : pass through	(X3, X2, X1, X0)	→	(X3, X2, X1, X0)
1 : 4:2:2 to A:4:4:4 with downsampling	(Y1, V0, Y0, U0)	→	(A, V0, (Y1+Y0)/2, U0)
2 : 4:2:2 to A:4:4:4	(Y1, V0, Y0, U0)	→	(A, V0, Y0, U0) (A, V0, Y1, U0)
3 : RESERVED			

Mode 1 achieves a 2:1 horizontal downsampling of the input data. With mode 2 a single input word creates two output words which effectively upsamples the input data. When digital line blank data or field blank data is received, then no conversion is performed, irrespective of the mode.

26.3.4 Stage2 (Down sampling)

This stages does optional 2:1 downsampling of the input. It has four modes of operation.

0 : pass through	(X3, X2, X1, X0)	→	(X3, X2, X1, X0)
1 : A:4:4:4 downsampling	(A0, V0,Y0, U0),(A1, V1,Y1, U1)	→	((A1+A0)/2, (V1+V0)/2, (Y1+Y0)/2, (U1+U0)/2)
2 : 4:2:2 downsampling	(Y1, V0, Y0, U0) (Y3, V1, Y2, U1)	→	((Y3+Y2)/2, (V1+V0)/2, (Y1+Y0)/2, (U1+U0)/2)
3 : 8-bit downsampling	(X7, X6, X5, X4) (X3, X2, X1, X0)	→	((X7+X6)/2, (X5+X4)/2, (X3+X2)/2, (X1+X0)/2)

Mode 1 together with mode 1 of stage 1 achieves a 4:1 horizontal downsampling of the input data stream. As with the previous stage no transformation is performed on digital blanking data. Note that modes 1 to 3 require two input words from the previous stage before generating an output. If a pre-amble is detected while this stage is still waiting for the second input word, the saved word is discarded for this would indicate an error condition.

26.3.5 Stage3 (Colour conversion)

This stage does the optional A:4:4:4 to ARGB conversion. The coefficients in the conversion matrix are programmable to allow for fine tuning of individual requirements and for various other applications. It has two modes.

0 : pass through	(X3, X2, X1, X0)	→	(X3, X2, X1, X0)
1 : A:4:4:4 to ARGB	(A, V, Y, U)	→	(A, R, G, B)

The standard YUV to RGB conversion matrix for CCIR656 data is as follows:

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} 1 & 0 & 1.371 \\ 1 & -0.336 & -0.698 \\ 1 & 1.732 & 0 \end{vmatrix} \begin{vmatrix} Y \\ U \\ V \end{vmatrix}$$

The coefficients are however programmable via control registers 4, 5 and 6. The coefficients have a sign bit and an eight bit value in the 1.7 format (1 bit before the decimal comma and 7 after). They can therefore vary from -1.9921875 to +1.9921875 (+- 1.1111111). This allows for fine tuning the colour conversion matrix.

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} \begin{vmatrix} Y \\ U \\ V \end{vmatrix}$$

26.3.6 FIFO (128 deep by 32-bit wide)

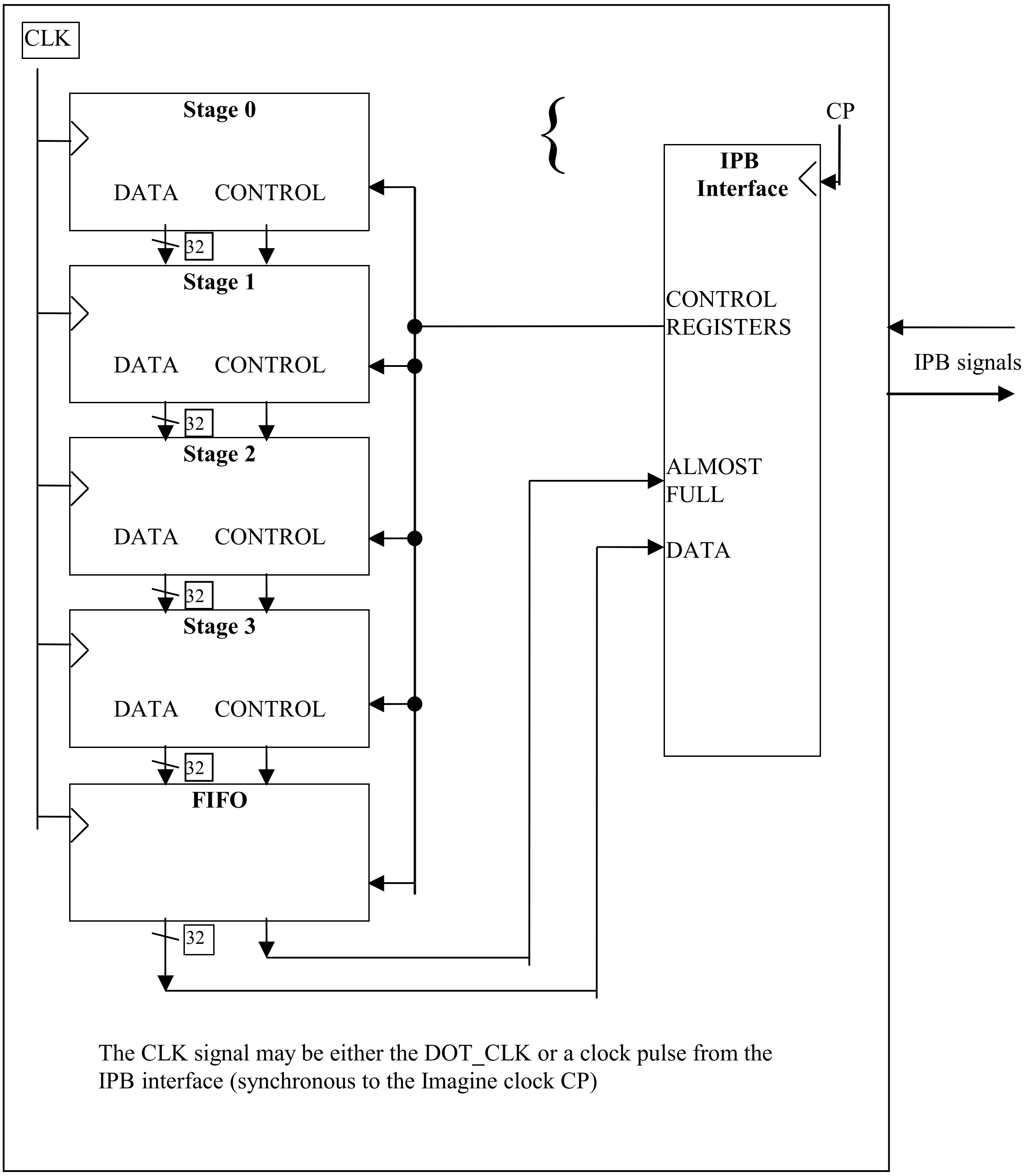
The FIFO stores the data coming from stage 3. It has a programmable watermark to determine the ALMOST FULL state. As it is not possible to stall the input data stream, it should NEVER be allowed to become completely full. The almost full flag generates an interrupt to the processor to indicate that data is available.

26.4 Functional description of the Video Input Unit

The VIN is mainly intended for accepting digital input data as per the CCIR 656 recommendation. All the programmable options do however allow for other user defined applications. It is for instance possible to accept any incoming data and pass it through to the processor where any software manipulation can be performed. The VIN contains downsampling and colour conversion circuitry to reduce the burden on the processor, especially for applications which do not require the full resolution.

The operation of the VIN will now be described with an assumed CCIR656 input data stream.

Block diagram of the Video Input Unit



The CLK signal may be either the DOT_CLK or a clock pulse from the IPB interface (synchronous to the Imagine clock CP)

26.4.1 Stage0 (Input stage)

This stage is the most complex of the four stages. It's main function is to determine what data to pass into the rest of the pipeline. After a reset the first task this unit has to perform is to synchronise on the incoming data stream (if synchronisation is enabled). After a reset the incoming data could be at an arbitrary position in the video frame. In order to determine synchronisation, the unit uses the bits 16 to 23 in control register 2 to detect a valid FVH combination, which forms part or the byte after a received pre-amble (the pre-amble is defined as hex FF 00 00 in CCIR656).

The exact method used to determine a valid FVH combination needs some elaboration as it is also used in detecting the Horizontal and Vertical reset pulses from the input data stream.

- Firstly there are the three compare values for F, V and H. (Refer to control register 2 bit definition) These bits determine what the values of F, V and H should be in the input for a valid detection. They only apply however if the corresponding enable bit is logic '1'.
- The three enable bits determine which of the three values (F, V and H) should be used in the comparison. If the enable bit is logic '0', the corresponding value is NOT used in the comparison and the compare value is then a don't care.
- The transition detect is used to determine a transition in the state of F and/or V. This means that not only should the current F and/or V value be correct, but it/they should have changed from the previously received F and V. This allows for an unambiguous placement of the selected signal be it the global sync, horizontal or vertical reset pulse. There is no transition check for H because according to the CCIR specification, the H values changes with every received pre-amble.

To clarify the method, a few examples will be given.

- The required values for a "normal" horizontal reset pulse at the end of the active video line (start of H_sync) would be FVH_enable=(001), FVH_compare=(xx1), FV_trans=(00). Only the H value is used in the comparison and no transition detection is required. The pulse will therefore be generated every time the received H is a logic '1'. If the H compare value is set to '0', the pulse will be generated at the start of the active video (end of H_sync).
- The required values for a vertical reset pulse would be FVH_enable=(010), FVH_compare=(x1x) and FV_trans=(01). Only the V value is used and it should change TO a '1' to create the pulse. This will create two reset pulses at the start of the V_sync during the first and second fields (interlaced). If it is required only for the second field, then the values should be FVH_enable=(110), FVH_compare=(11x) and FV_trans=(01).
- For global synchronisation a logical position would be at the start of the first field. The required values for this are FVH_enable=(100), FVH_compare=(0xx) and FV_trans=(10).

It is important to note that various other values could be used, but they would not be useful with CCIR656 input data. For instance checking for a transition in BOTH F and V would not work because in CCIR656 data these two values never have transitions SIMULTANEOUSLY. This might however not be applicable in other applications.

The vertical reset pulse is also used to reset the line counter within stage 0. This counter holds the current input line number and is increased by the horizontal reset pulse. Because these pulses are however programmable, the line number does not correspond to the same line number as defined in the CCIR656 specification. This line number is further used to ignore certain lines (if required. See bits 0-7 of control reg 3). Ignoring lines allows for elementary vertical downsampling of the input stream.

In general a single frame consists of three distinct data areas.

- Normal line data (YUV pixel data in groups of four received as Cb, Y0, Cr, Y1)
- Digital line blank data (H=1)
- Digital field blank data (V=1)

By setting the required bits (bits0-11 of control register 3) the application program can determine exactly what data to send to the next stage. Note that the exact line numbers do not correspond to the CCIR656 specification, but depend on the user placement of the horizontal and vertical synchronisation pulses. When accepting pre-ambles, the EAV (end of active video) is part of the normal line data and the SAV (start of active video) is part of the line blank data.

26.5 The control registers

The video input unit has six control registers. The first control register is mainly a status register and is intended for testing. The other five control registers determine all the modes of operation for the different stages, the FIFO watermark etc. The complete bit definitions are as follows

Control Register 1

0	RESET
1	HOLD
2	FIFO empty
3	FIFO almost full
4	Input Synchronized
5	Synchronization error
6	H_reset
7	V_reset
19:8	Line counter
20	Stage 0 data valid
21	Stage 1 data valid
22	Stage 2 data valid
23	Stage 3 data valid
31:24	Input byte

Control Register 2

	Horizontal reset pulse control
0,1,2	H,V,F compare value
3,4,5	H,V,F compare enable
6,7	V,F transition check enable
	Vertical reset pulse control
8,9,10	H,V,F compare value
11,12,13	H,V,F compare enable
14,15	V,F transition check enable
	Start synchronisation control
16,17,18	H,V,F compare value
19,20,21	H,V,F compare enable
22,23	V,F transition check enable
	General control
24	Enable Horizontal reset 0
25	Enable Vertical reset 0
26	Enable Horizontal reset 1
27	Enable Vertical reset 1
28	Enable start synchronisation
29	Enable re-sync on error
30	Enable Hamming error correction
31	Reserved

Control Register 3

0	Enable sampling in even field lines xx00
1	Enable sampling in even field lines xx01
2	Enable sampling in even field lines xx10
3	Enable sampling in even field lines xx11
4	Enable sampling in odd field lines xx00
5	Enable sampling in odd field lines xx01
6	Enable sampling in odd field lines xx10
7	Enable sampling in odd field lines xx11
8	Accept field blank data
9	Accept line blank data
10	Accept normal data
11	Accept pre-ambles
13:12	Mode stage 1
15:14	Mode stage 2
16	Mode stage 3
24:17	Alpha
31:25	Watermark

Control Register 4

31:24	a
23:16	b
15:8	c
7:0	d

Control Register 5

31:24	e
23:16	f
15:8	g
7:0	h

Control Register 6

31:24	i
23:9	Reserved
8:0	9 sign bits (a-i) 0=poitive 1=negative

26.6 Interfacing with the Video Input Unit through the IPB

All read and write accesses to and from the Video Input Unit require multiple cycles, except reading from the FIFO which completes within one cycle. There are seven different accesses depending on the selected address (IPB_ADDRESS).

1. FIFO address (VIN_BASE + 0)
2. Control register 1 (VIN_BASE + 1)
3. Control register 2 (VIN_BASE + 2)
4. Control register 3 (VIN_BASE + 3)
5. Control register 4 (VIN_BASE + 4)
6. Control register 5 (VIN_BASE + 5)
7. Control register 6 (VIN_BASE + 6)

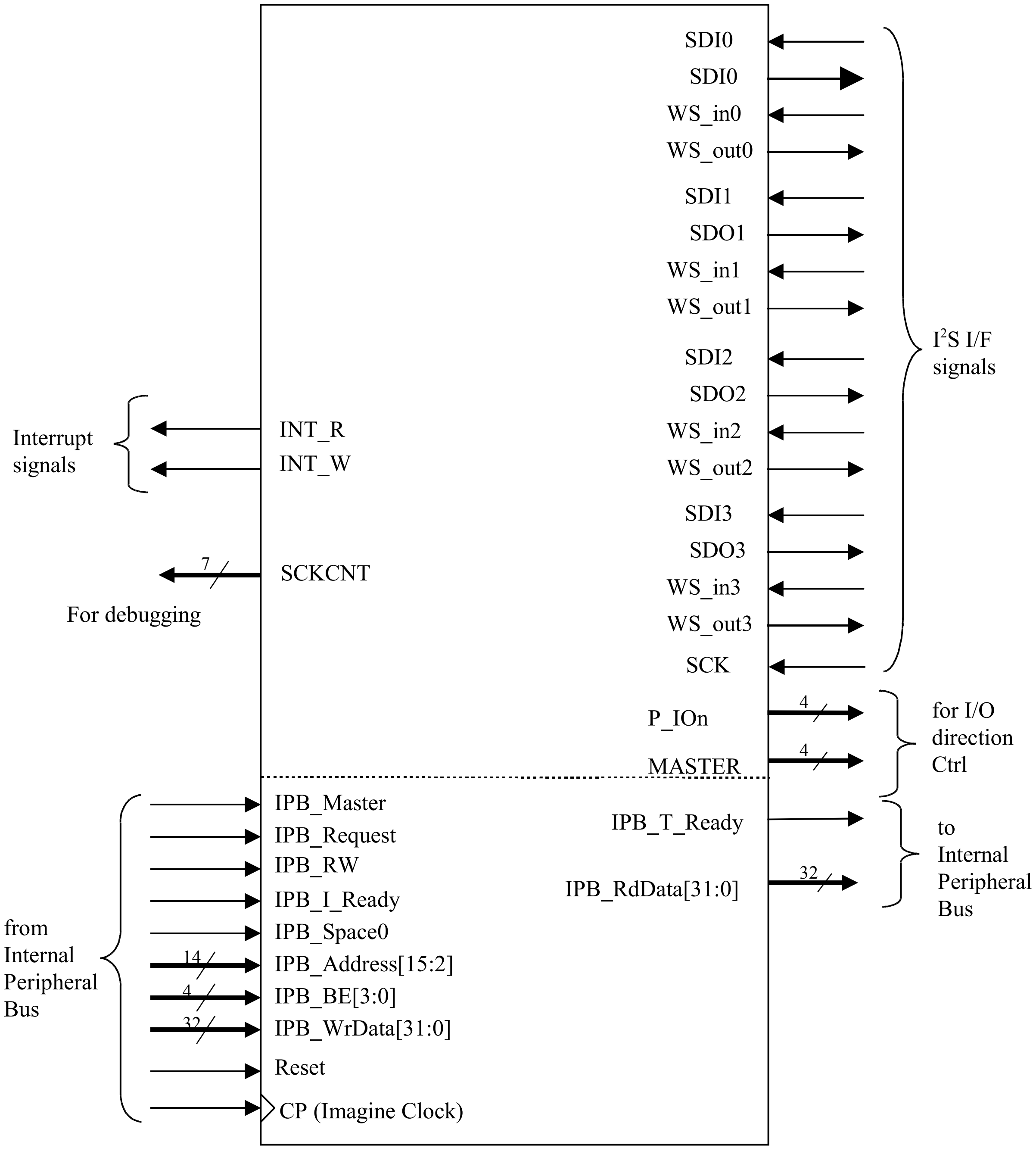
I. The FIFO address can only be read and completes within a single cycle. This allows the FIFO to be emptied with a burst read when it signals that it is almost full. Control register 1 is a status register and is intended mainly for testing purposes. Most of the bits are read only. If the unit is in the hold mode, the dot clock is replaced by a test pulse to facilitate testing. Every WRITE to control register 1 generates another test pulse. During hold mode the top byte (bits 31:24) of control register 1 is also used as the input byte instead of the actual byte received. Hereby a program can simulate any possible input byte stream in a controlled manner for testing purposes.

Chapter

29. THE I^{2}S AUDIO INTERFACE

The I^{2}S audio interface has four different serial audio channels which can be individually programmed for input and output. Almost all serial formats are supported including the Sony S format. The polarity of the Left / Right indicator is programmable as well as the number of bits per audio sample. An on chip RAM of 128 words of 32 bit can be used as a fifo(s) for one to four channels with progammable fifo size. The fifo(s) can generate interrupts to the Imagine core processor based on programmable watermarks.

29.1 The Input/Output Signals of I^{2}S Interface Unit



29.1.1 Input/ Output signals definitions

I^2S Bus Interface signals		
SCK	input	Serial data clock for every I^2S Port
SDI0	input	Serial two time-multiplexed data input for I^2S Port 0. Input data is synchronized with SCK. When P_Ion[0] is set to 1, SDI0 is used as I^2S input data port.
SDO1	output	Serial two time-multiplexed data output for I^2S Port 0. When P_Ion[0] is set to 0, SDO0 is used as I^2S output data port. The data is synchronized with CP (Imagine clock).
WS_in0	input	Word Select signal for I^2S Port generated by external I^2S master device. When MASTER is logic 0, it is used as word select signal. It is synchronized with SCK. If JPMODE[0] (Main Control register, bit 12) is set to 0 (I^2S input format), “WS_in0 = 0” indicates the left channel data and “WS_in0 = 1” indicates the right channel data. In case of “JPMODE[0] = 1” (Japanese input format), “WS_in0 = 0” indicates the right channel data and “WS_in0 = 1” indicates the left channel data.
WS_out0	output	Word select signal for I^2S Port 0. When MASTER[0] is set to 1, it is used as word select signal. This signal is generated by Serial Timing Generator in this unit. This signal is synchronized with CP (Imagine clock).
SDI1..3	input	Serial two time-multiplexed data input for I^2S Port 1..3. (same as SDI0).
SDO1..3	output	serial two time-multiplexed data output for I^2S Port 1..3 (same as SDO0).
WS_in1..3	input	Word Select signal input for I^2S Port 1..3 (same as WS_in0).
WS_out1..3	input	Word Select signal ouput for I^2S Port 1..3 (same as WS_out0).
I^2S Bus Interface signals (bidirectional control for I/O buffers)		
P_Ion[3:0]	output	This is a register output (Main Control register, bit23..20) and can be written or read from Internal Peripheral Bus. These signals are used as the DIR control signal of bidirectional SDI/SDO I/O buffer.
MASTER [3:0]	output	This is a register output (Main Control register, bit 11:8) and can be written or read from Internal Peripheral Bus. These signals are used as the DIR control signal of bidirectional MASTER I/O buffer.
Interrupt signals (to Interrupt Vector Generator)		
INT_R	output	Read interrupt flag.
INT_W	output	Write interrupt flag.

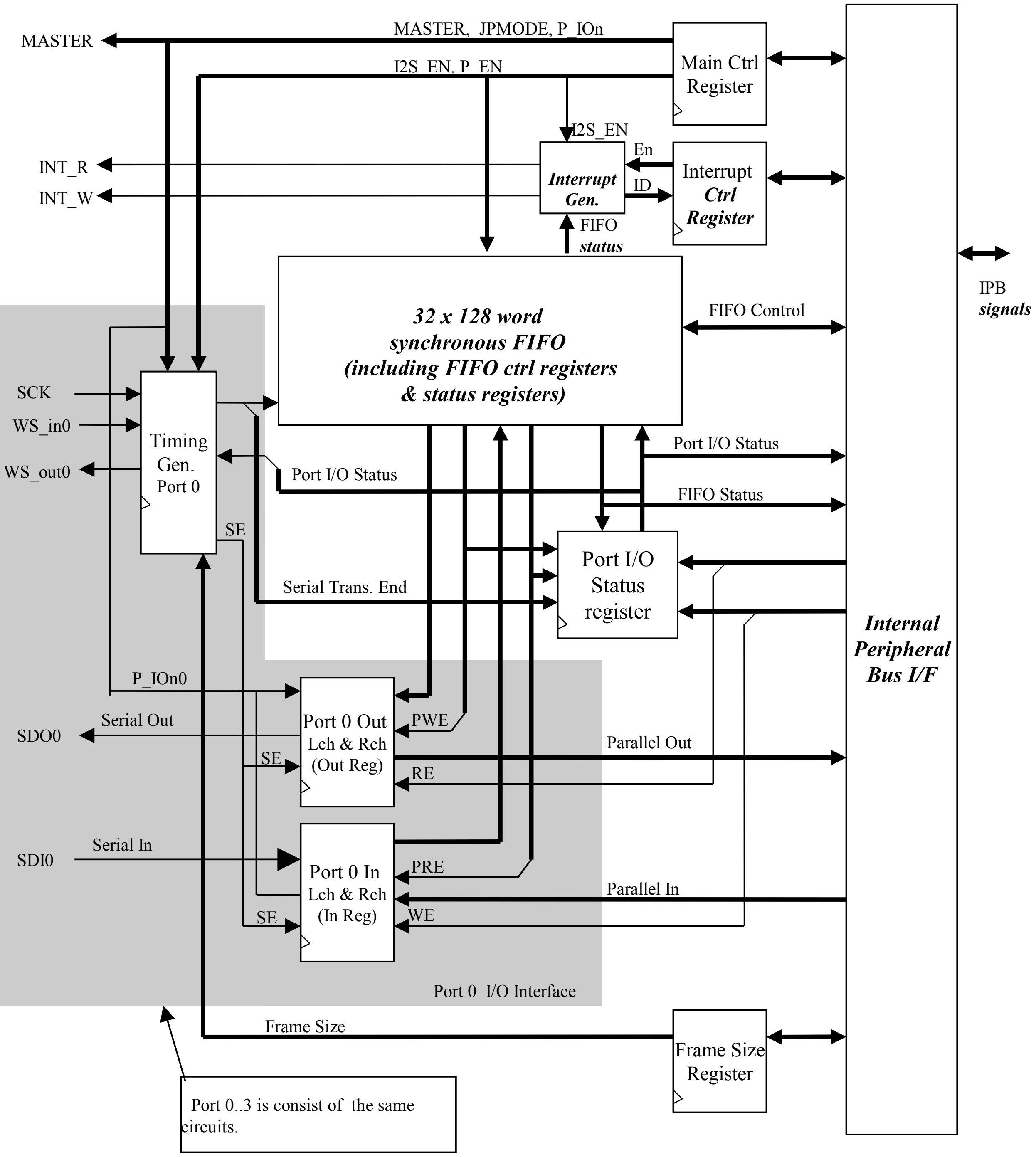
Internal Peripheral Bus Interface		
CP (Imagine Clock)	input	See document *“The Protocol of the INTERNAL PERIPHERAL BUS, revision 0.9a”*
Reset	input	
IPB_Master	input	
IPB_Request	input	
IPB_RW	input	
IPB_T_Ready	output	
IPB_I_Ready	input	
IPB_Space0	input	
IPB_Address[15:2]	input	
IPB_BE [3:0]	input	
IPB_RdData [31:0]	output	
IPB_WrData [31:0]	input	

29.2 I^{2}S Bus Interface Unit overview

I^{2}S Bus Interface Unit has three interfaces, four port I^{2}S Interfaces, Internal Peripheral Bus, and two interrupt output signals. Four port I^{2}S (inter-IC sound) bus interfaces are used for communication with the external digital audio devices. Each ports can be used as input port or output port. These ports are independent (except SCK) and can be set to master or slave. This interface is based on *"I^{2}S bus specification"* and also supports Japanese(SONY) input/output format.

Internal Peripheral Bus is connected with some internal control registers and I/O registers. These registers are on IPB_Space0, 64 bytes address area. Internal Peripheral Bus communicates with I^{2}S input or output ports via internal 128 word x 32 bit FIFO. This FIFO is separated to four areas, and each areas have a independent I/O registers and Read/Write pointers.

The two interrupt lines (INT_R, INT_W) are connected to the Interrupt Vector Generator via the Interrupt Router. When input or output FIFO almost full/empty occurs, this module generates interrupt pulse. All flip-flops and FIFO are synchronized with Imagine Clock.



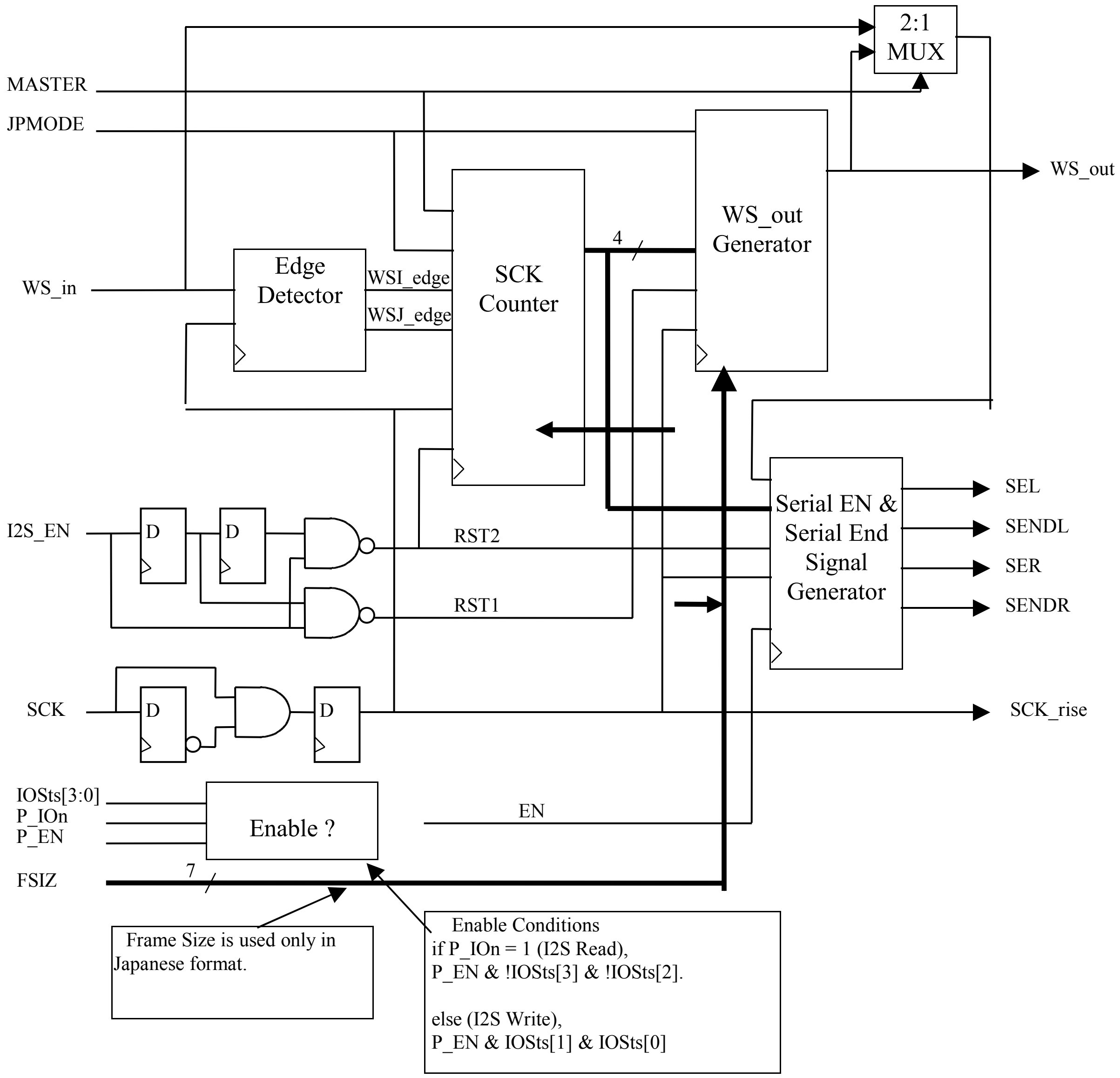
29.3 Serial Timing Generator (I2S_TGEN.v)

This block generates serial word select signal (WS) and serial data enable (SE), and serial transfer end signal (SEND). This block samples SCK by Imagine Clock (CP), detects leading edge of SCK. The SCK counter (SCKCNT) counts up the leading edge of SCK, generates WS signal (If MASTER is asserted with 1, this block uses this signal for WS output signal) and serial input/output timing.

This block also generates I^{2}S serial data write or read enables. The read enables are used for separating each audio data from multiple serial input (SDATA_IN). The separated audio data are store to 32-bit FIFO input registers. The write enables are used for making of multiple serial output (SDATA_OUT) from 32-bit FIFO output registers. All enables are synchronized with Imagine Clock and based on the value of SCKCNT.

I^{2}S Bus Interface Unit has four Timing Generators for each I^{2}S I/O Ports and separately programmable.

29.3.1 Block Diagram (for I^{2}S I/O Port 0)

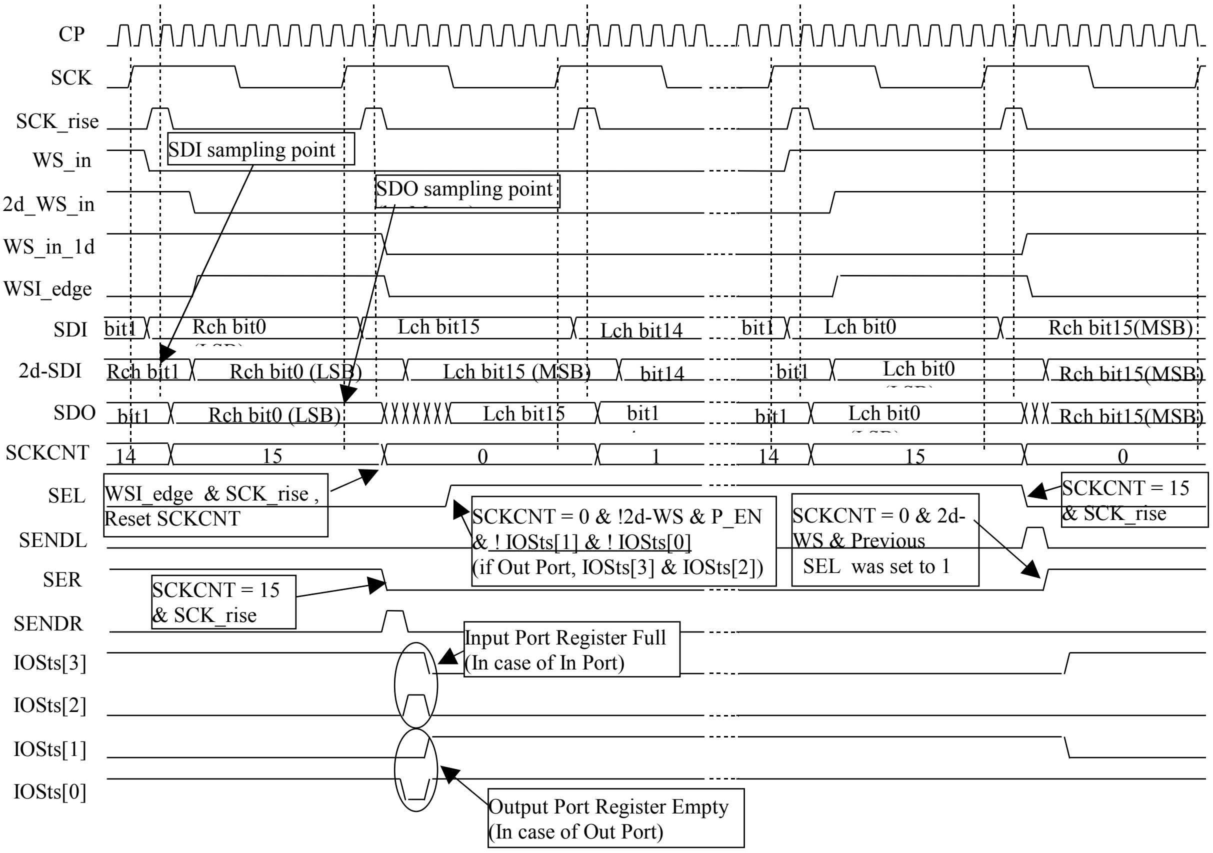


Frame Size is used only in Japanese format.

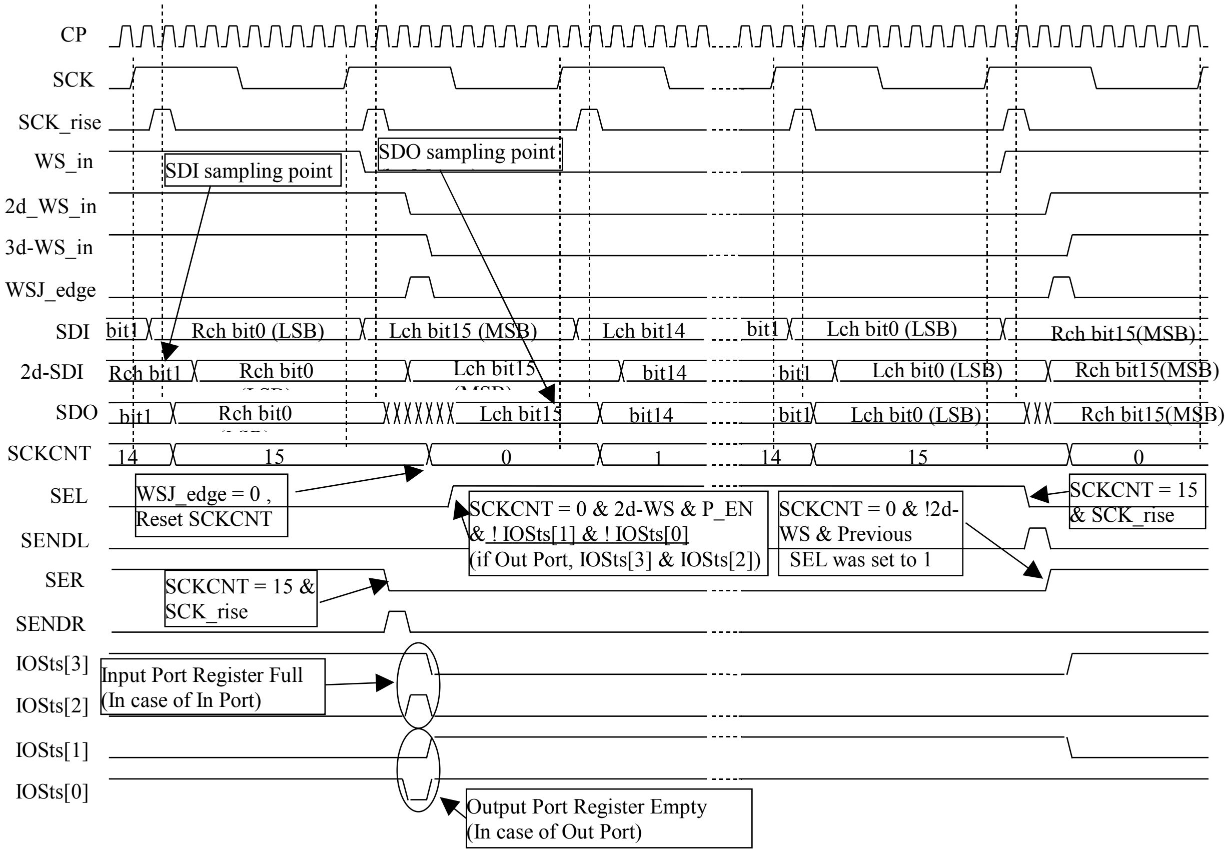
Enable Conditions
if P_IOn = 1 (I2S Read),
P_EN & !IOSts[3] & !IOSts[2].

else (I2S Write),
P_EN & IOSts[1] & IOSts[0]

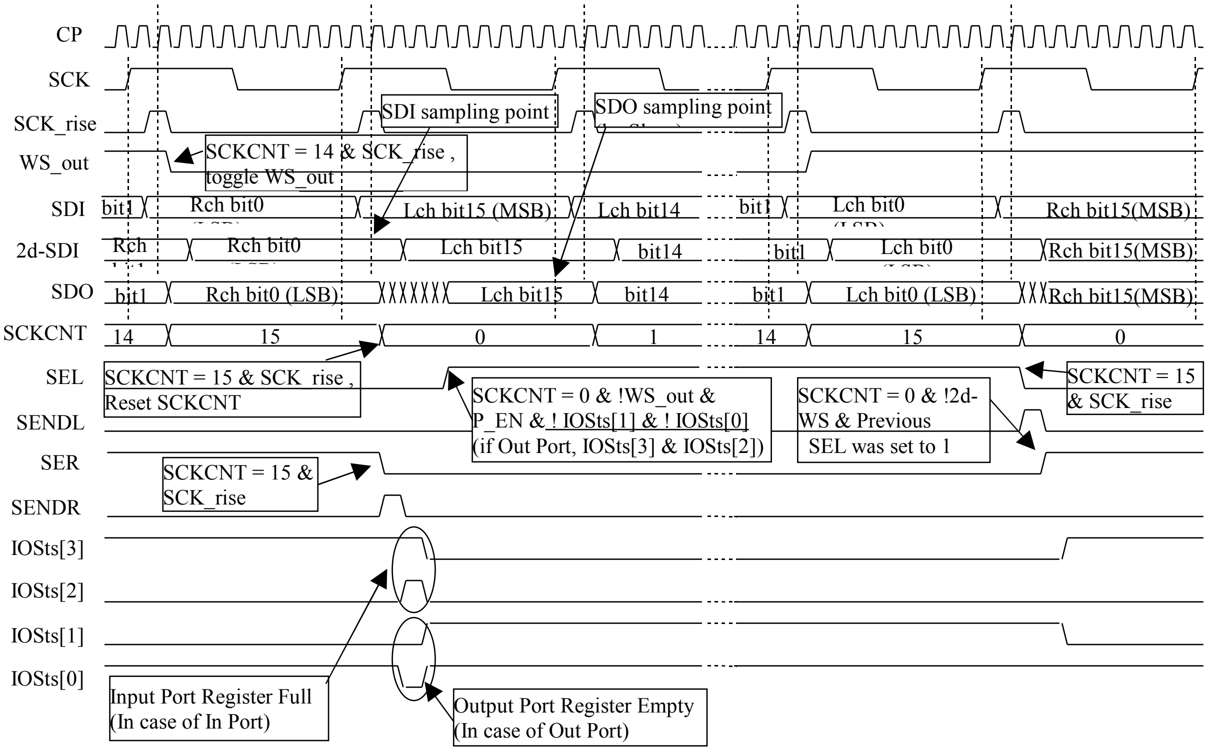
29.3.2 Serial Timing (Slave, I^{2}S format)



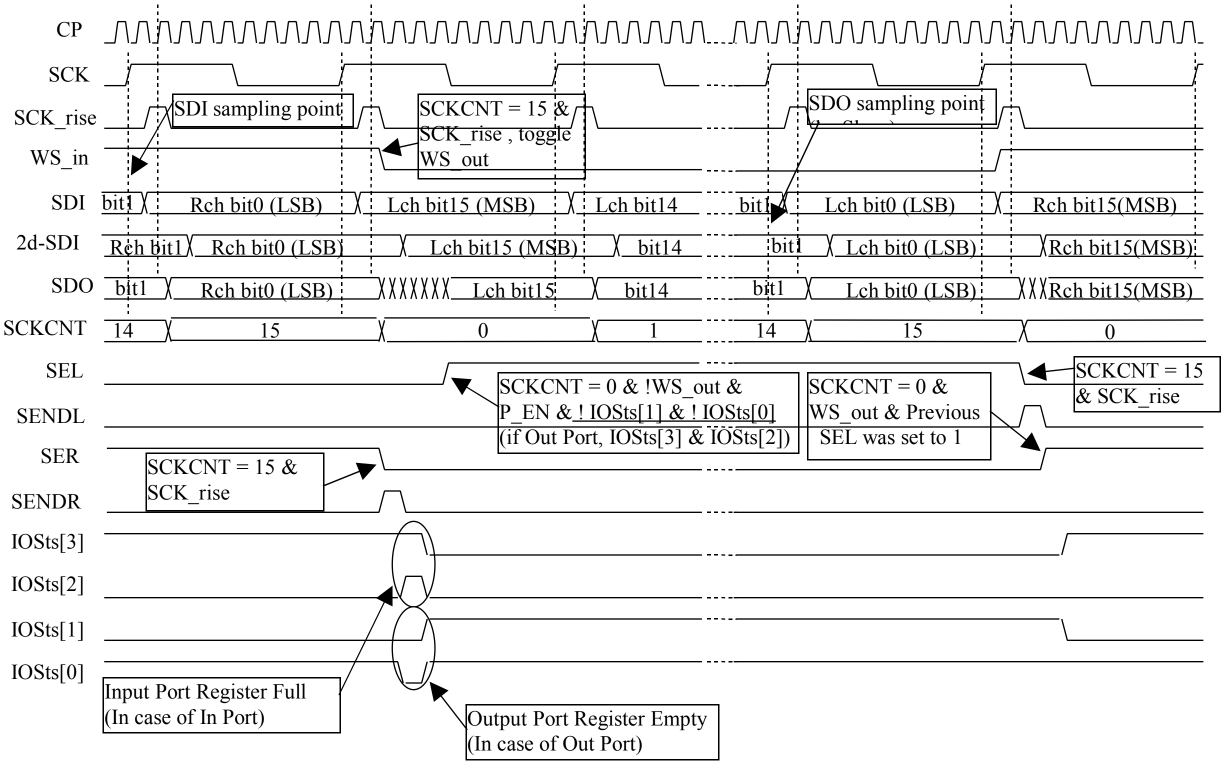
29.3.3 Serial Timing (Slave, Japanese format)



29.3.4 Serial Timing (Master, I^{2}S format)



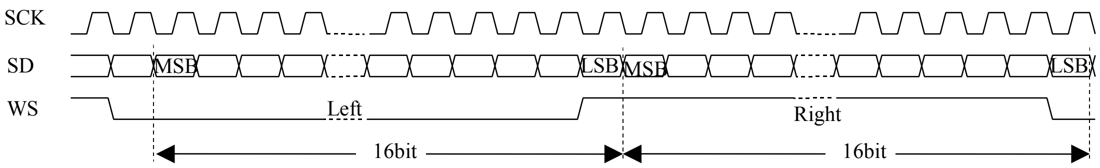
29.3.5 Serial Timing (Master, Japanese format)



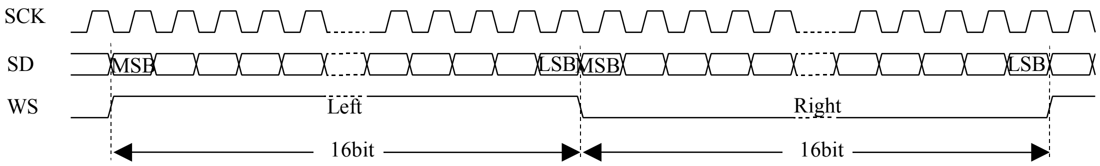
29.3.6 Serial Data Format

This unit supports both I^{2}S format and Japanese format. The Word length is fixed to dual 16-bit. In slave mode, when the incoming serial data length is grater than 16-bit, the trailing invalid bits are ignored. In case of output ports, the position of invalid bits are filled with 0’s.

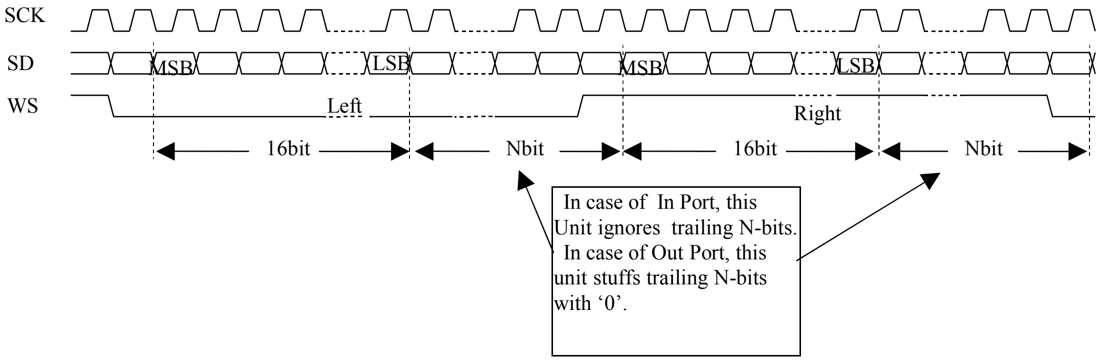
I^{2}S format (Data Length 16-bit)



Japanese format (Data Length 16-bit)

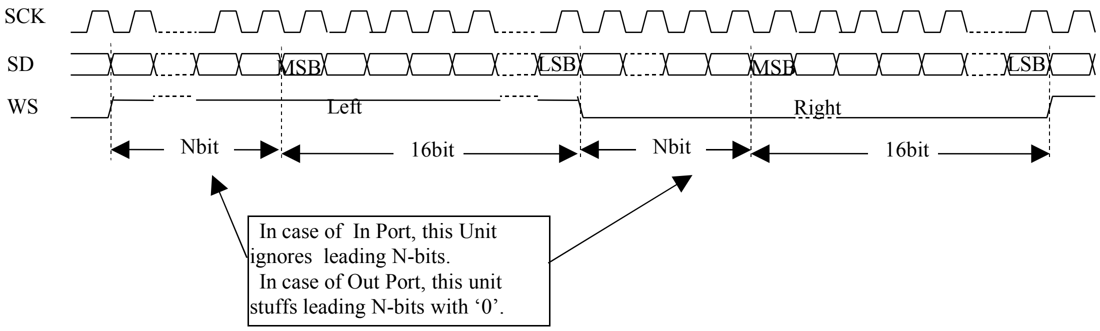


I^{2}S format (Data Length > 16-bit)



In case of In Port, this Unit ignores trailing N-bits.
In case of Out Port, this unit stuffs trailing N-bits with ‘0’.

Japanese format (Data Length > 16-bit)



In case of In Port, this Unit ignores leading N-bits.
In case of Out Port, this unit stuffs leading N-bits with ‘0’.

29.4 FIFO Input/Output Registers (I2S_FR.v)

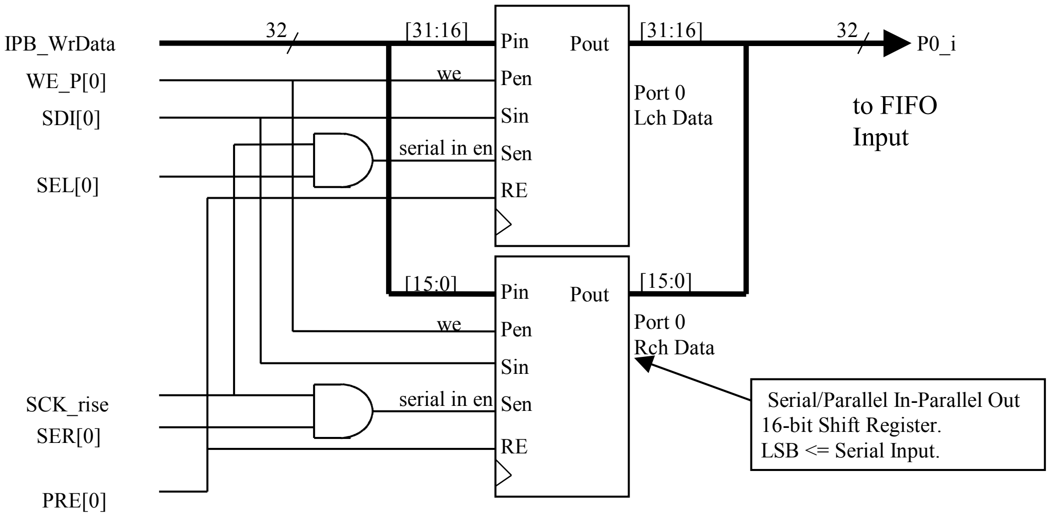
This block has eight 32-bit registers. Internal Peripheral Bus communicates with I^{2}S ports by using these registers. It is placed between Internal Peripheral Bus and FIFO or I^{2}S ports and FIFO.

FIFO Input Registers are used in two ways. If the I^{2}S port is set to output, Internal Peripheral Bus writes data to this register. The data must be written to the register, 32-bit simultaneously. Byte write is not supported. If FIFO is not full and this register contains 32-bit data, the data is written to FIFO. On the other hand, if the I^{2}S port is set to input, it's used for I^{2}S port input registers. This registers is 32-bit shift register with serial input and parallel outputs. Serial input is connected with serial data in of I^{2}S Input Port (SDI). After the serial transfer, if FIFO is not full, the data is written to FIFO, 32-bit simultaneously.

FIFO Output Registers are also used for two ways. One is FIFO to IPB registers. Internal Peripheral Bus read data from this register, 32-bit simultaneously. After the read by Internal Peripheral Bus, if FIFO is not empty, FIFO writes 32-bit data to the register. Another one is FIFO to I^{2}S output ports registers. This registers is 32-bit shift register with 32-bit parallel inputs and a serial output. Serial output is connected to serial output of I^{2}S output ports (SDO). After the serial data transfer, if FIFO is not empty, the FIFO writes 32-bit data to register using parallel inputs.

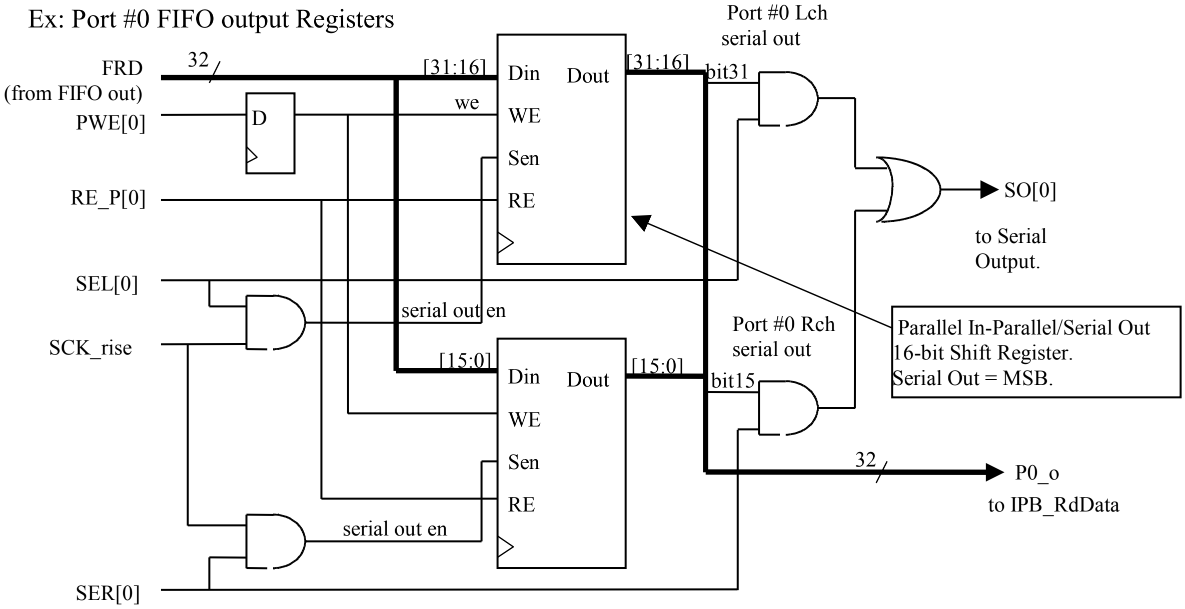
29.4.1 Block Diagram (FIFO Input Registers)

Ex: Port #0 FIFO input Registers



29.4.2 Block Diagram (FIFO Output Registers)

Ex: Port #0 FIFO output Registers



29.5 128 x 32 bit FIFO (F_I2S.v)

This block has 128 x 32 bit user configurable FIFO. The FIFO has one write port and one read port, independently. The FIFO is divided to four parts for I^{2}S Port 0..3. Each part has individually FIFO start address, end address, watermark, write pointer, and read pointer. Start address, end address, and watermark are able to be written or read from Internal Peripheral Bus. These values are stored to registers. The write pointer and read pointer are able to be only read from Internal Peripheral Bus. These registers are automatically increment when write to FIFO or read from FIFO.

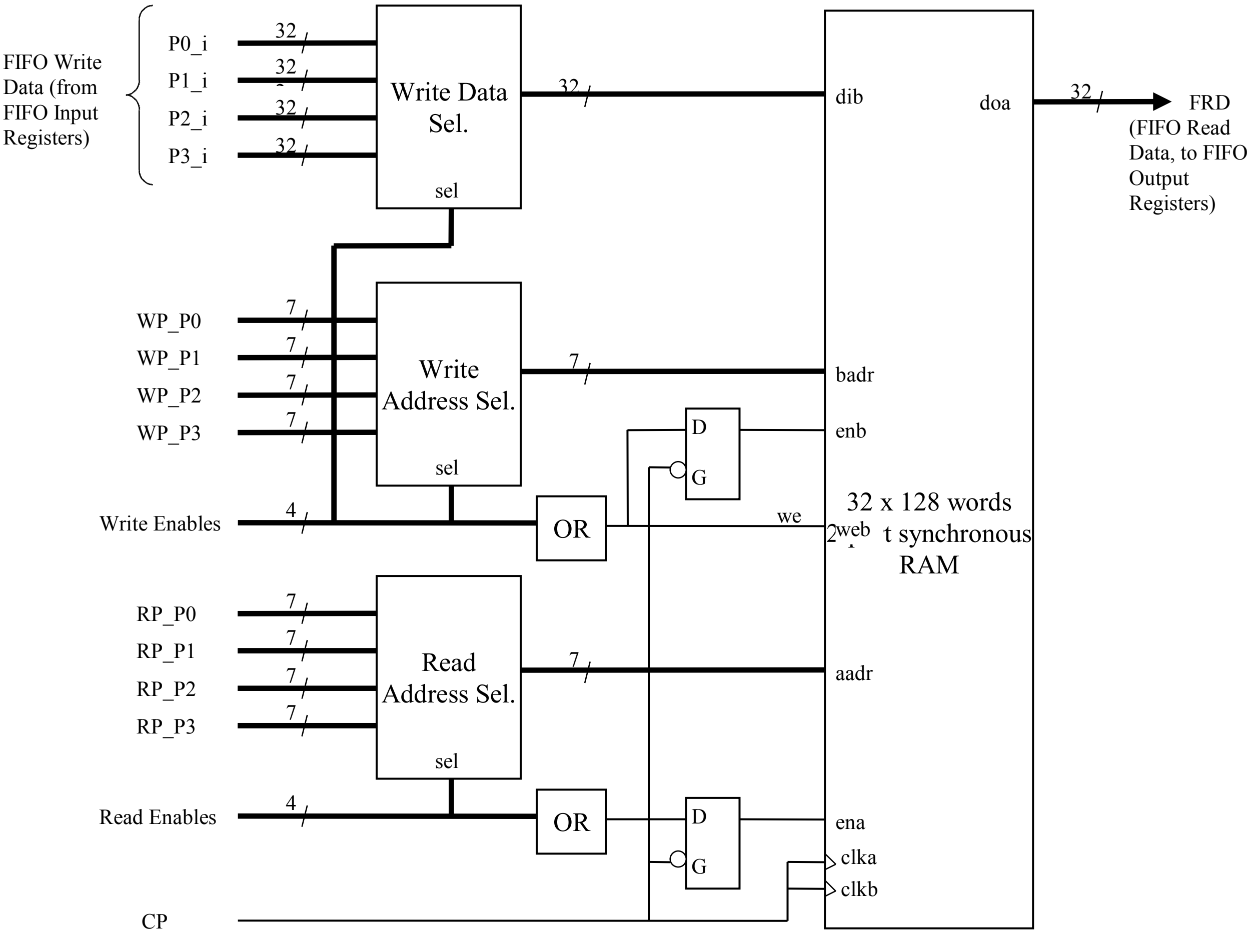
Each part has four flags, FF, FE, FH, and FI. FF and FE respectively indicates FIFO full and empty conditions. FH indicates half full condition (while a selected number of words is stored in memory.

In case of FIFO which is written by Internal Peripheral Bus (when I^{2}S Port is set to output), FI is asserted to 1 only one Imagine clock cycle, when a number of stored data is less than selected number (watermark). In case of FIFO which is read from Internal Peripheral Bus (when I^{2}S Port is set to input), FI is asserted to 1 only one Imagine clock cycle, when a number of stored data is greater than selected number (watermark). FI flags are used for Interrupt Generator to generate interrupt signals. These flags are read from Internal Peripheral Bus at any time.

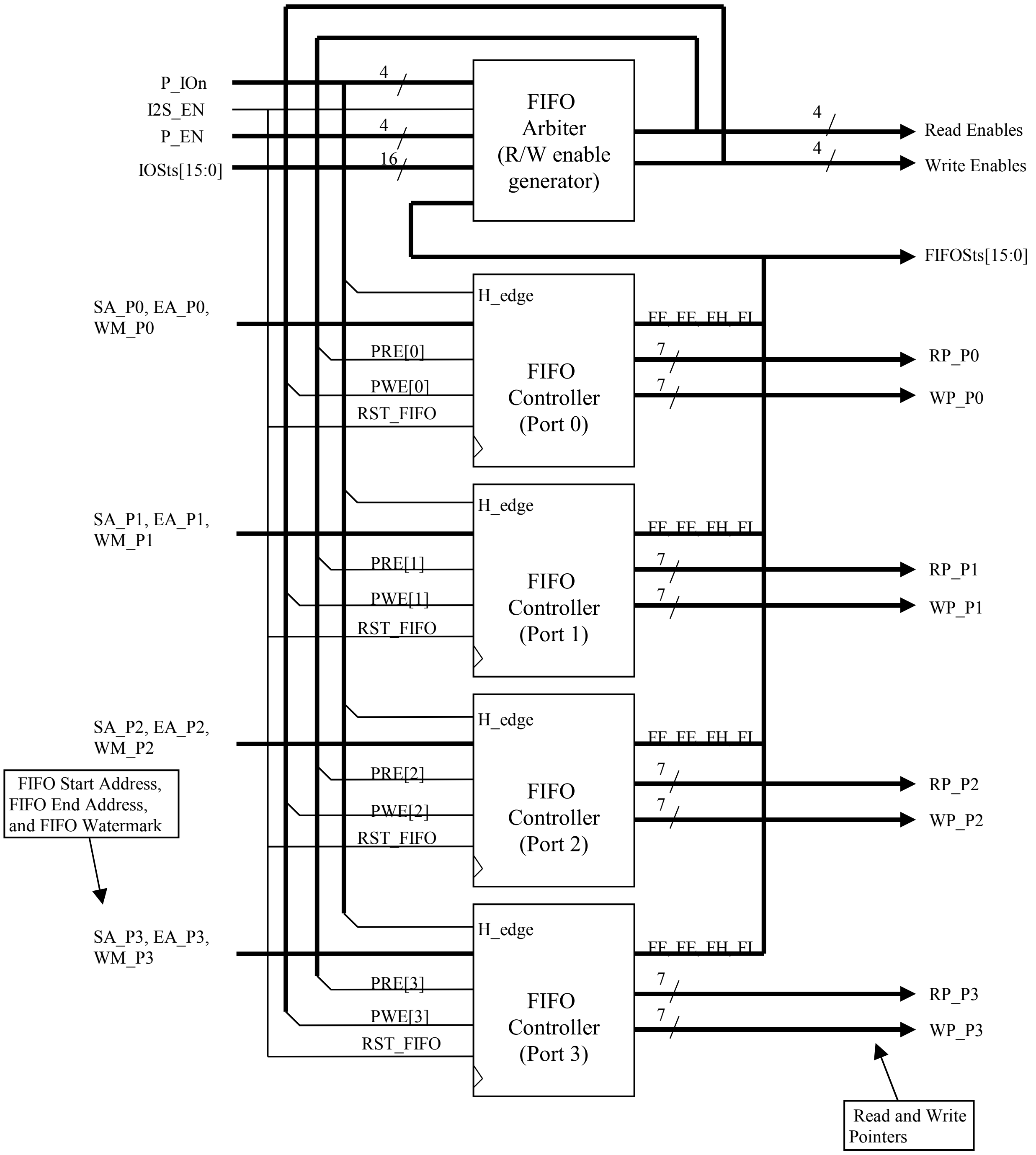
When accesses from Internal Peripheral Bus and I^{2}S ports simultaneously, the I^{2}S's access has priority. In case of write access is the same.

This block is completely synchronized with Imagine clock (not including delay cell or etc. to generate write pulse).

29.5.1 Block Diagram 1/2 (FIFO block)



29.5.2 Block Diagram 1/2 (Controller & Arbiter block)

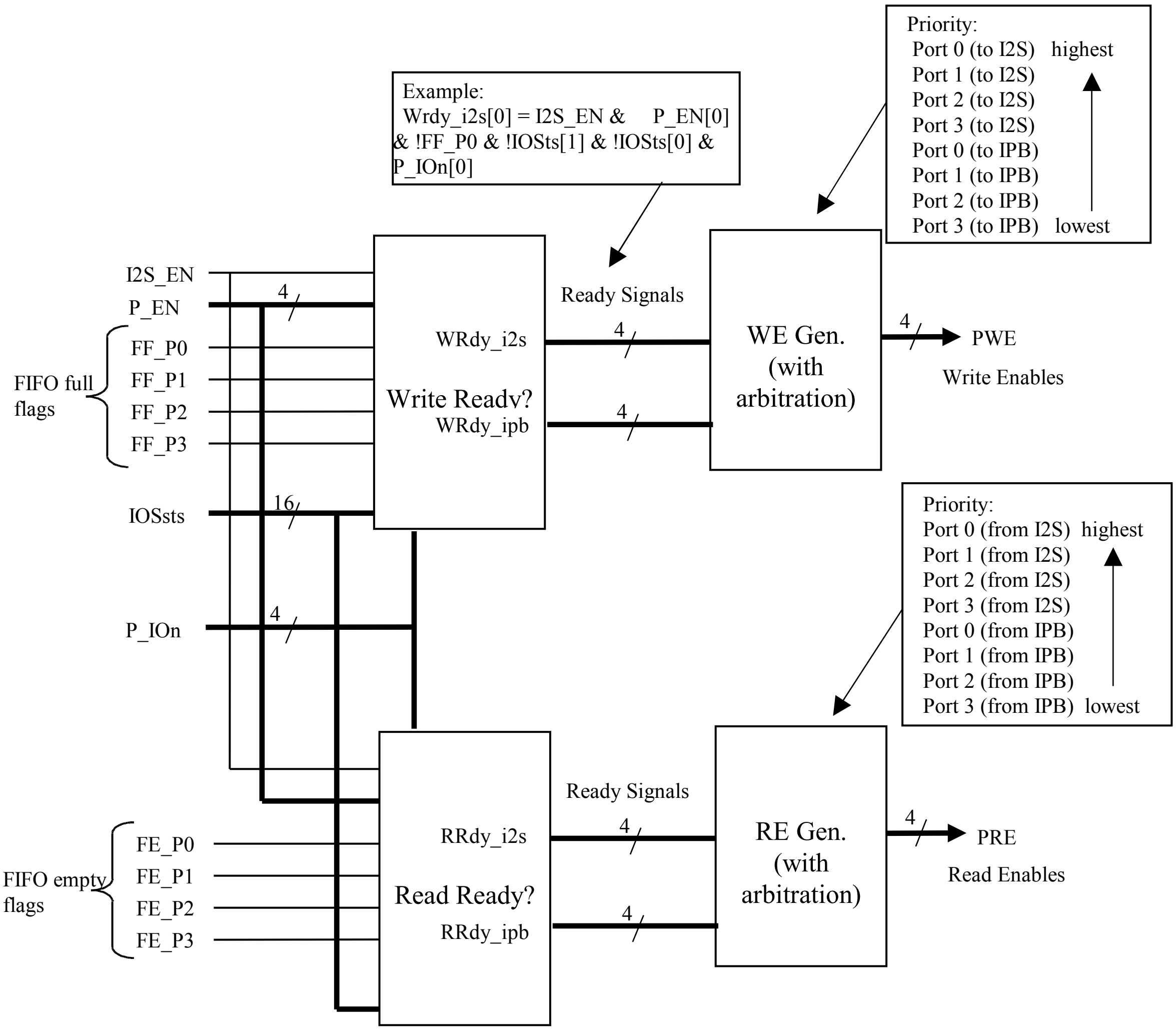


29.5.3 FIFO Arbiter (I2SFABT.v)

This block arbitrates write and read accesses from FIFO input/output registers and generates write and read enable signals. The write enable signals are used for FIFO write access and clears I/O status registers' IH, IL bits. The read enable signals are user for FIFO read access and set I/O status registers' OH, OL bits.

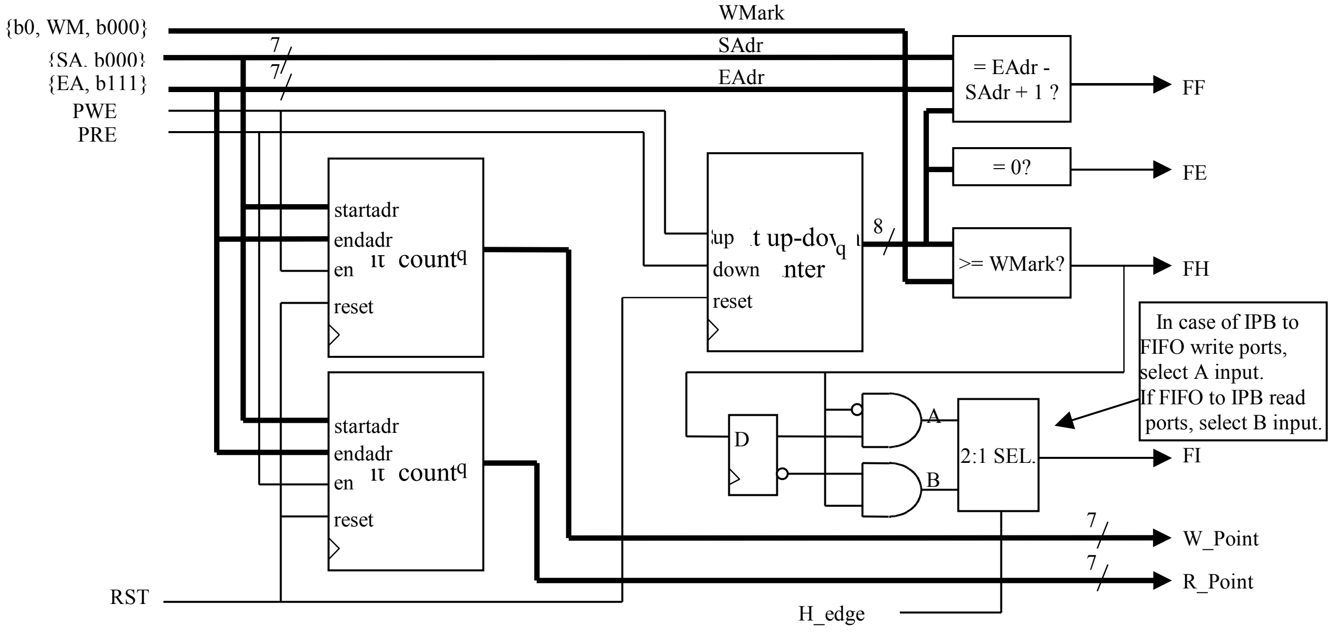
When accesses from Internal Peripheral Bus and from I^{2}S ports simultaneously, the I^{2}S's access has priority. In case of write access is the same.

If IH and IL bit (I/O status register) is 1 and FF flag (FIFO status register) is 0, the arbiter asserts write enable to 1. Besides, if OH and OL bit (I/O status register) is both 0 and FE flag (FIFO status register) is 0, the arbiter asserts read enable to 1.



29.5.4 FIFO Controller (I2SFCTRL.v)

This block two 7-bit up counters and one 8-bit up-down counter. Two 7-bit up counters are used for FIFO write pointer (memory write address) and FIFO read pointer (memory read address). One 8-bit up-down counter counts a number of data which written into FIFO. Start and end address (higher 4 bit) are defined by FIFO control registers. Lower 3 bit of start address are filled with '0'. Lower 3 bit of end address are filled with '1'. The block diagram of this module is as follows.



29.6 Interrupt Generator (I2S_IGEN.v)

This module generates two interrupt lines (INT_R, INT_W). The interrupt lines are connected to the Interrupt Vector Generator via the Interrupt Router. When output FIFO half empty states occurs, this module asserts INT_W to 1 while one Imagine clock cycle.

When input FIFO full empty states occurs, this module asserts INT_R to 1 while one Imagine clock cycle.

When some bit of Interrupt Enable Register (Interrupt Control register, bit 11:8) is set to 0, interrupts corresponding to the bit are disabled.

Interrupt conditions is assign to Interrupt Identify Register (Interrupt Control register, bit 3:0). This register is able to be read from Internal Peripheral Bus.

Conditions which INT_W is asserted to 1 are:

1. One or more I²S Port is set to Output (Main Control registers, bit 23..20). and
2. FIFO half full flag for the I²S Port turns to 0 (FIFO status registers, bit 13, 9, 5, 1). and
3. Interrupt enable for the I²S Port is set to 1 (Interrupt registers, bit 11..8).

EX: P_IOn[0] & !FIFOSts[1] & INT_EN[0]

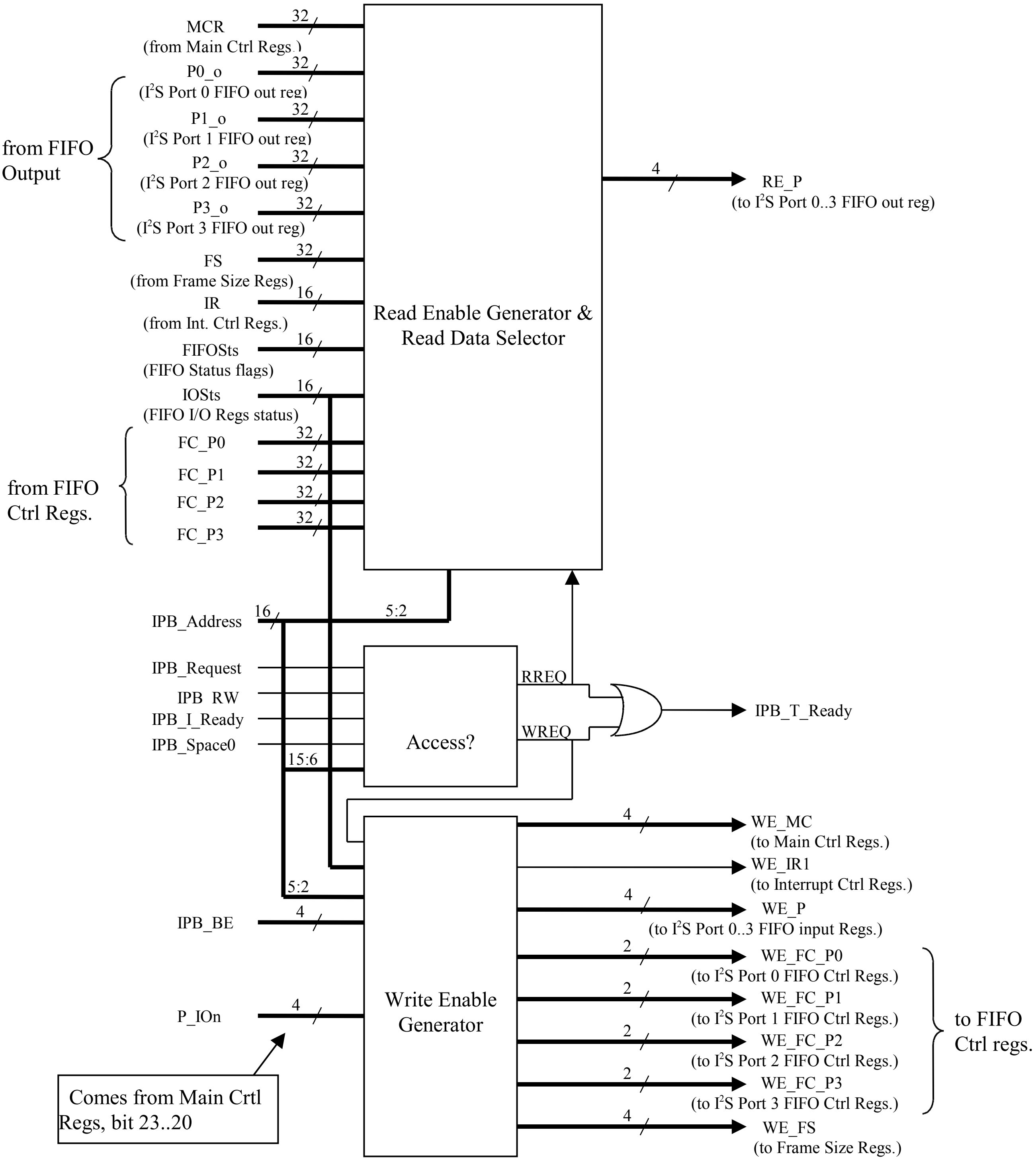
Conditions which INT_R is asserted to 1 are:

1. One or more I²S Port is set to Input (Main Control registers, bit 23..20). and
2. FIFO half full flag for the I²S Port turns to 1 (FIFO status registers, bit 13, 9, 5, 1). and
3. Interrupt enable for the I²S Port is set to 1 (Interrupt registers, bit 11..8).

EX: !P_IOn[0] & FIFOSts[1] & INT_EN[0]

29.7 Internal Peripheral Bus I/F (I2SIPBIO.v)

This module has Internal Peripheral Bus(IPB) interface giving direct access to the registers (include some FIFO Input/ Output registers) of this module. These are mapped on IPB address space 0 (IPB_Space0). 64 byte ranges are needed. These registers are addressed directly by IPB_Address[15:2] from IPB and can be written or read at any time, except some FIFO data ports. In any case, write or read transfer to control registers always successfully terminate at first try. In case of accesses to FIFO data write ports, if this register is empty, only 1 Imagine clock cycle is needed to this transfer. If this register contains data, the Initiator must wait until this register is empty. In case of accesses to FIFO data read ports, if this register contains data, only 1 Imagine clock cycle is needed to this transfer. If this register is empty, the Initiator must wait until this register contains data. When Initiator writes to read only registers (e.g. I^{2}S Port is set to input), this module only set IPB_T_Ready to 1, vice versa..



29.8 I^{2}S Registers

29.8.1 I^{2}S Controller base address

IPB_Address

<table>
<tr><td colspan="10">I^{2}S BASE
(10b0000000110)</td><td colspan="4">I^{2}S
register
space</td></tr>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td></tr>
</table>

29.8.2 I^{2}S Controller registers memory map

<table>
<tr><td></td><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td colspan="32">I^{2}S Main Control
Register</td></tr>
<tr><td>1</td><td colspan="16">I^{2}S Input/Output Port 0
(Left channel)</td><td colspan="16">I^{2}S Input/Output Port 0
(Right channel)</td></tr>
<tr><td>2</td><td colspan="16">I^{2}S Input/Output Port 1
(Left channel)</td><td colspan="16">I^{2}S Input/Output Port 1
(Right channel)</td></tr>
<tr><td>3</td><td colspan="16">I^{2}S Input/Output Port 2
(Left channel)</td><td colspan="16">I^{2}S Input/Output Port 2
(Right channel)</td></tr>
<tr><td>4</td><td colspan="16">I^{2}S Input/Output Port 3
(Left channel)</td><td colspan="16">I^{2}S Input/Output Port 3
(Right channel)</td></tr>
<tr><td>5</td><td colspan="32">Reserved (0x00000000)</td></tr>
<tr><td>6</td><td colspan="32">I^{2}S Interrupt Control
Register</td></tr>
<tr><td>7</td><td colspan="32">I^{2}S Input/ Output FIFO
Status Register</td></tr>
<tr><td>8</td><td colspan="32">I^{2}S FIFO Input / Output Port
Status Register</td></tr>
<tr><td>9</td><td colspan="32">I^{2}S Port 0 FIFO
Control / Status Register</td></tr>
<tr><td>10</td><td colspan="32">I^{2}S Port 1 FIFO
Control / Status Register</td></tr>
<tr><td>11</td><td colspan="32">I^{2}S Port 2 FIFO
Control / Status Register</td></tr>
<tr><td>12</td><td colspan="32">I^{2}S Port 3 FIFO
Control / Status Register</td></tr>
<tr><td>13</td><td colspan="32">Reserved (0x00000000)</td></tr>
<tr><td>14</td><td colspan="32">I^{2}S Frame Size
Register</td></tr>
<tr><td>15</td><td colspan="32">Reserved (0x00000000)</td></tr>
</table>

IPB_Address[5:2]

29.8.3 I^{2}S Main Control Registers

This registers totally controls I^{2}S Controller.

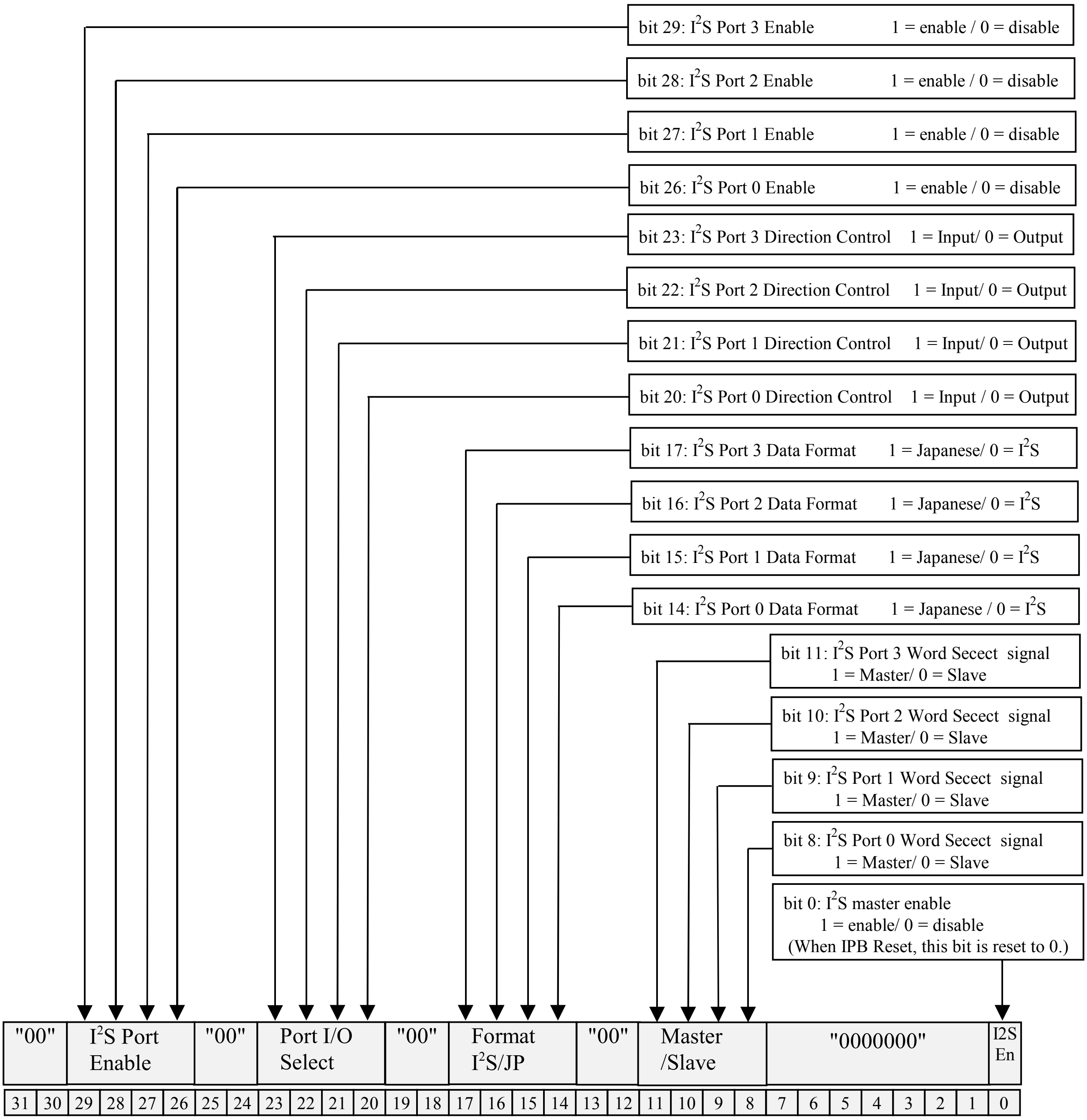
Bit 0 is master enable bit of the I^{2}S controller (this module). When this bit is set to 0, all functions in this module are disabled, and all FIFO pointers are zero clear. When IPB_RESET, this bit is reset to 0.

Bit 29:26 are Port Enables. controls each ports of I^{2}S input/output data streams. These registers enables or disable each ports of FIFO inputs/outputs.

Bit 23:20 are used as the DIR control signal of bidirectional SDI/SDO I/O buffers. When this bit is set to 1, SDI/SDO port is set to input.

Bit 11:8 are used as the DIR control signal of bidirectional MASTER I/O buffers. When this bit is set to 1, WS signal is set to output, and this module uses the internal timing signal.

This registers can be write or read from Internal Peripheral Bus. Register values are not initialized except bit 0.



bit 29: I^{2}S Port 3 Enable 1 = enable / 0 = disable

bit 28: I^{2}S Port 2 Enable 1 = enable / 0 = disable

bit 27: I^{2}S Port 1 Enable 1 = enable / 0 = disable

bit 26: I^{2}S Port 0 Enable 1 = enable / 0 = disable

bit 23: I^{2}S Port 3 Direction Control 1 = Input/ 0 = Output

bit 22: I^{2}S Port 2 Direction Control 1 = Input/ 0 = Output

bit 21: I^{2}S Port 1 Direction Control 1 = Input/ 0 = Output

bit 20: I^{2}S Port 0 Direction Control 1 = Input / 0 = Output

bit 17: I^{2}S Port 3 Data Format 1 = Japanese/ 0 = I^{2}S

bit 16: I^{2}S Port 2 Data Format 1 = Japanese/ 0 = I^{2}S

bit 15: I^{2}S Port 1 Data Format 1 = Japanese/ 0 = I^{2}S

bit 14: I^{2}S Port 0 Data Format 1 = Japanese / 0 = I^{2}S

bit 11: I^{2}S Port 3 Word Secect signal
1 = Master/ 0 = Slave

bit 10: I^{2}S Port 2 Word Secect signal
1 = Master/ 0 = Slave

bit 9: I^{2}S Port 1 Word Secect signal
1 = Master/ 0 = Slave

bit 8: I^{2}S Port 0 Word Secect signal
1 = Master/ 0 = Slave

bit 0: I^{2}S master enable
1 = enable/ 0 = disable
(When IPB Reset, this bit is reset to 0.)

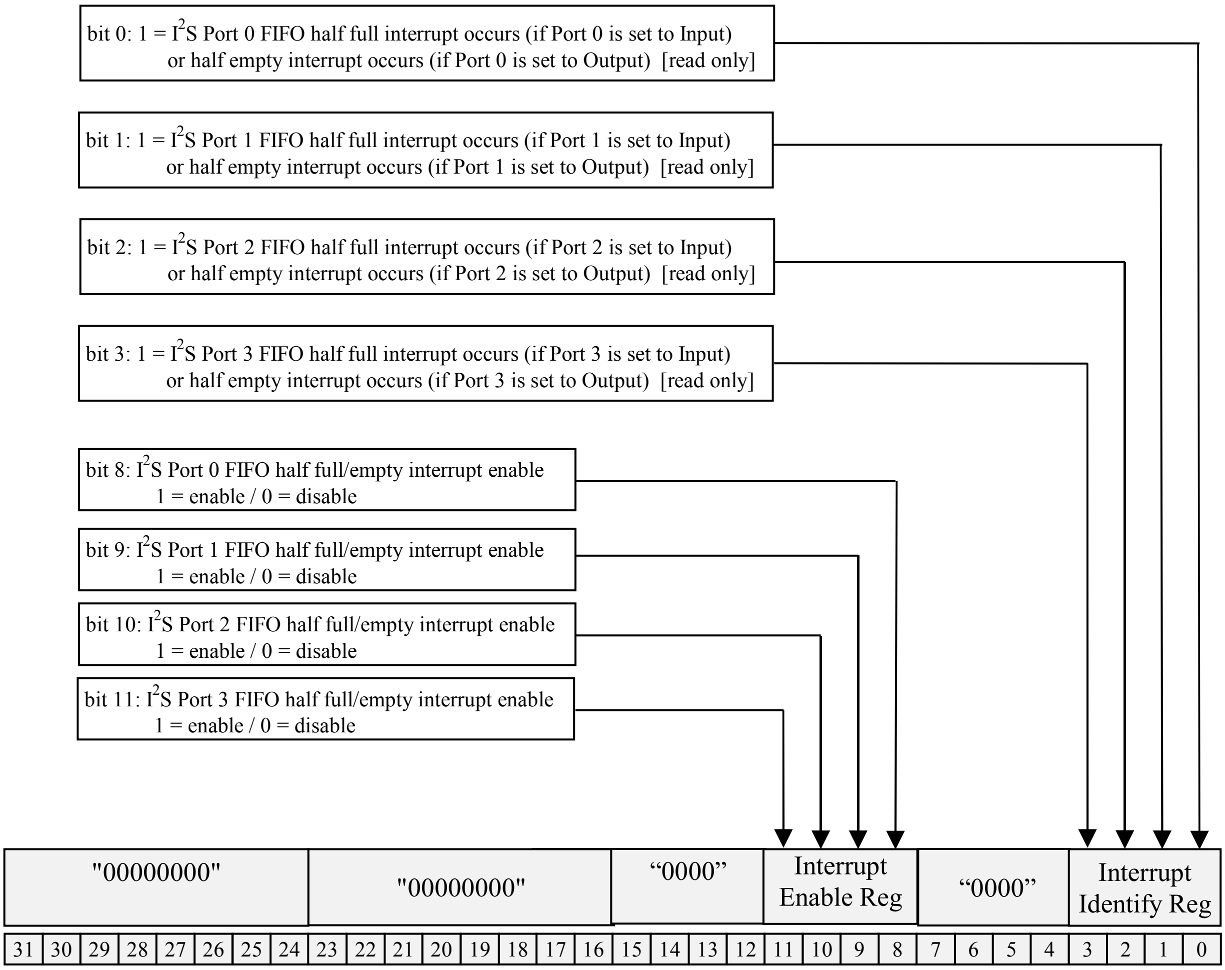
"00"	I^{2}S Port Enable	"00"	Port I/O Select	"00"	Format I^{2}S/JP	"00"	Master /Slave	"0000000"	I2S En
31 30	29 28 27 26	25 24	23 22 21 20	19 18	17 16 15 14	13 12	11 10 9 8	7 6 5 4 3 2 1	0

29.8.4 Interrupt Control Register

This register controls two interrupt lines, INT_R and INT_W. When each parts of FIFO half empty/full state occurs, this module generates interrupt.

Bit 11:8 are interrupt enables for each FIFO input / output ports (including command write/read ports, which does not use the FIFO). If some bit of this enables are set to 0, the corresponding FIFO full or empty interrupts are disabled. These bit are not initialized. This register is able to be written or read from Internal Peripheral Bus at any time.

Bit 3:0 are interrupt identify registers for checking the interrupt status. This register indicates which port is in the interrupt state. These bit are able to be only read from Internal Peripheral Bus.



bit 0: 1 = I^{2}S Port 0 FIFO half full interrupt occurs (if Port 0 is set to Input)
or half empty interrupt occurs (if Port 0 is set to Output) [read only]

bit 1: 1 = I^{2}S Port 1 FIFO half full interrupt occurs (if Port 1 is set to Input)
or half empty interrupt occurs (if Port 1 is set to Output) [read only]

bit 2: 1 = I^{2}S Port 2 FIFO half full interrupt occurs (if Port 2 is set to Input)
or half empty interrupt occurs (if Port 2 is set to Output) [read only]

bit 3: 1 = I^{2}S Port 3 FIFO half full interrupt occurs (if Port 3 is set to Input)
or half empty interrupt occurs (if Port 3 is set to Output) [read only]

bit 8: I^{2}S Port 0 FIFO half full/empty interrupt enable
1 = enable / 0 = disable

bit 9: I^{2}S Port 1 FIFO half full/empty interrupt enable
1 = enable / 0 = disable

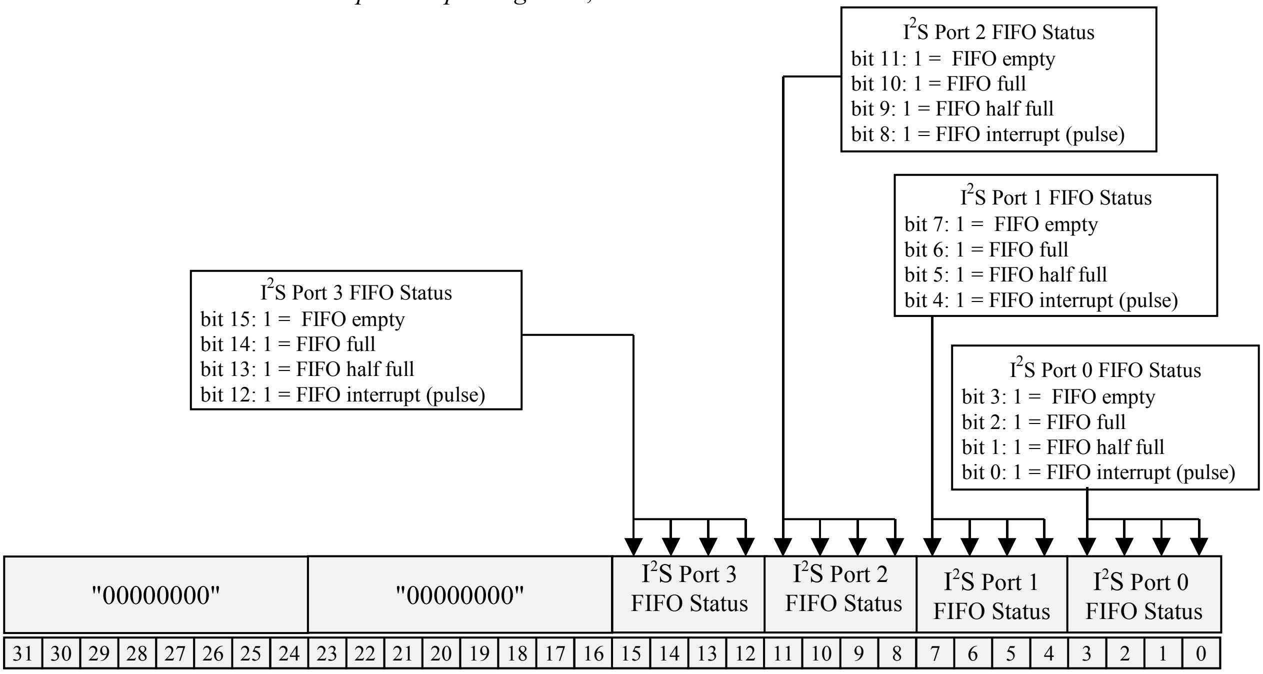
bit 10: I^{2}S Port 2 FIFO half full/empty interrupt enable
1 = enable / 0 = disable

bit 11: I^{2}S Port 3 FIFO half full/empty interrupt enable
1 = enable / 0 = disable

"00000000"								"00000000"								"0000"				Interrupt Enable Reg				"0000"				Interrupt Identify Reg			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

29.8.5 Input/Output FIFO Status Register

This registers contains FIFO status. The FIFOs which is separated to four independent blocks have each four status flags, FIFO empty (FE), FIFO full (FF), FIFO half empty or full (FH), and FIFO interrupt (FI). Every FI bit are used for generating interrupts. This registers are able to be only read from Internal Peripheral Bus at any time. See subsection *2.4 FIFO Input/ Output Registers*, for more details.



I^{2}S Port 2 FIFO Status
bit 11: 1 = FIFO empty
bit 10: 1 = FIFO full
bit 9: 1 = FIFO half full
bit 8: 1 = FIFO interrupt (pulse)

I^{2}S Port 1 FIFO Status
bit 7: 1 = FIFO empty
bit 6: 1 = FIFO full
bit 5: 1 = FIFO half full
bit 4: 1 = FIFO interrupt (pulse)

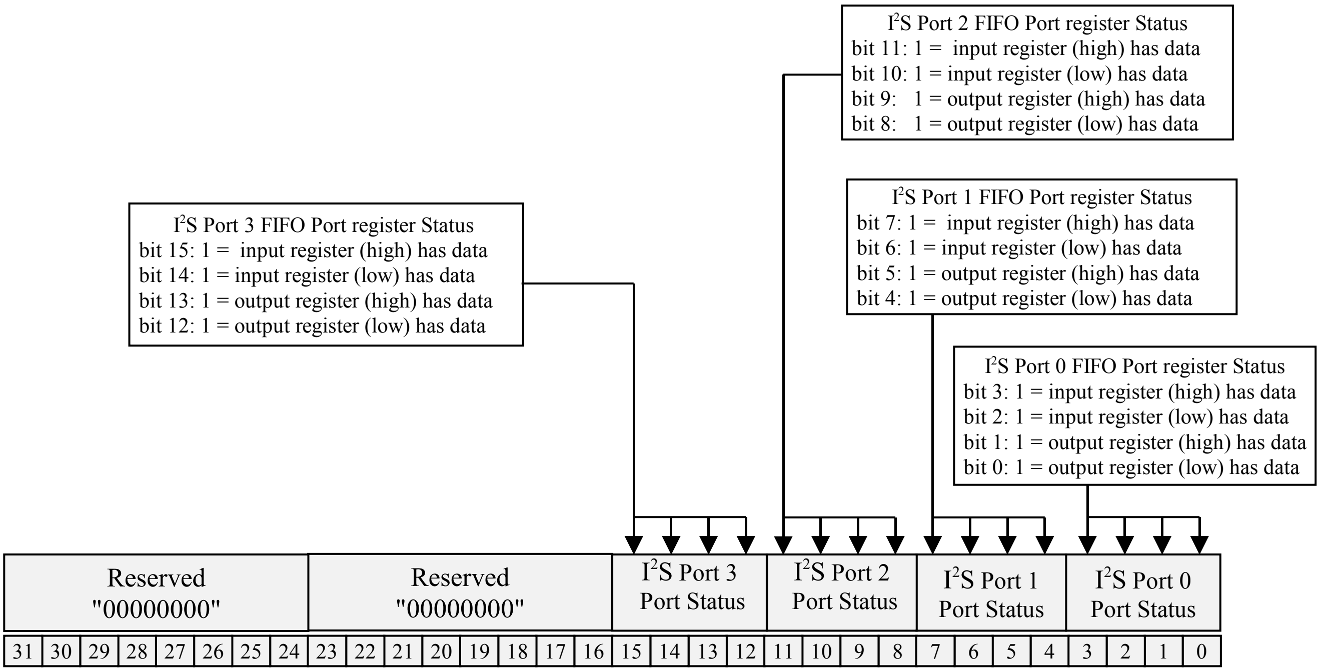
I^{2}S Port 3 FIFO Status
bit 15: 1 = FIFO empty
bit 14: 1 = FIFO full
bit 13: 1 = FIFO half full
bit 12: 1 = FIFO interrupt (pulse)

I^{2}S Port 0 FIFO Status
bit 3: 1 = FIFO empty
bit 2: 1 = FIFO full
bit 1: 1 = FIFO half full
bit 0: 1 = FIFO interrupt (pulse)

31–24	23–16	15–12	11–8	7–4	3–0
"00000000"	"00000000"	I^{2}S Port 3 FIFO Status	I^{2}S Port 2 FIFO Status	I^{2}S Port 1 FIFO Status	I^{2}S Port 0 FIFO Status

29.8.6 FIFO Input/Output Port Status Register

This registers contains input/output port (FIFO Input/Output register) status. The four FIFO input or output ports have each four status flags, higher 16-bit of FIFO input register has data (IH), lower 16-bit of FIFO input register has data (IL), higher 16-bit of FIFO output register has data (OH), and lower 16-bit of FIFO output register has data (OL). These registers able to be only read from Internal Peripheral Bus at any time. See subsection *2.4 Interrupt Generator*, for more details.



I^{2}S Port 2 FIFO Port register Status
bit 11: 1 = input register (high) has data
bit 10: 1 = input register (low) has data
bit 9: 1 = output register (high) has data
bit 8: 1 = output register (low) has data

I^{2}S Port 1 FIFO Port register Status
bit 7: 1 = input register (high) has data
bit 6: 1 = input register (low) has data
bit 5: 1 = output register (high) has data
bit 4: 1 = output register (low) has data

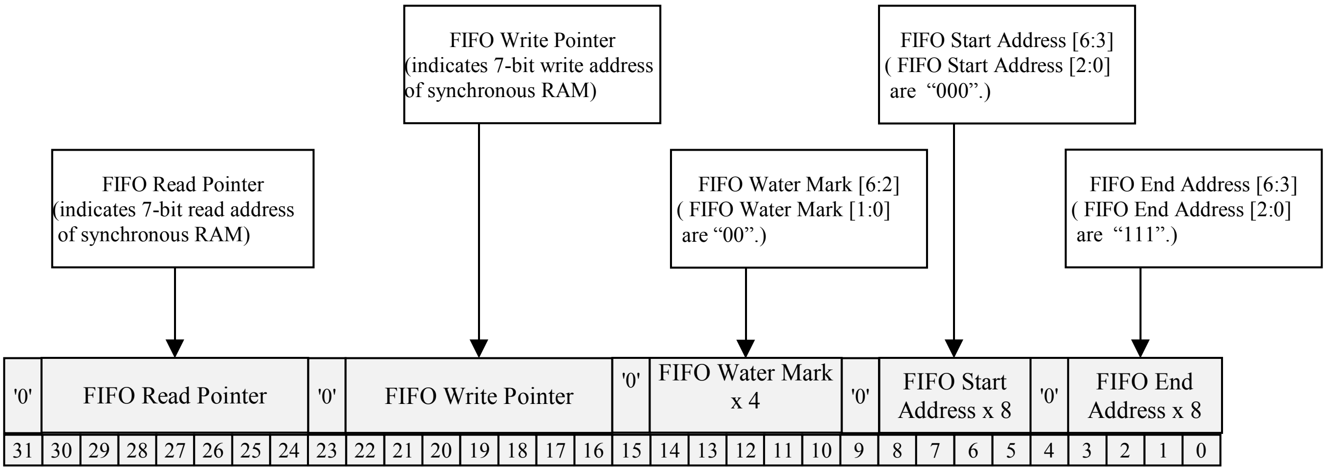
I^{2}S Port 3 FIFO Port register Status
bit 15: 1 = input register (high) has data
bit 14: 1 = input register (low) has data
bit 13: 1 = output register (high) has data
bit 12: 1 = output register (low) has data

I^{2}S Port 0 FIFO Port register Status
bit 3: 1 = input register (high) has data
bit 2: 1 = input register (low) has data
bit 1: 1 = output register (high) has data
bit 0: 1 = output register (low) has data

31–24	23–16	15–12	11–8	7–4	3–0
Reserved "00000000"	Reserved "00000000"	I^{2}S Port 3 Port Status	I^{2}S Port 2 Port Status	I^{2}S Port 1 Port Status	I^{2}S Port 0 Port Status

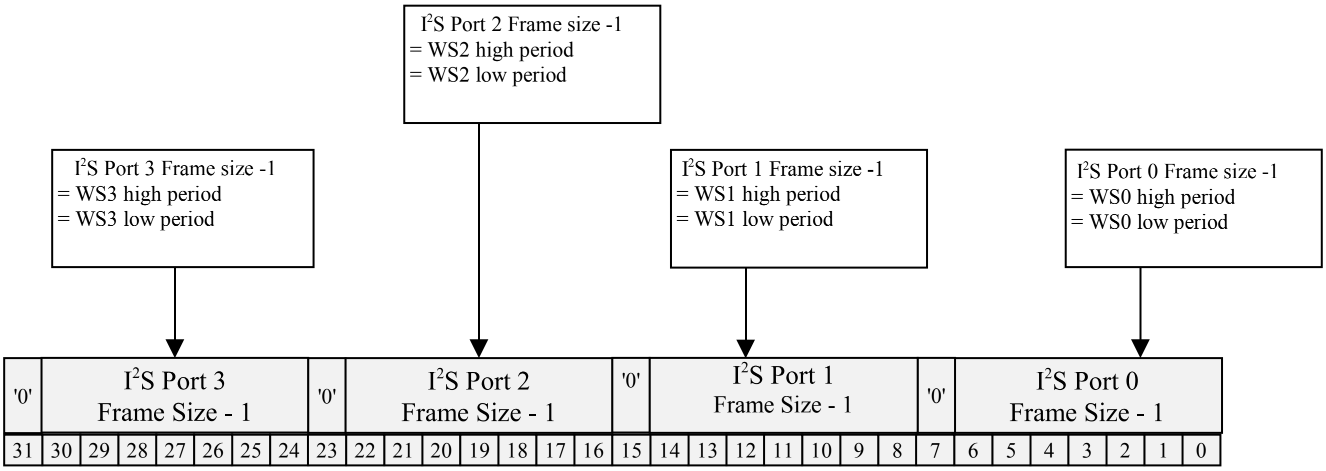
29.8.7 FIFO Control/Status Registers

This registers are used for definitions of the FIFO. This registers contains FIFO start address, FIFO end address, and FIFO watermark. Each FIFO block has one FIFO Control /Status Registers, amount to five registers. These values are able to be written or read from Internal Peripheral Bus at any time. Bit 30:24 are FIFO read pointer, Bit 22:16 are FIFO write pointer. These registers are able to be only read from Internal Peripheral Bus. See subsection *2.3.4 FIFO Controller*, for more details.



29.8.8 Frame Size registers

This registers are used for definitions of the frame size of the data streams. The value plus 1 is define to Word Select (WS) signal's period of the left channel and right channel. This values are used only in Japanese format (when in I^{2}S mode, this values are not referenced). The frame sizes are programmable for each ports individually. These registers are able to be written or read from Internal Peripheral Bus at any time.



29.9 I^{2}S Data Access Ports

In order to communicate with I^{2}S digital interface, eight 32-bit registers are prepared. Each registers are connected via internal FIFO. These registers are read only or write only registers. The status of registers are able to read the FIFO port status registers.

29.9.1 I^{2}S FIFO input port registers

If I^{2}S data port is set to output, this registers is connected to Internal Peripheral Bus. This register is always connected to FIFO Input. Bit 31:16 are 16-bit left channel data, and bit 15:0 are right channel data. This value is used for I^{2}S serial output data. This registers are write only registers, and must be written 32-bit simultaneously.
If I^{2}S data port is set to input, this registers is disconnected to Internal Peripheral Bus, and used for I^{2}S serial data input.

29.9.2 I^{2}S FIFO output port registers

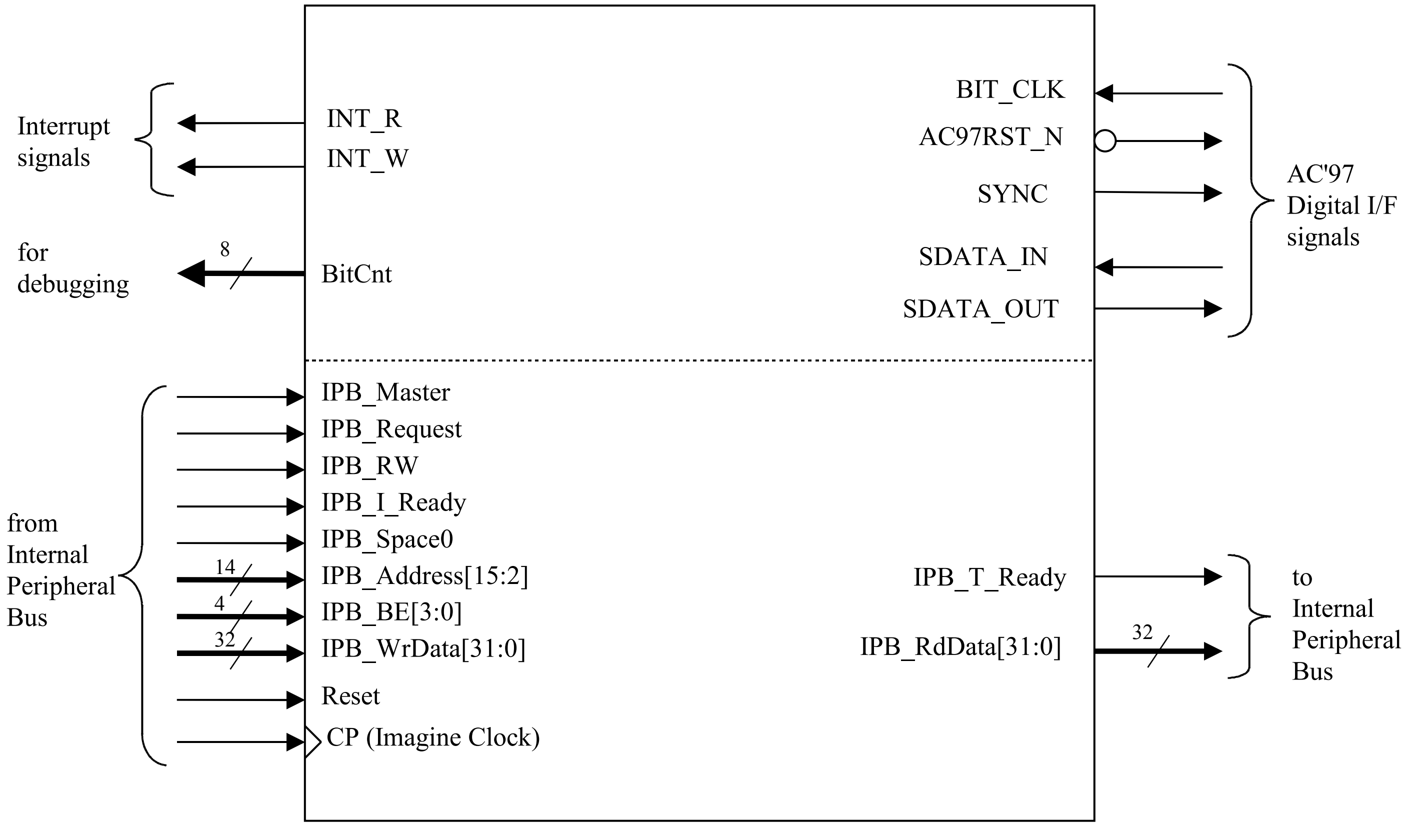
If I^{2}S data port is set to input, this registers is connected to Internal Peripheral Bus. This register is always connected to FIFO Output. Bit 31:16 are 16-bit left channel data, and bit 15:0 are right channel data. This values comes from I^{2}S serial input port. This registers are read only registers, and must be read 32-bit simultaneously.
If I^{2}S data port is set to output, this registers is disconnected to Internal Peripheral Bus, and used for I^{2}S serial data output.

Chapter

30. THE AC97 AUDIO CODEC

The AC97 audio codec transmits and receives audio data via a 5 line serial interface connected to an external AC97 codec which includes AD and DA converters plus a mixer. This external codec can be controlled via this interface. An on chip 128 by 32 bit word RAM can be programmed as one or more fifo(s) for all the defined audio and modem I/O channels: Playback output, Record input, Modem input and Modem output and Microphone input. The sizes of the individual fifos are programmable as well as the watermarks which can generate interrupts for the Imagine 2 core processor on almost full / almost empty detection.

30.1 The Input/Output Signals of AC'97 Controller



omit the RAMBIST signals.

30.1.1 Input/ Output signals definitions

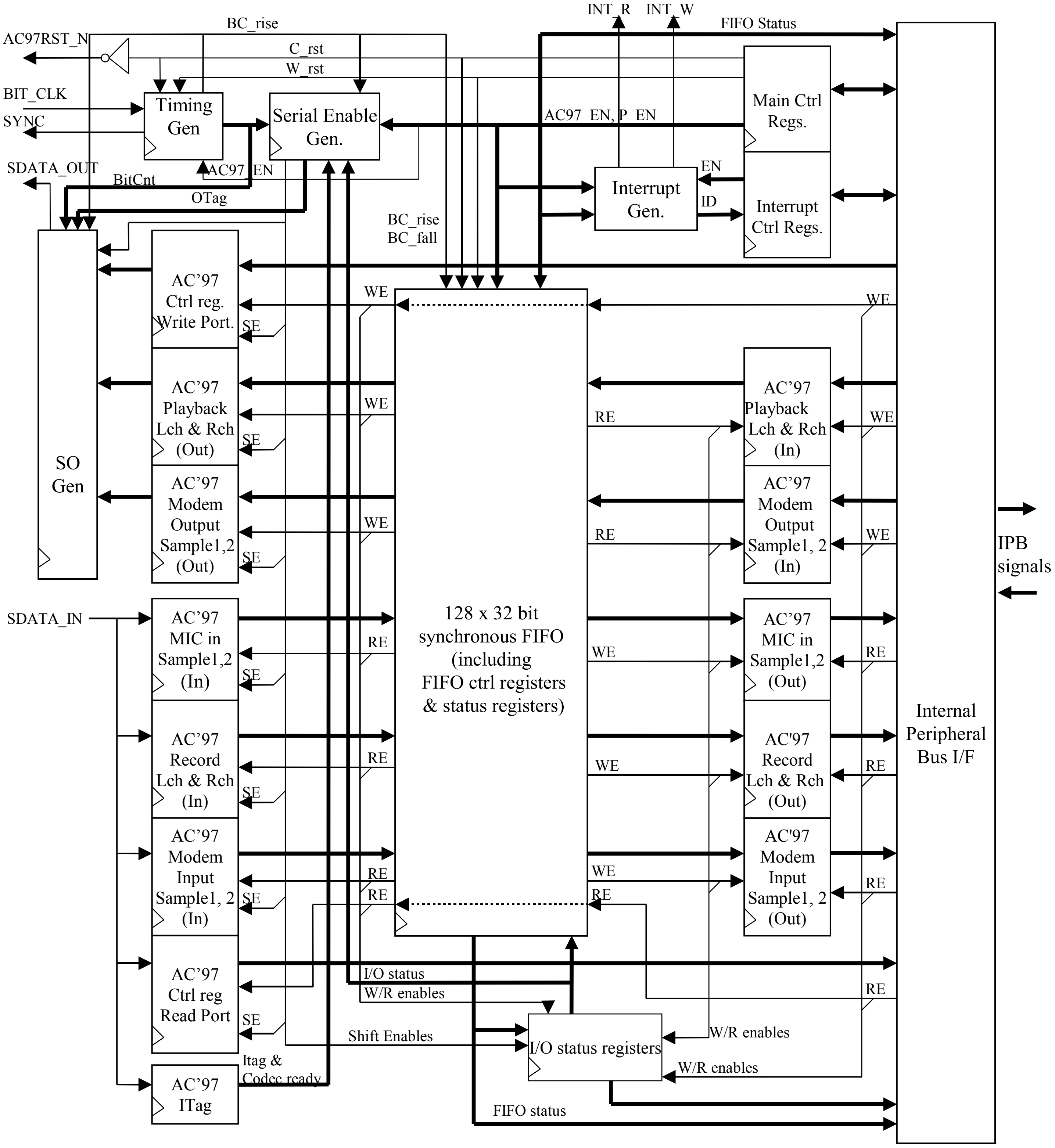
AC'97 Interface signals		
BIT_CLK	input	12.288 MHz serial data clock
AC97RST_N	output	AC'97 Master H/W Reset signal. When C_RST (Main Control register, bit 0) is set to 1, this signal is asserted to 0 (activate).
SYNC	output	48 KHz fixed rate sample sync. This signal is synchronized with Imagine Clock.
SDATA_IN	input	Serial, time division multiplexed, AC'97 input stream. This signal is synchronized with Imagine Clock.
SDATA_OUT	output	Serial, time division multiplexed, AC'97 output stream. This signal is synchronized with Imagine Clock.

Interrupt signals (to Interrupt Vector Generator)		
INT_R	output	Read interrupt flag.
INT_W	output	Write interrupt flag.
Internal Peripheral Bus I/F signals		
CP (Imagine Clock)	input	See document *“The Protocol of the INTERNAL PERIPHERAL BUS, revision 0.9a”*
Reset	input	
IPB_Master	input	
IPB_Request	input	
IPB_RW	input	
IPB_T_Ready	output	
IPB_I_Ready	input	
IPB_Space0	input	
IPB_Address[15:2]	input	
IPB_BE [3:0]	input	
IPB_RdData [31:0]	output	
IPB_WrData [31:0]	input	

30.2 AC'97 controller module overview

AC'97 controller has three interfaces, AC-Link, Internal Peripheral Bus, and two interrupt output signals. AC-link (Audio Codec '97 controller digital serial link) is used to communicate with AC'97 chips. This interface is based on *"Audio Codec '97 Component Specification, Revision 1.03"*.

Internal Peripheral Bus is connected with some internal control registers and I/O registers. These registers are on IPB_Space0, 64 bytes address area. Internal Peripheral Bus communicates with AC-link via internal 128 word x 32 bit FIFO. This FIFO is separated to five areas, and each areas have a independent I/O registers and Read/Write pointers. The two interrupt lines (INT_R, INT_W) are connected to the Interrupt Vector Generator via the Interrupt Router. When input or output FIFO almost full/empty occurs, this module generates interrupt pulse. All flip-flops and FIFO are synchronized with Imagine Clock.



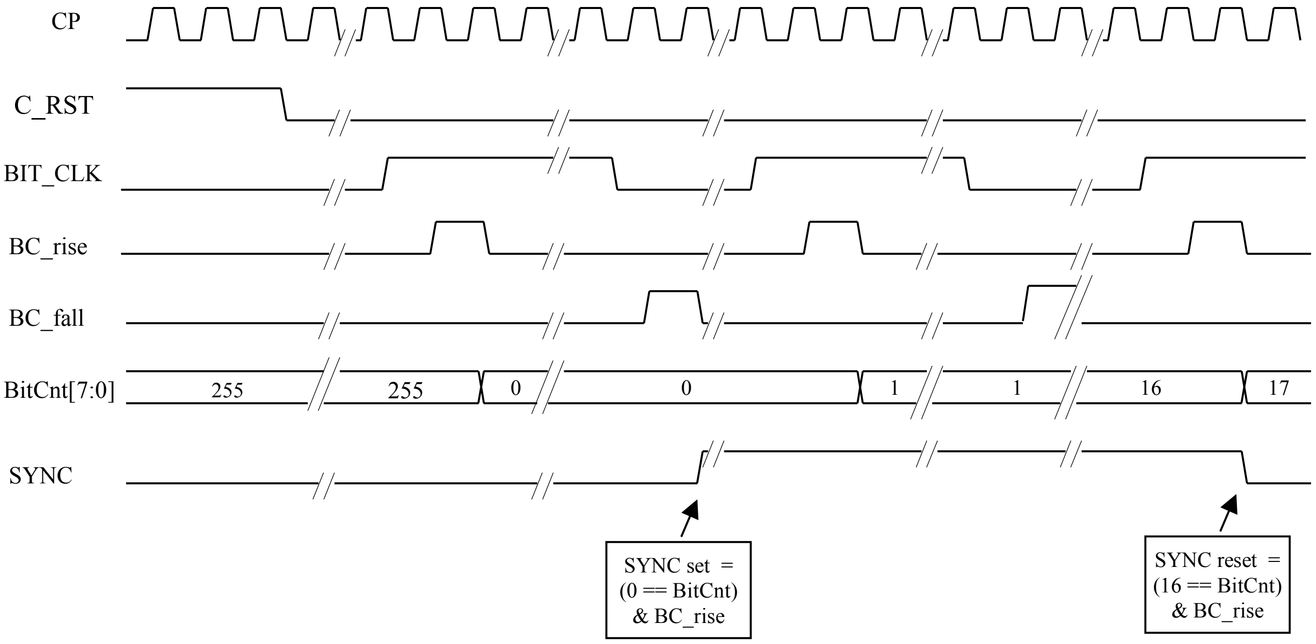
AC97RST_N
BIT_CLK
SYNC
SDATA_OUT
SDATA_IN
BC_rise
C_rst
W_rst
INT_R
INT_W
FIFO Status
Timing Gen
Serial Enable Gen.
AC97_EN, P_EN
Main Ctrl Regs.
EN
ID
Interrupt Gen.
Interrupt Ctrl Regs.
BitCnt
AC97_EN
OTag
BC_rise
BC_fall
SO Gen
AC'97 Ctrl reg. Write Port.
AC'97 Playback Lch & Rch (Out)
AC'97 Modem Output Sample1,2 (Out)
AC'97 MIC in Sample1,2 (In)
AC'97 Record Lch & Rch (In)
AC'97 Modem Input Sample1, 2 (In)
AC'97 Ctrl reg Read Port
AC'97 ITag
Itag & Codec ready
WE
SE
RE
128 x 32 bit synchronous FIFO (including FIFO ctrl registers & status registers)
AC'97 Playback Lch & Rch (In)
AC'97 Modem Output Sample1, 2 (In)
AC'97 MIC in Sample1,2 (Out)
AC'97 Record Lch & Rch (Out)
AC'97 Modem Input Sample1, 2 (Out)
I/O status
W/R enables
Shift Enables
I/O status registers
W/R enables
W/R enables
FIFO status
Internal Peripheral Bus I/F
IPB signals

30.3 AC'97 Serial Timing Generator (T_GEN.v)

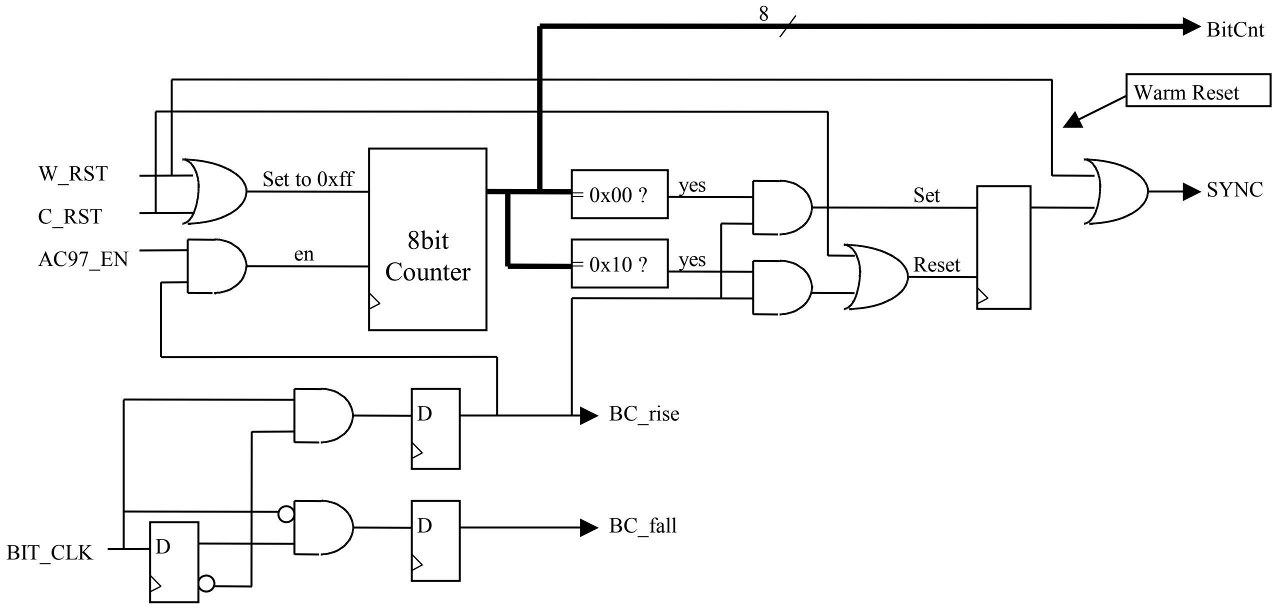
This block generates 48 kHz fixed rate serial sync signal (SYNC). This block samples BIT_CLK by Imagine Clock (CP), detects rising and falling edge of BIT_CLK. The BIT_CLK counter counts up the rising edge of BIT_CLK, generates SYNC signal and serial input/output timing.

When C_RST (Main Control register, bit 0) is set to 1, BIT_CLK counter is reset to 0. When W_RST (Main Control register, bit 1) is set to 1, BIT_CLK counter is reset to 0, and SYNC signal is asserted to 1 for Warm AC'97 reset state.

30.3.1 The timing of Serial Timing Generator


CP
C_RST
BIT_CLK
BC_rise
BC_fall
BitCnt[7:0]
255
255
0
0
1
1
16
17
SYNC
SYNC set = (0 == BitCnt) & BC_rise
SYNC reset = (16 == BitCnt) & BC_rise

30.3.2 Block Diagram



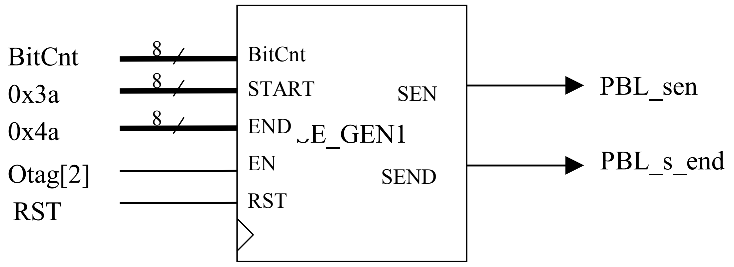
30.4 AC’97 Serial Data Enable Generator (SE_GEN.v)

This block generates AC’97 serial data write and read enables. The read enables are used to separate each audio data from multiple serial input (SDATA_IN). The separated audio data are store to 32-bit FIFO input registers. The write enables are used for making of multiple serial output (SDATA_OUT) from 32-bit FIFO output registers. All enables are synchronized with Imagine Clock and based on the value of BitCnt.

30.4.1 Block diagram

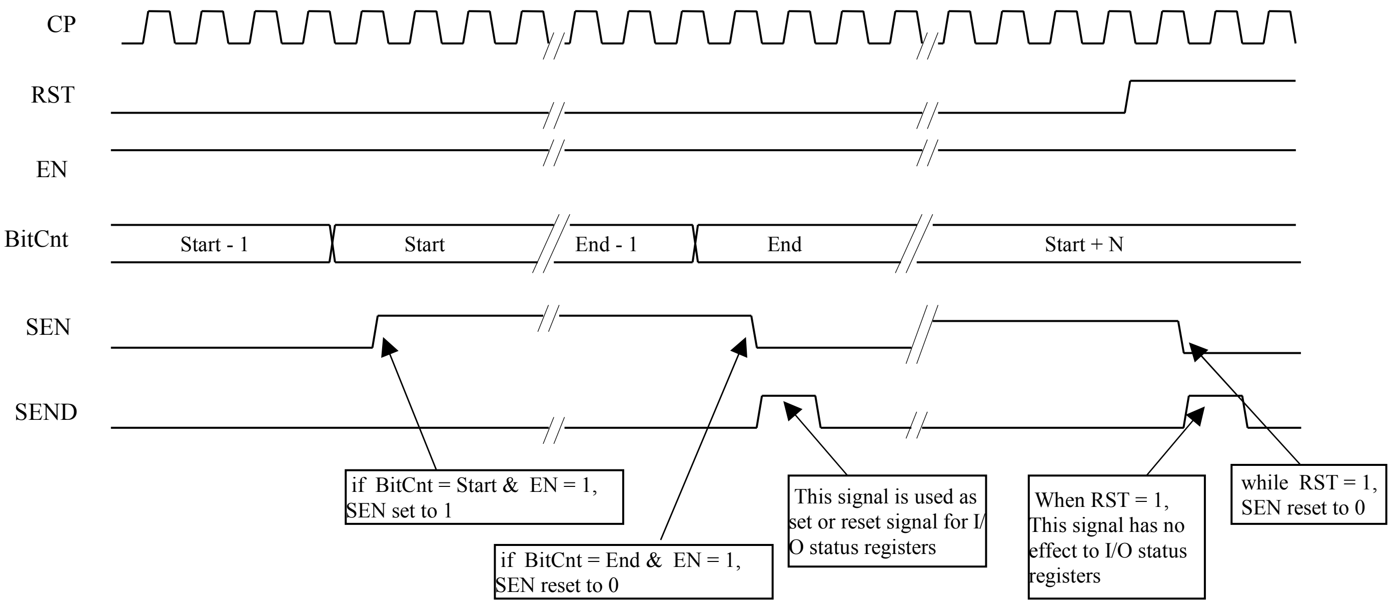
Serial Enable Generator (SE_GEN.v) has eleven sub modules names *SE_GEN1* for serial enable signals generation. BitCnt, RST (= C_RST | W_RST), and CP are connected to each *SE_GEN1*s as common inputs. Each modules has different inputs and outputs for the other signals.
The example of this block is as follows.

Example: Generate Playback Lch



START and END is a fixed values. PBL_sen is used as serial enable signal of Playback FIFO output register (Lch). PBL_s_en is used as reset signal of IOSts bit 5 (I/O status register Playback out OH).

Internal Timing of SE_GEN1


CP
RST
EN
BitCnt
SEN
SEND
Start - 1
Start
End - 1
End
Start + N
if BitCnt = Start & EN = 1, SEN set to 1
if BitCnt = End & EN = 1, SEN reset to 0
This signal is used as set or reset signal for I/O status registers
When RST = 1, This signal has no effect to I/O status registers
while RST = 1, SEN reset to 0

30.4.2 Set/Reset Conditions of Serial enables

Data name	Serial enable	Serial enable end	Enable condition (EN)	START	END
Command Address	AD_sen	AD_s_end	OTag[0]	0x12	0x1a
Command Write Data	WD_sen	WD_s_end	OTag[1]	0x26	0x36
Playback Lch FIFO output	PBL_sen	PBL_s_end	OTag[2]	0x3a	0x4a
Playback Rch FIFO output	PBR_sen	PBR_s_end	OTag[3]	0x4e	0x5e
Modem Out FIFO output	MO_sen	MO_s_end	OTag[4]	0x62	0x72
Command Echo Address	EAD_sen	EAD_s_end	R_EN & !IOSts[3] & ITag[0]	0x13	0x1a
Command Read Data	RD_sen	RD_s_end	R_EN & !IOSts[2] & ITag[1]	0x26	0x36
Record Lch FIFO input	RL_sen	RL_s_end	R_EN & P_EN[1] & !IOSts[11] & ITag[2]	0x3a	0x4a
Record Rch FIFO input	RR_sen	RR_s_end	R_EN & P_EN[1] & !IOSts[10] & ITag[3]	0x4e	0x5e
Modem In FIFO input	MI_sen	MI_s_end	R_EN & P_EN[3] & !IOSts[19] & ITag[4]	0x62	0x72
MIC In FIFO input	MIC_sen	MIC_s_end	R_EN & P_EN[4] & !IOSts[23] & ITag[5]	0x76	0x86

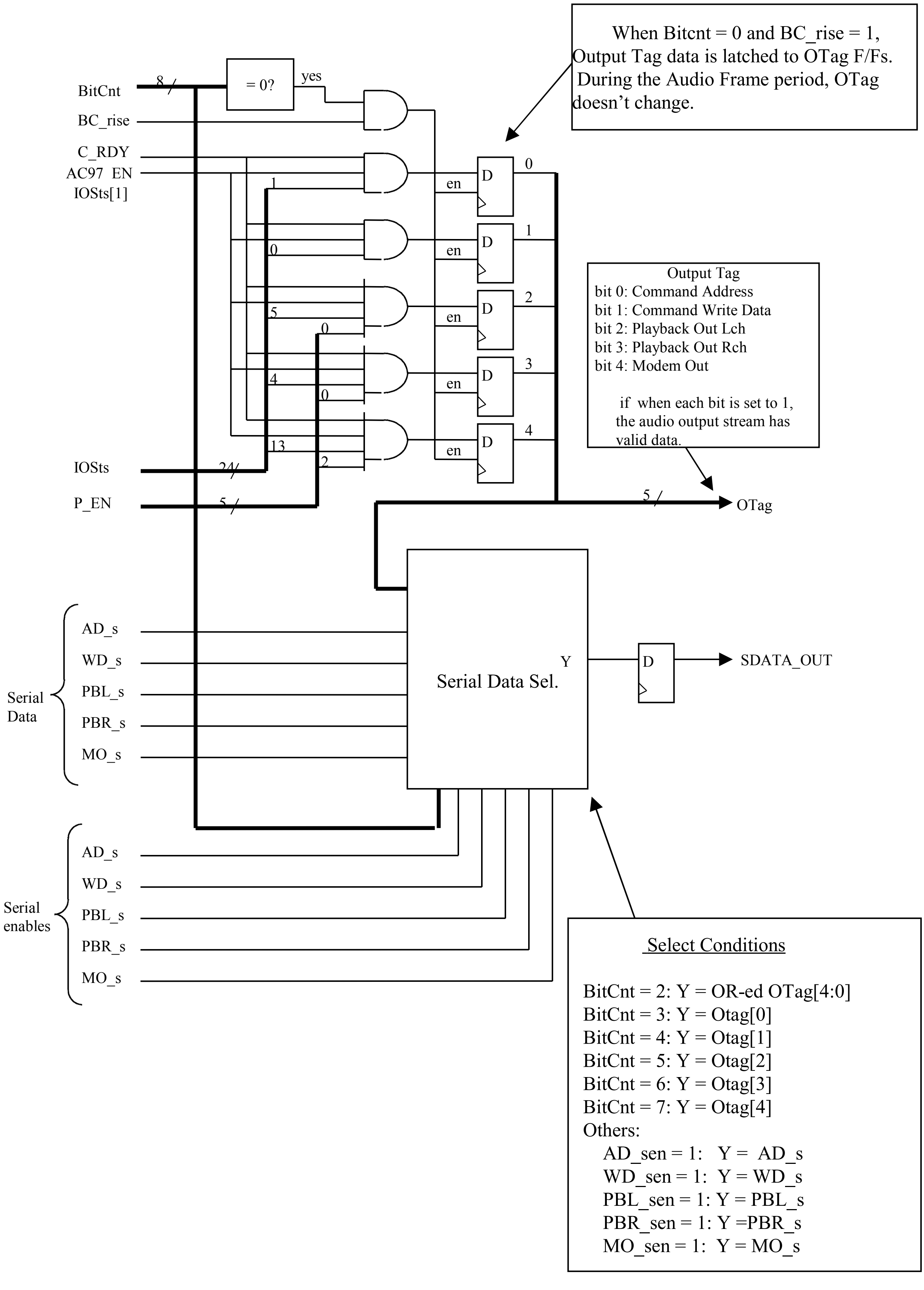
NOTE: R_EN = C_RDY & AC97_EN.
START and END are the BitCnt values.

30.5 AC’97 Serial Output Generator (SO_GEN.v)

This block generates AC’97 serial output data. The serial data output (SDATA_OUT) is directly connected to AC’97 chip and send digital audio streams to AC’97 chip. The MSBs of each FIFO output registers, Write command address, Write data are used for serial output data. The MSB of command address (AD_s), MSB of command write data (WD_s), MSB of 2channel composite PCM output stream (PBL_s and PBR_s) , and MSB of Modem Line Codec DAC input stream (MO_s) are multiplexed according to the Internal Serial Timing Generator (T_GEN.v).

Each outgoing streams has 20-bit sample resolution. However, this module supports only 16-bit output resolution, this block always stuffs all trailing non-valid bit positions (last 4 bit positions) with 0’s.

30.5.1 Block Diagram



When Bitcnt = 0 and BC_rise = 1, Output Tag data is latched to OTag F/Fs. During the Audio Frame period, OTag doesn't change.

Output Tag
bit 0: Command Address
bit 1: Command Write Data
bit 2: Playback Out Lch
bit 3: Playback Out Rch
bit 4: Modem Out

if when each bit is set to 1, the audio output stream has valid data.

Select Conditions

BitCnt = 2: Y = OR-ed OTag[4:0]
BitCnt = 3: Y = Otag[0]
BitCnt = 4: Y = Otag[1]
BitCnt = 5: Y = Otag[2]
BitCnt = 6: Y = Otag[3]
BitCnt = 7: Y = Otag[4]
Others:
AD_sen = 1: Y = AD_s
WD_sen = 1: Y = WD_s
PBL_sen = 1: Y = PBL_s
PBR_sen = 1: Y =PBR_s
MO_sen = 1: Y = MO_s

30.6 FIFO Input/Output Registers (F_REGS.v)

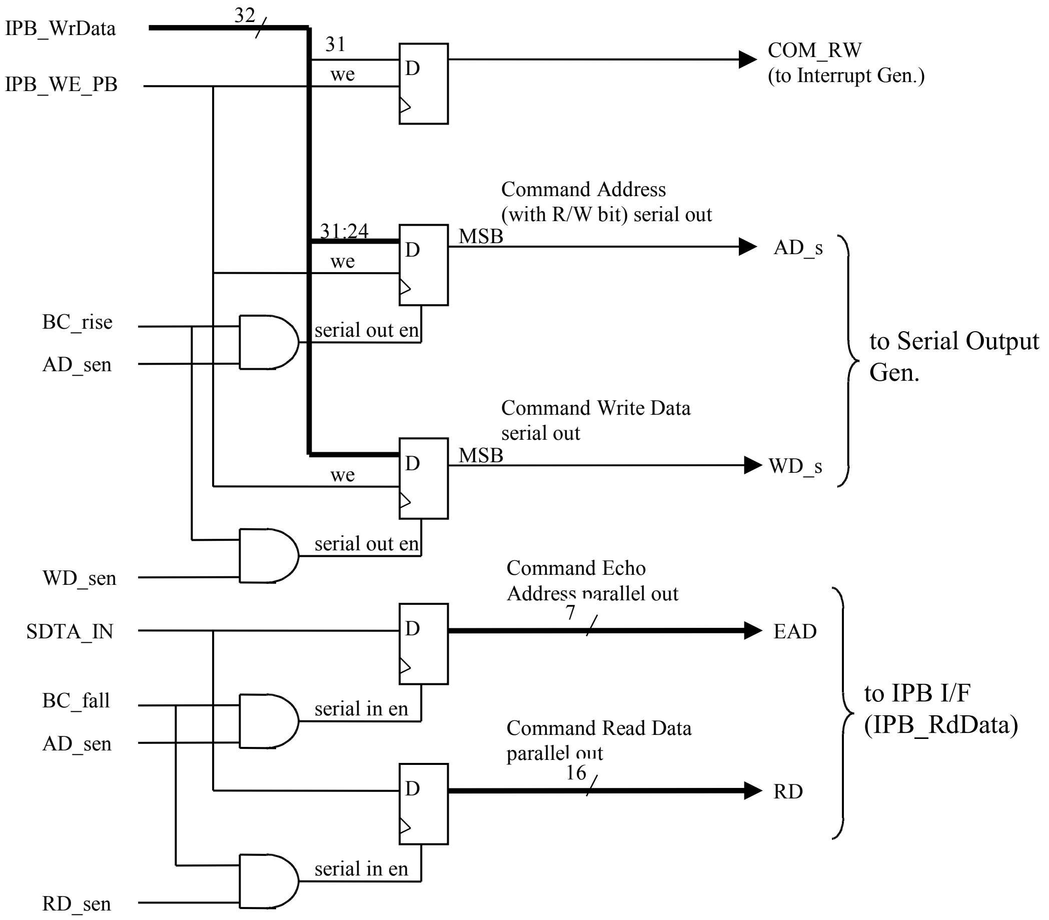
This block has fourteen 32-bit registers. Internal Peripheral Bus communicates with AC-link by using these registers. It is placed between Internal Peripheral Bus and FIFO or AC-link and FIFO, except some registers.

FIFO Input Registers are classified into two types. One is IPB to FIFO registers. Internal Peripheral Bus writes data to this register. The data must be written to the register, 32-bit simultaneously. Byte write is not supported. If FIFO is not full and this register contains 32-bit data, the data is written to FIFO. Another one is FIFO to AC-link registers. This registers is 32-bit shift register with serial input and parallel outputs. Serial input is connected with serial input of AC-link (SDATA_IN). After the serial transfer, if FIFO is not full, the data is written to FIFO, 32-bit simultaneously.

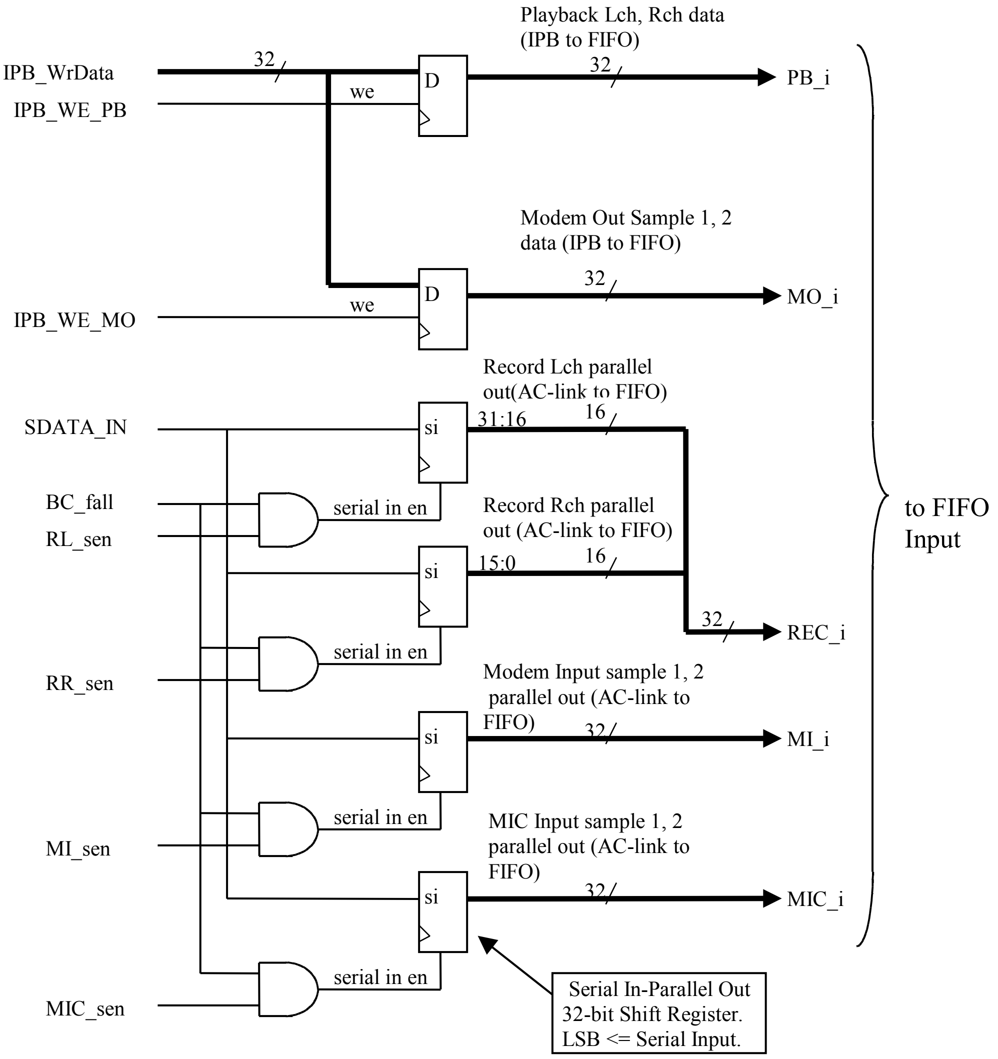
FIFO Output Registers are classified into two types. One is FIFO to IPB registers. Internal Peripheral Bus read data from this register, 32-bit simultaneously. After the read by Internal Peripheral Bus, if FIFO is not empty, FIFO writes 32-bit data to the register. Another one is FIFO to AC-link registers. This registers is 32-bit shift register with 32-bit parallel inputs and a serial output. Serial output is connected to serial output of AC-link (SDATA_OUT) via Serial Output Generator. After the serial data transfer, if FIFO is not empty, the FIFO writes 32-bit data to register using parallel inputs.

AC’97 command register, write data register, command echo register, and read data register are including FIFO Input/Output Registers for convenience' sake, however, these registers are directly connected with Internal Peripheral Bus and AC-link, not using FIFO.

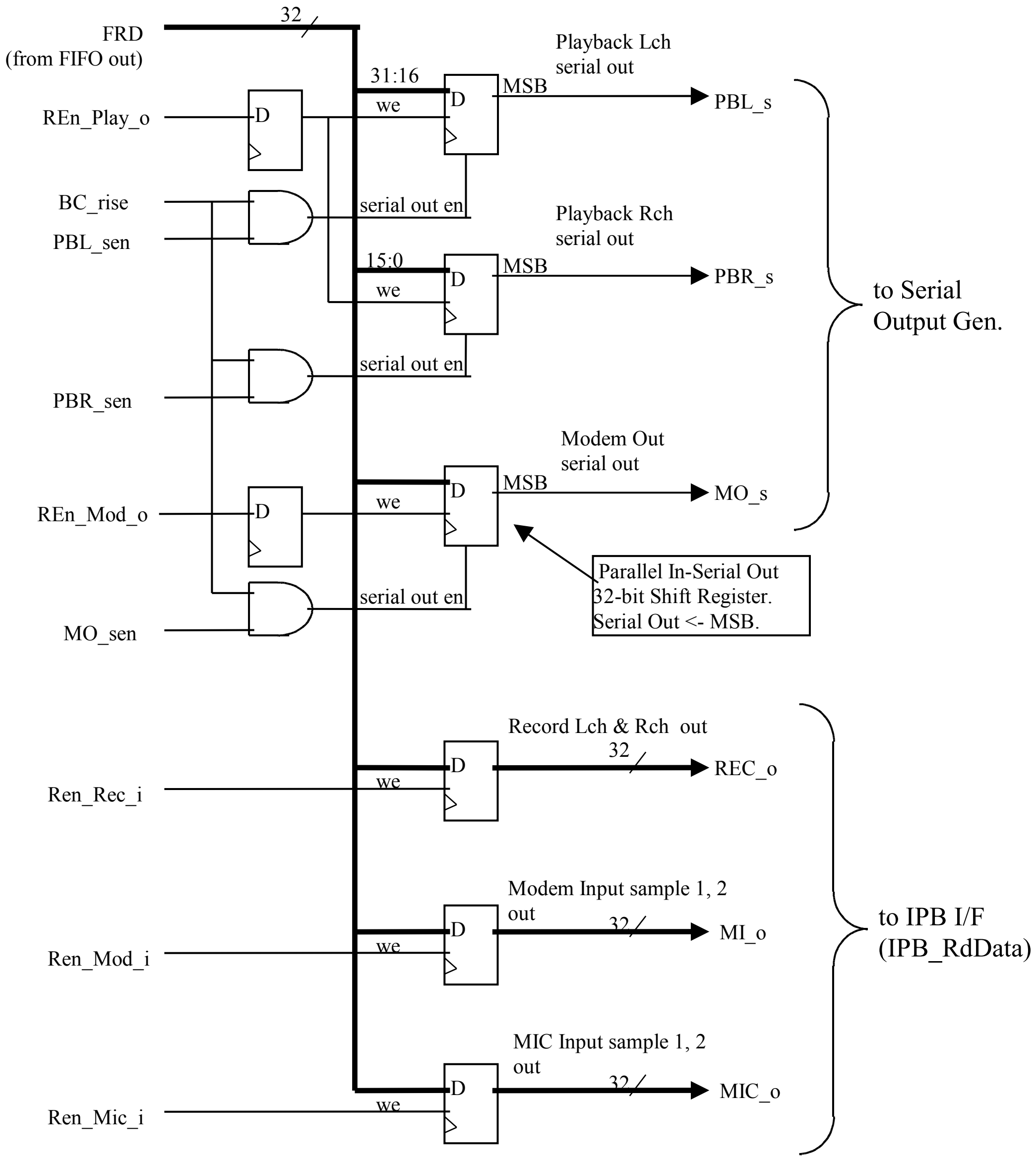
30.6.1 Block Diagram (AC’97 Commend R/W Registers)



30.6.2 Block Diagram (FIFO Input Registers)

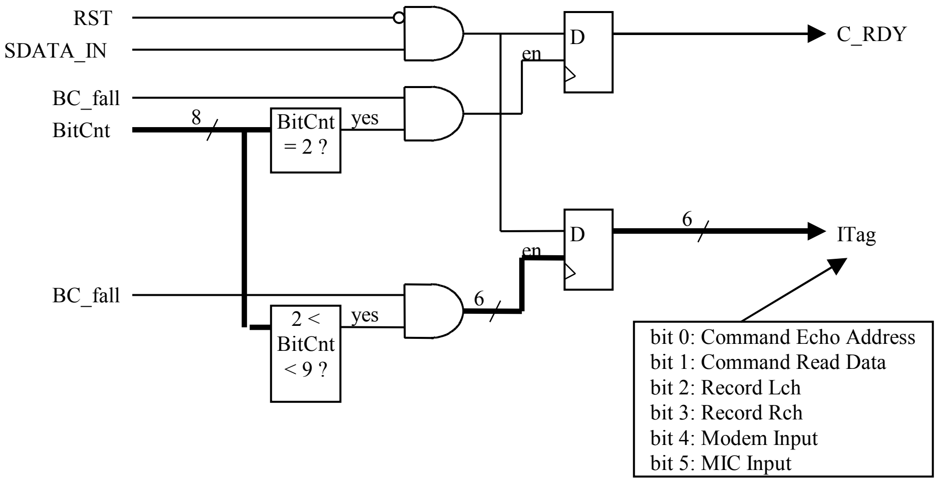


30.6.3 Block Diagram (FIFO Output Registers)



30.7 Read Input Tags (R_TAG.v)

This block reads Codec Ready bit and 6-bit Input Tag bit from AC'97 Serial Data Input (SDATA_IN).
While SYNC is asserted to 1, this block read tag bits. Codec Ready bit (stored in C_RDY register) indicates whether AC'97 is in the "Codec Ready" state or not. While it is 0, Serial Enable Generator is disabled. Input Tag bit (stored in ITag[5:0]) means each input audio slot contains valid data or not.



30.8 128 x 32 bit FIFO (F_AC97.v)

This block has 132 x 32 bit user configurable FIFO. The FIFO has one write port and one read port, independently. The FIFO is divided to 5 parts for Playback Out, Record In, Modem Out, Modem In, and MIC In. Each part has individually FIFO start address, end address, watermark, write pointer, and read pointer. Start address, end address, and watermark are able to be written or read from Internal Peripheral Bus. These values are stored to registers. The write pointer and read pointer are able to be only read from Internal Peripheral Bus. These registers are automatically increment when write to FIFO or read from FIFO.

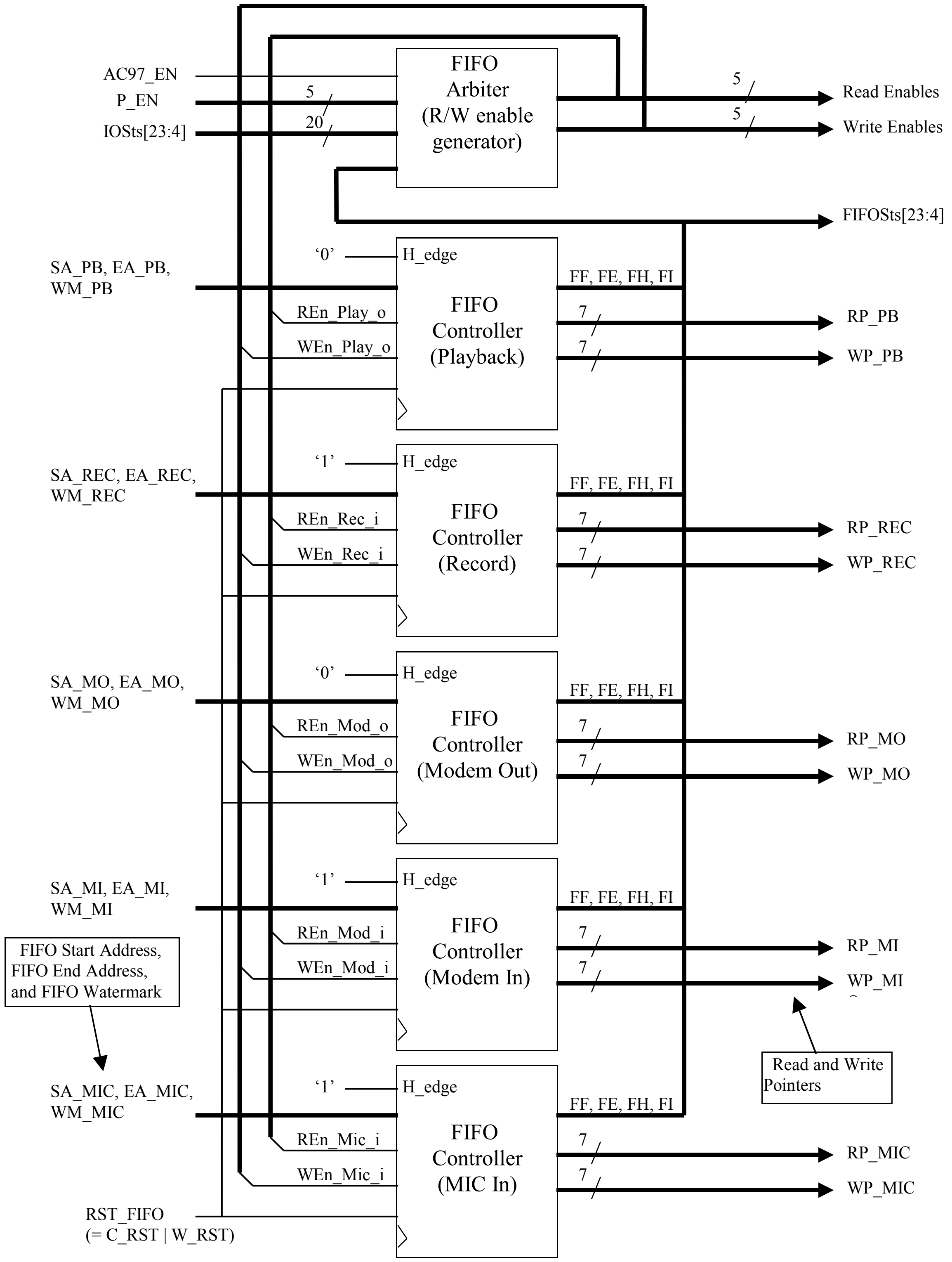
Each part has four flags, FF, FE, FH, and FI. FF and FE respectively indicates FIFO full and empty conditions. FH indicates half full condition (while a selected number of words is stored in memory.

In case of FIFO which is written by Internal Peripheral Bus, FI is asserted to 1 only one Imagine clock cycle, when a number of stored data is less than selected number (watermark). Playback Out and Modem Out fall under this type. In case of FIFO which is read from Internal Peripheral Bus, FI is asserted to 1 only one Imagine clock cycle, when a number of stored data is greater than selected number (watermark). Record In, Modem In, and MIC In are this type. FI flags are used for Interrupt Generator to generate interrupt signals. These flags are read from Internal Peripheral Bus at any time.

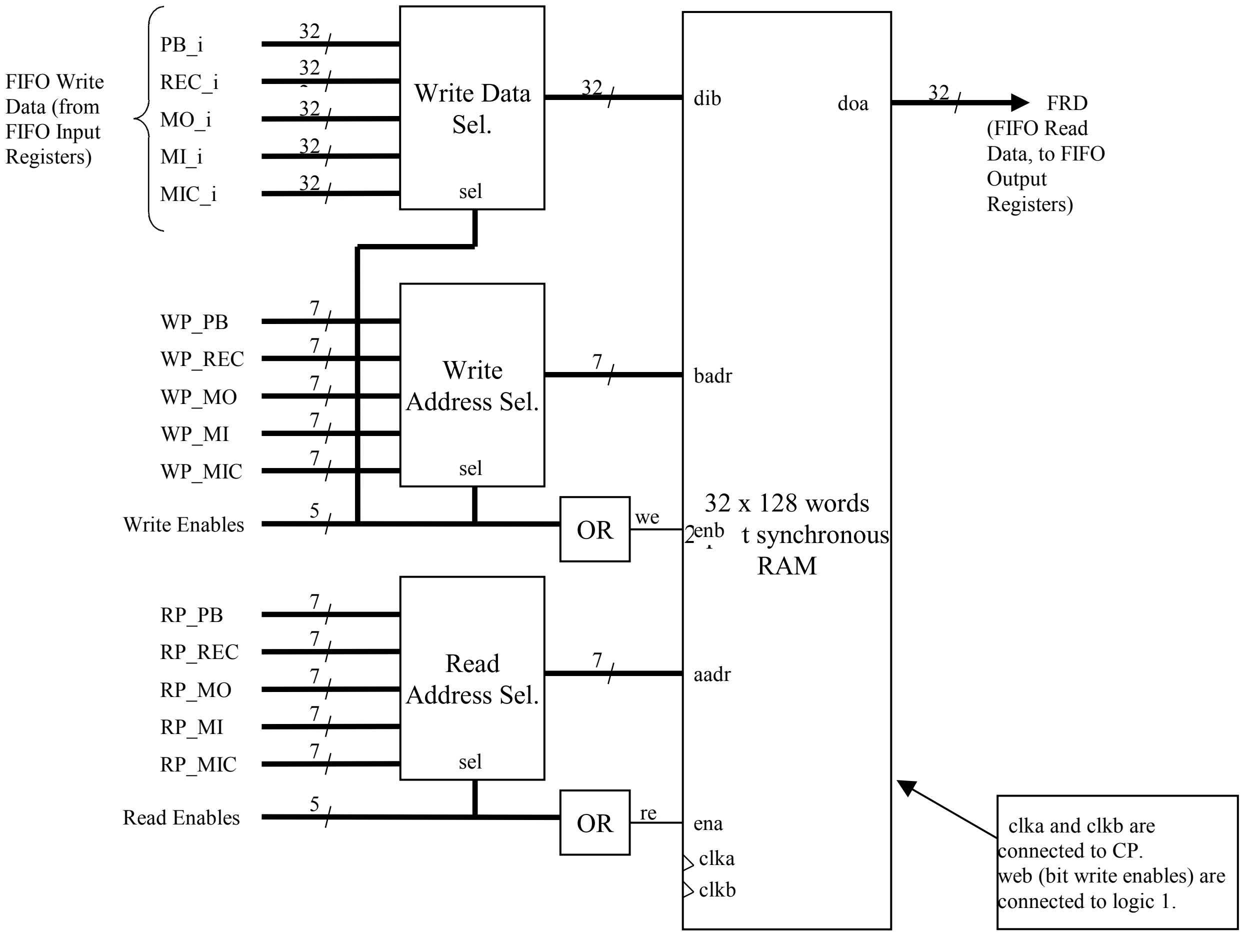
When accesses from Internal Peripheral Bus and AC-link simultaneously, the AC-link's access has priority. In case of write access is the same.

This block is completely synchronized with Imagine clock (not including delay cell or etc. to generate write pulse).

30.8.1 Block Diagram 1/2 (Controller & Arbiter block)



30.8.2 Block Diagram 2/2 (FIFO block)

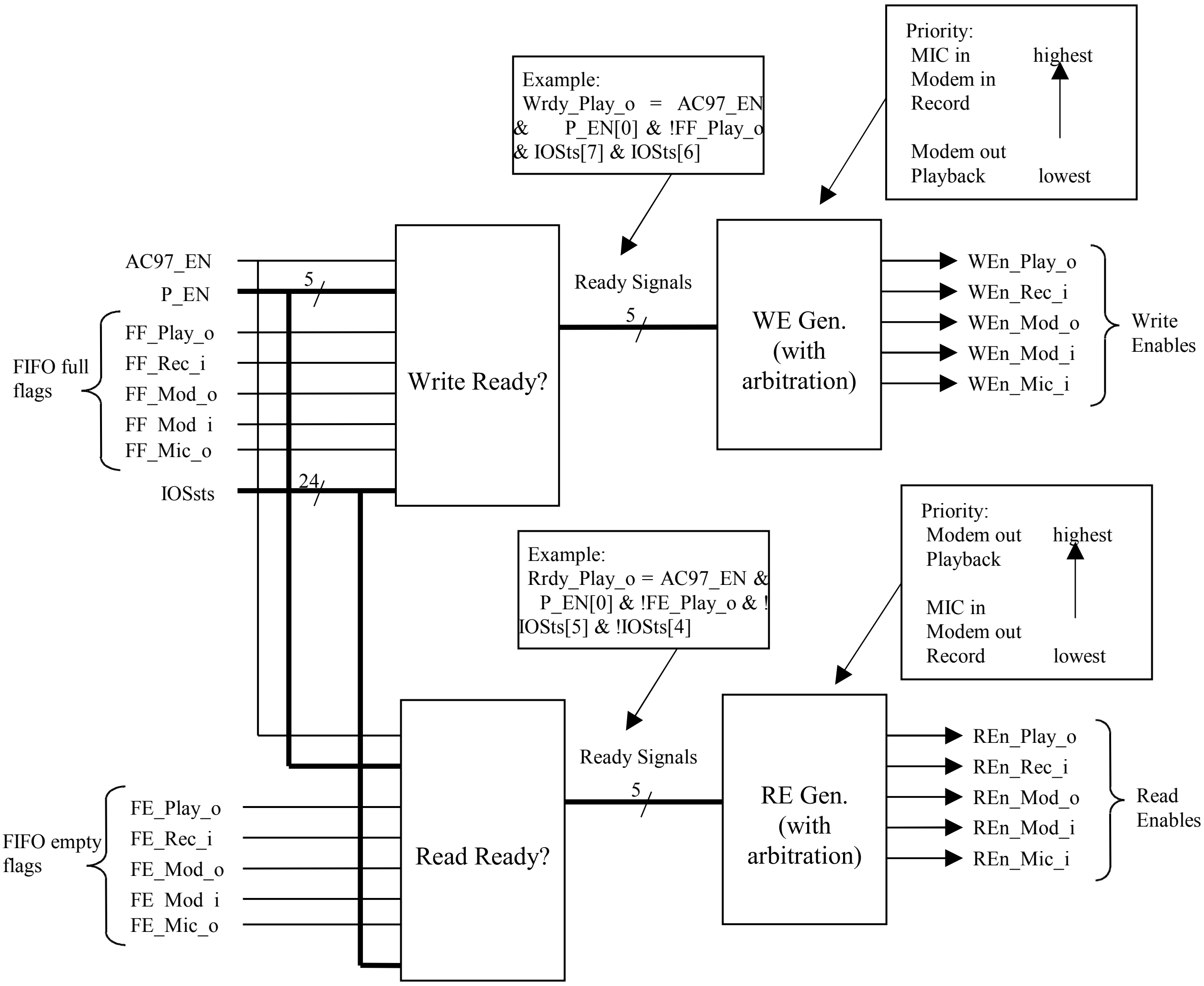

FIFO Write Data (from FIFO Input Registers)
PB_i
REC_i
MO_i
MI_i
MIC_i
32
Write Data Sel.
sel
WP_PB
WP_REC
WP_MO
WP_MI
WP_MIC
7
Write Address Sel.
Write Enables
5
OR
we
RP_PB
RP_REC
RP_MO
RP_MI
RP_MIC
Read Address Sel.
Read Enables
re
dib
badr
enb
aadr
ena
clka
clkb
doa
32 x 128 words 2-port synchronous RAM
FRD (FIFO Read Data, to FIFO Output Registers)
clka and clkb are connected to CP.
web (bit write enables) are connected to logic 1.

30.8.3 FIFO Arbiter (F_ABT.v)

This block arbitrates write or read accesses from FIFO input/output registers and generates write and read enable signals. The write enable signals are used for FIFO write access and clears I/O status registers' IH, IL bits. The read enable signals are user for FIFO read access and set I/O status registers' OH, OL bits.

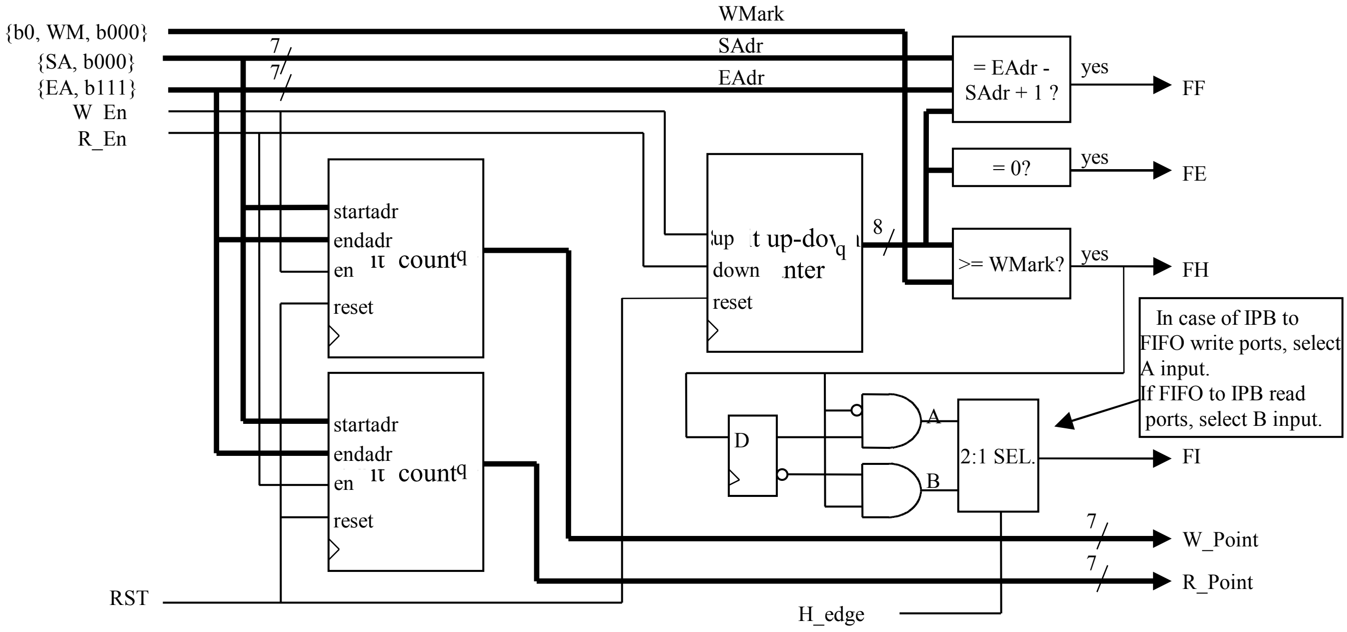
When accesses from Internal Peripheral Bus and AC-link simultaneously, the AC-link's access has priority. In case of write access is the same.

If IH and IL bit (I/O status register) is 1 and FF flag (FIFO status register) is 0, the arbiter asserts write enable to 1. Besides, if OH and OL bit (I/O status register) is both 0 and FE flag (FIFO status register) is 0, the arbiter asserts read enable to 1.



30.8.4 FIFO Controller (F_CTRL.v)

This block two 7-bit up counters and one 8-bit up-down counter. Two 7-bit up counters are used for FIFO write pointer (memory write address) and FIFO read pointer (memory read address). One 8-bit up-down counter counts a number of data which written into FIFO. Start and end address (higher 4 bit) are defined by FIFO control registers. Lower 3 bit of start address are filled with '0'. Lower 3 bit of end address are filled with '1'. The block diagram of this module is as follows.



In case of IPB to FIFO write ports, select A input.
If FIFO to IPB read ports, select B input.

30.9 Interrupt Generator (INT_GEN.v)

This module generates two interrupt lines (INT_R, INT_W). The interrupt lines are connected to the Interrupt Vector Generator via the Interrupt Router. When output FIFO half empty states (or AC’97 Control Write register empty) occurs, this module asserts INT_W to 1 while one Imagine clock cycle.
When input FIFO full empty states (or AC’97 Control Read register full) occurs, this module asserts INT_R to 1 while one Imagine clock cycle.
When some bit of Interrupt Enable Register (Interrupt Control register, bit 13:8) is set to 0, interrupts corresponding to the bit are disabled.
Interrupt conditions is assign to Interrupt Identify Register (Interrupt Control register, bit 5:0). This register is able to be read from Internal Peripheral Bus.

Conditions which INT_W is asserted to 1 are:

1. Command Write data empty and enable interrupt for this port (FIFOSts[0] & INT_EN[0])
2. Playback FIFO half empty and enable interrupt for this port (FIFOSts[4] & INT_EN[1])
3. Modem Out FIFO half empty and enable interrupt for this port (FIFOSts[12] & INT_EN[3])

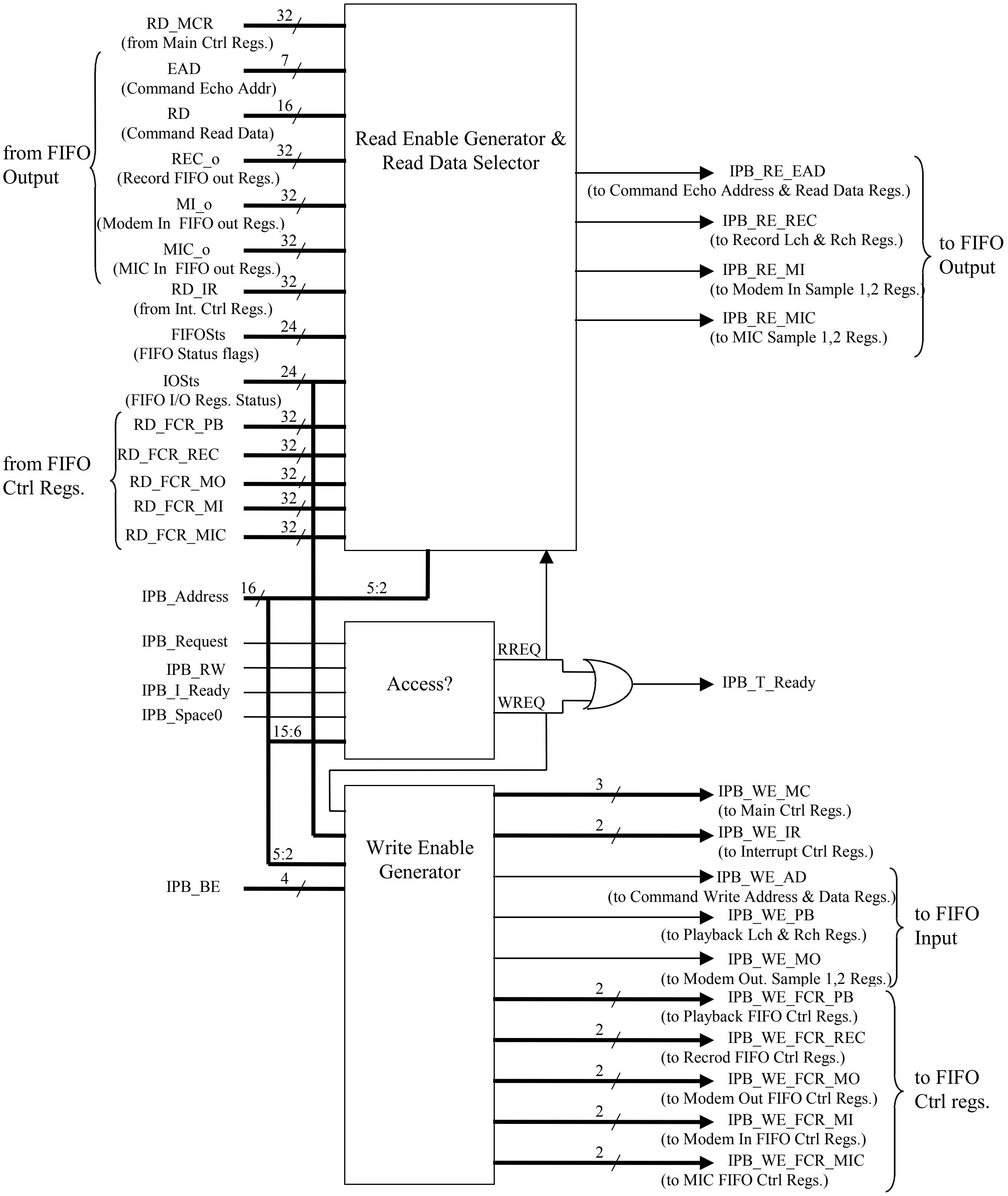
Conditions which INT_R is asserted to 1 are:

1. Command Read data full and enable interrupt for this port (FIFOSts[2] & INT_EN[0])
2. Record FIFO half full and enable interrupt for this port (FIFOSts[8] & INT_EN[2])
3. Modem In FIFO half full and enable interrupt for this port (FIFOSts[16]& INT_EN[4])
4. MIC In FIFO half full and enable interrupt for this port (FIFOSts[20] & INT_EN[5])

30.10 Internal Peripheral Bus I/F (ACIPBIO.v)

This module has Internal Peripheral Bus(IPB) interface giving direct access to the registers (include some FIFO Input/ Output registers) of this module. These are mapped on IPB address space 0 (IPB_Space0). 64 byte ranges are needed.
These registers are addressed directly by IPB_Address[10:2] from IPB and can be written or read at any time, except some FIFO data ports. In any case, write or read transfer to control registers always successfully terminate at first try. In case of accesses to FIFO data write ports, if this register is empty, only 1 Imagine clock cycle is needed to this transfer. If this register contains data, the Initiator must wait until this register is empty. In case of accesses to FIFO data read ports, if this register contains data, only 1 Imagine clock cycle is needed to this transfer. If this register is empty, the Initiator must wait until this register contains data. When Initiator writes to read only registers (e.g. Record FIFO port register), this module only set IPB_T_Ready to 1, vice versa..

Internal Peripheral Bus Interface



30.11 AC’97 Registers

30.11.1 AC’97 Controller base address

IPB_Address

15	14	13	12	11	10	9	8	7	6	5	4	3	2
AC’97 BASE (10b0000000101)										AC’97 register space			

* AC'97 Controller registers are on IPB_Space0.

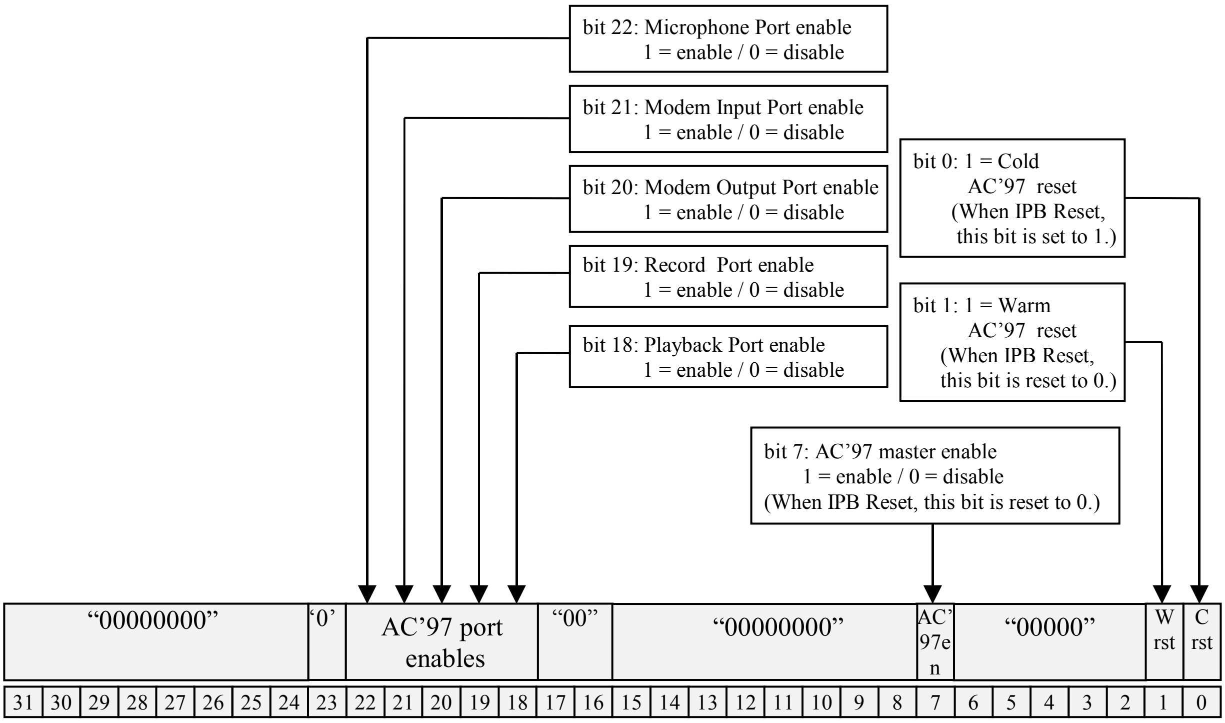
30.11.2 AC’97 Controller registers memory map

	31	30 – 24	23 – 16	15 – 0
0	AC’97 Main Control Register			
	AC’97 Command Control Register Access Port			
1	R W	Command Write Address	“00000000”	Command Write Data
	AC’97 Command Control Register Read Return Port			
2	‘0’	Command Address Echo	“00000000”	Command Read Data
3	AC’97 Playback Left channel Output Port			AC’97 Playback Right channel Output Port
4	AC’97 Record Left channel Input Port			AC’97 Record Right channel Input Port
5	AC’97 Modem Line Output Port, Sample 1			AC’97 Modem Line Output Port, Sample 2
6	AC’97 Modem Line Input Port, Sample 1			AC’97 Modem Line Input Port, Sample 2
7	AC’97 Microphone Input Port, Sample 1			AC’97 Microphone Input Port, Sample 2
8	AC’97 Interrupt Control Register			
9	AC’97 Input/ Output FIFO Status Register			
10	AC’97 FIFO Input / Output Port Status Register			
11	AC’97 Playback Output FIFO Control / Status Register			
12	AC’97 Record Input FIFO Control / Status Register			
13	AC’97 Modem Output FIFO Control / Status Register			
14	AC’97 Modem Input FIFO Control / Status Register			
15	AC’97 Microphone Input FIFO Control / Status Register			

IPB_Address[5:2]

30.11.3 AC’97 Main Control Registers

This registers totally controls Audio Codec ’97 Controller.
Bit 7 is master enable bit of the AC’97 controller (this module). When this bit is set to 0, all functions in this module are disabled.
Bit 22:18 are Port Enables. controls each ports of audio input/output streams. These registers enables or disable each ports of FIFO inputs/outputs.
Bit1:0 are reset control bit of the AC’97 analog chip. When bit 1 is 1, AC’97 controller generates a warm AC’97 reset state. When bit 0 is 1, AC’97 controller generates a Cold AC’97 reset state. If Internal Peripheral Bus asserts (IPB) RESET to 1, this bit is set to 1 (Cold Reset). The reset state is kept until clear the bit.
This registers can be write or read from Internal Peripheral Bus. Bit 22:18 are not initialized.



bit 22: Microphone Port enable
1 = enable / 0 = disable

bit 21: Modem Input Port enable
1 = enable / 0 = disable

bit 20: Modem Output Port enable
1 = enable / 0 = disable

bit 19: Record Port enable
1 = enable / 0 = disable

bit 18: Playback Port enable
1 = enable / 0 = disable

bit 0: 1 = Cold
AC’97 reset
(When IPB Reset,
this bit is set to 1.)

bit 1: 1 = Warm
AC’97 reset
(When IPB Reset,
this bit is reset to 0.)

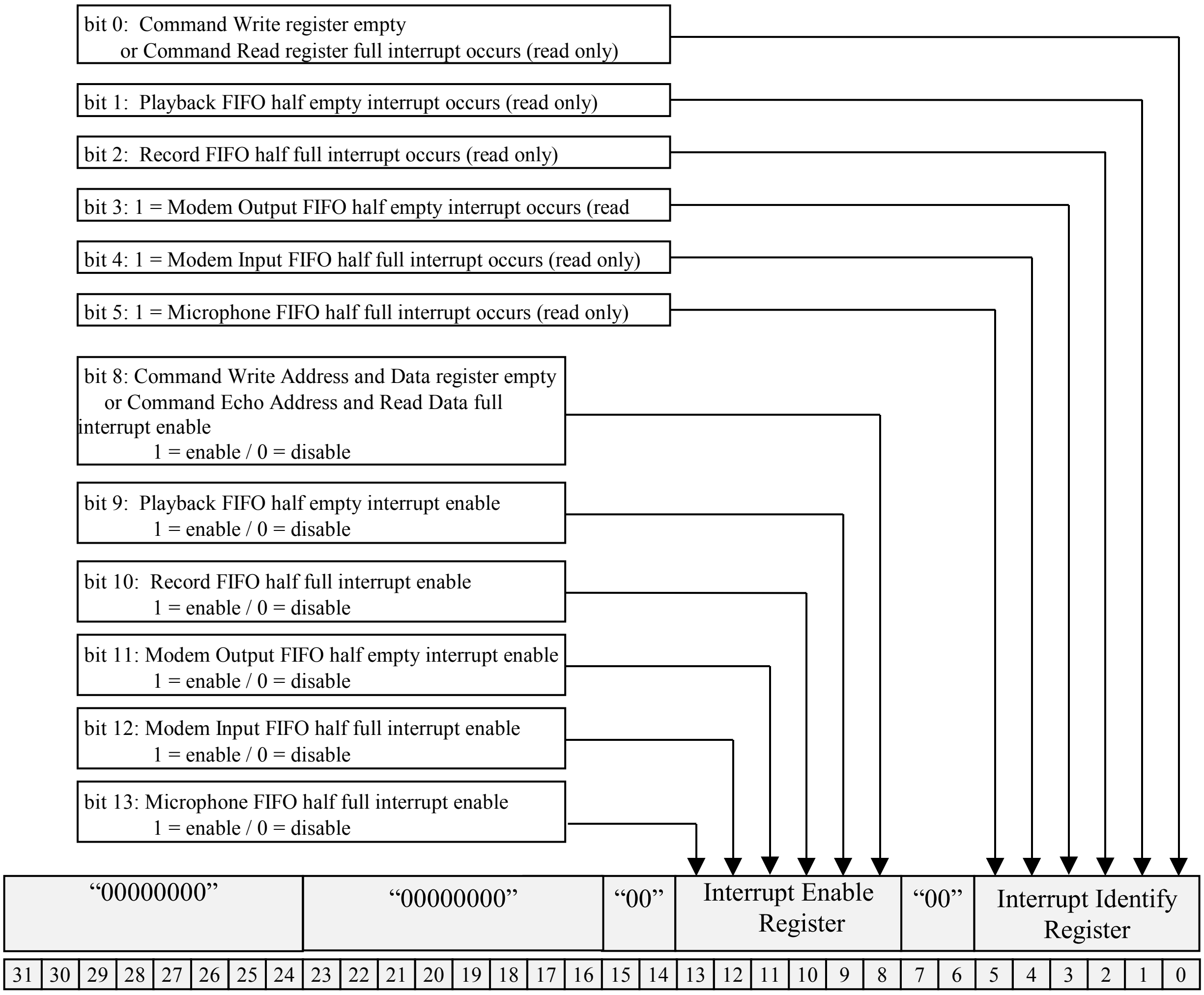
bit 7: AC’97 master enable
1 = enable / 0 = disable
(When IPB Reset, this bit is reset to 0.)

“00000000”								‘0’	AC’97 port enables					“00”		“00000000”								AC’97en	“00000”					W rst	C rst
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

30.11.4 Interrupt Control Register

This register controls two interrupt lines, INT_R and INT_W. When each parts of FIFO half empty/full state occurs, this module generates interrupt.

Bit 13:8 are interrupt enables for each FIFO input / output ports (including command write/read ports, which does not use the FIFO). If some bit of this enables are set to 0, the corresponding FIFO full or empty interrupts are disabled. These bit are not initialized. This register is able to be written or read from Internal Peripheral Bus at any time. In case of ports which is written by Internal Peripheral Bus, a change of bit has a effect after the next AC-link audio output frame. On the other hand, the ports which is read from Internal Peripheral Bus, it has a effect immediately. Bit 5:0 are interrupt identify registers for checking the interrupt status. This register indicates which port is in the interrupt state. These bit are able to be only read from Internal Peripheral Bus.



bit 0: Command Write register empty
or Command Read register full interrupt occurs (read only)

bit 1: Playback FIFO half empty interrupt occurs (read only)

bit 2: Record FIFO half full interrupt occurs (read only)

bit 3: 1 = Modem Output FIFO half empty interrupt occurs (read

bit 4: 1 = Modem Input FIFO half full interrupt occurs (read only)

bit 5: 1 = Microphone FIFO half full interrupt occurs (read only)

bit 8: Command Write Address and Data register empty
or Command Echo Address and Read Data full
interrupt enable
1 = enable / 0 = disable

bit 9: Playback FIFO half empty interrupt enable
1 = enable / 0 = disable

bit 10: Record FIFO half full interrupt enable
1 = enable / 0 = disable

bit 11: Modem Output FIFO half empty interrupt enable
1 = enable / 0 = disable

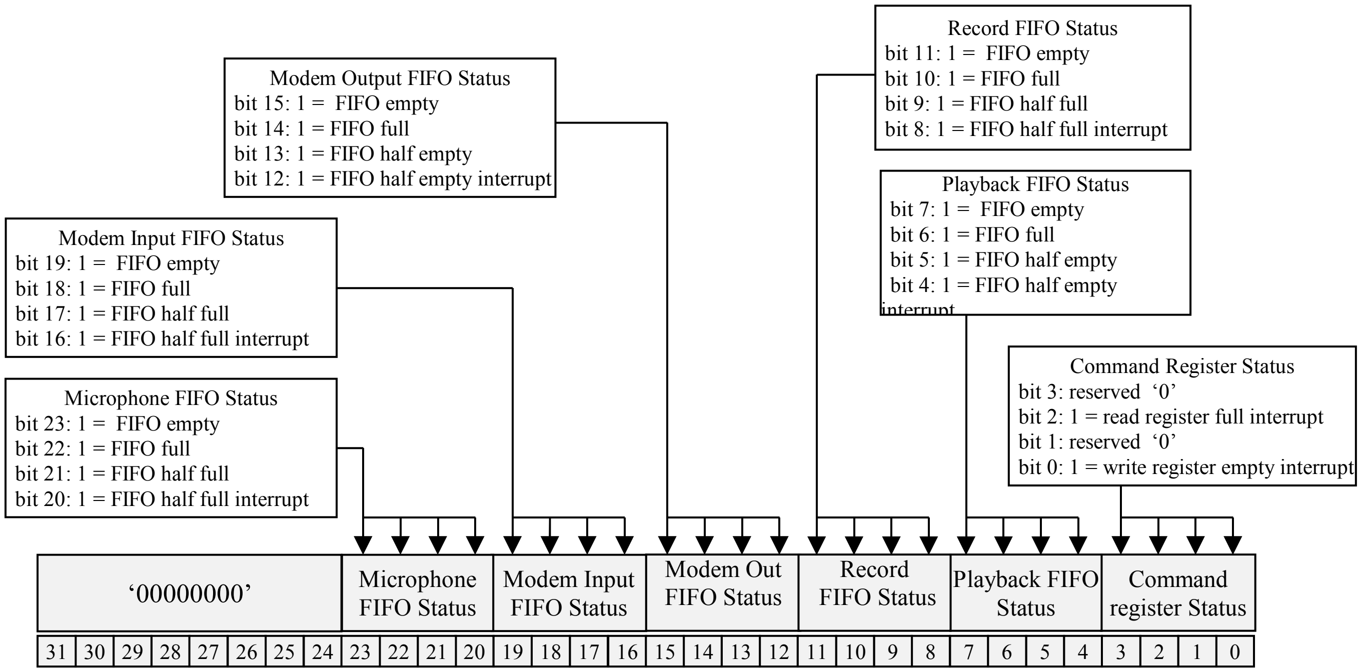
bit 12: Modem Input FIFO half full interrupt enable
1 = enable / 0 = disable

bit 13: Microphone FIFO half full interrupt enable
1 = enable / 0 = disable

31–24	23–16	15–14	13–8	7–6	5–0
“00000000”	“00000000”	“00”	Interrupt Enable Register	“00”	Interrupt Identify Register

30.11.5 Input/Output FIFO Status Register

This registers contains FIFO status. The FIFOs which is separated to five independent blocks have each four status flags, FIFO empty (FE), FIFO full (FF), FIFO half empty or full (FH), and FIFO interrupt (FI). Bit 2 is used for Command Read registers full interrupt (II). Bit 0 is used for Command Write registers empty interrupt (OI). Every FI bit and II, and OI are used for generating interrupts. This registers are able to be only read from Internal Peripheral Bus at any time. See subsection *2.4 FIFO Input/ Output Registers*, for more details.



Record FIFO Status
bit 11: 1 = FIFO empty
bit 10: 1 = FIFO full
bit 9: 1 = FIFO half full
bit 8: 1 = FIFO half full interrupt

Modem Output FIFO Status
bit 15: 1 = FIFO empty
bit 14: 1 = FIFO full
bit 13: 1 = FIFO half empty
bit 12: 1 = FIFO half empty interrupt

Playback FIFO Status
bit 7: 1 = FIFO empty
bit 6: 1 = FIFO full
bit 5: 1 = FIFO half empty
bit 4: 1 = FIFO half empty interrupt

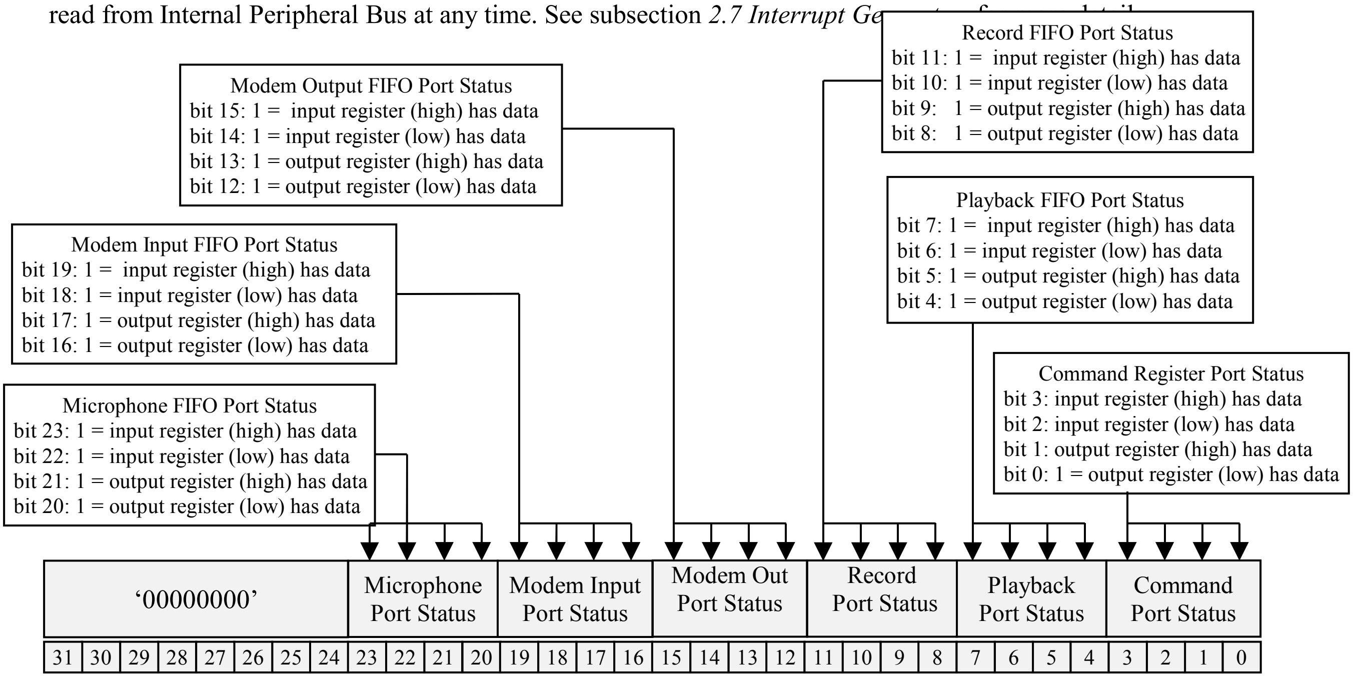
Modem Input FIFO Status
bit 19: 1 = FIFO empty
bit 18: 1 = FIFO full
bit 17: 1 = FIFO half full
bit 16: 1 = FIFO half full interrupt

Command Register Status
bit 3: reserved ‘0’
bit 2: 1 = read register full interrupt
bit 1: reserved ‘0’
bit 0: 1 = write register empty interrupt

Microphone FIFO Status
bit 23: 1 = FIFO empty
bit 22: 1 = FIFO full
bit 21: 1 = FIFO half full
bit 20: 1 = FIFO half full interrupt

30.11.6 FIFO Input/Output Port Status Register

This registers contains input/output port (FIFO Input/Output register) status. The five FIFO input or output ports have each four status flags, higher 16-bit of FIFO input register has data (IH), lower 16-bit of FIFO input register has data (IL), higher 16-bit of FIFO output register has data (OH), and lower 16-bit of FIFO output register has data (OL). Command registers’ status indicates the Input Output register has new data or not. These registers are directly connected with Internal Peripheral Bus and AC-link, not using FIFO. These registers able to be only read from Internal Peripheral Bus at any time. See subsection *2.7 Interrupt Ge*



Record FIFO Port Status
bit 11: 1 = input register (high) has data
bit 10: 1 = input register (low) has data
bit 9: 1 = output register (high) has data
bit 8: 1 = output register (low) has data

Modem Output FIFO Port Status
bit 15: 1 = input register (high) has data
bit 14: 1 = input register (low) has data
bit 13: 1 = output register (high) has data
bit 12: 1 = output register (low) has data

Playback FIFO Port Status
bit 7: 1 = input register (high) has data
bit 6: 1 = input register (low) has data
bit 5: 1 = output register (high) has data
bit 4: 1 = output register (low) has data

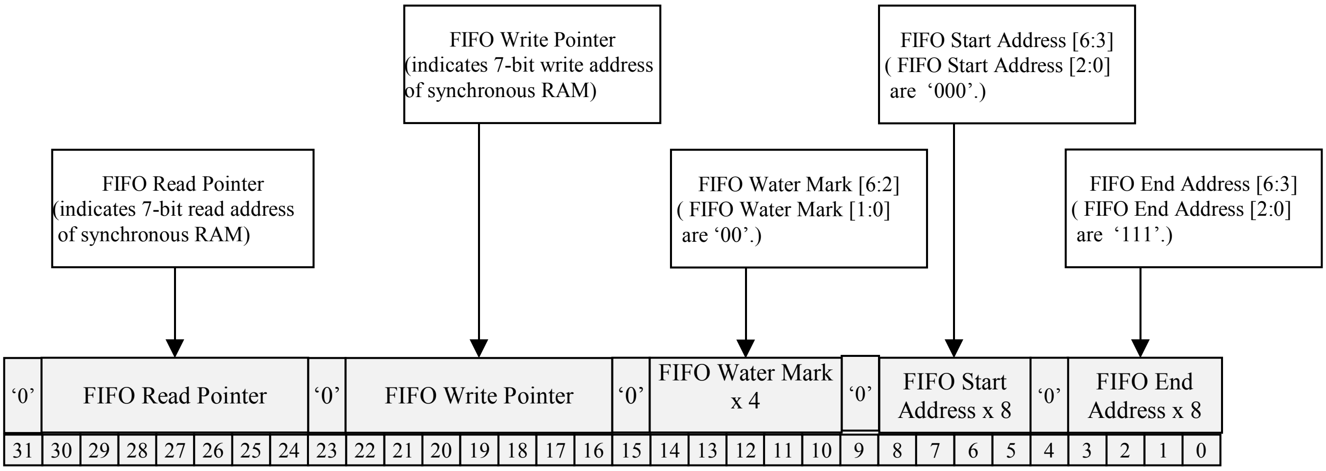
Modem Input FIFO Port Status
bit 19: 1 = input register (high) has data
bit 18: 1 = input register (low) has data
bit 17: 1 = output register (high) has data
bit 16: 1 = output register (low) has data

Command Register Port Status
bit 3: input register (high) has data
bit 2: input register (low) has data
bit 1: output register (high) has data
bit 0: 1 = output register (low) has data

Microphone FIFO Port Status
bit 23: 1 = input register (high) has data
bit 22: 1 = input register (low) has data
bit 21: 1 = output register (high) has data
bit 20: 1 = output register (low) has data

30.11.7 FIFO Control/Status Registers

This registers are used for definitions of the FIFO. This registers contains FIFO start address, FIFO end address, and FIFO watermark. Each FIFO block has one FIFO Control /Status Registers, amount to five registers. These values are able to be written or read from Internal Peripheral Bus at any time. Bit 30:24 are FIFO read pointer, Bit 22:16 are FIFO write pointer. These registers are able to be only read from Internal Peripheral Bus. See subsection *2.6.4 FIFO Controller*, for more details.



30.12 AC’97 Data Access Ports

In order to communicate with AC’97 digital interface, seven 32-bit registers are prepared. The command write/read registers are directly connected with Internal Peripheral Bus and with AC-link.
The others are connected via internal FIFO. These registers are read only or write only registers. The status of registers are able to read the FIFO port status registers.

30.12.1 AC’97 Control register access port

This registers is command write register, from Internal Peripheral Bus to AC-link. It is directly connected with Internal Peripheral Bus and AC-link. Bit 31 is R/W command bit. This value is used for Audio Output Frame Slot 1, bit 19. Bit 30:24 are 7-bit Command Control Address. This value is used for Audio Output Frame Slot 1, bit 18:12 (Control Register Index). Bit 15:0 are 16-bit command write data. When bit 31 is set to 1 (Read), bit 15:0 must be stuffed with 0’s. The value is used for Audio Output Frame Slot 2, bit 19:4 (Control Register Write Data). This registers is write only registers, and must be written 32-bit simultaneously.

30.12.2 AC’97 Control register read return port

This registers is command read register, from AC-link to Internal Peripheral Bus. It is directly connected with Internal Peripheral Bus and AC-link. Bit 30:24 are 7-bit Command Address echo from AC-link. This value is read from Audio Input Frame Slot 1, bit 18:12 (Control Register Index). Bit 15:0 are 16-bit command read data. The value is read from Audio Input Frame Slot 2, bit 19:4 (Control Register Read Data). This registers is read only registers, and must be read 32-bit simultaneously.

30.12.3 AC’97 Playback output port

This registers are PCM Playback write register, from Internal Peripheral Bus to AC-link. This register is connected to FIFO Input. Bit 31:16 are 16-bit PCM Playback left channel data. This value is used for Audio Output Frame Slot 3, bit 19:4. Bit 15:0 are 16-bit PCM Playback right channel data. This value is used for Audio Output Frame Slot 4, bit 19:4. This registers are write only registers, and must be written 32-bit simultaneously.

30.12.4 AC’97 Record input port

This registers are PCM Record read registers, from AC-link to Internal Peripheral Bus. This register is connected to FIFO output. Bit 31:16 are 16-bit PCM Record left channel data. This value is read from Audio Input Frame Slot 3, bit 19:4. Bit 15:0 are 16-bit PCM Record right channel data. This value is read from Audio Input Frame Slot 4, bit 19:4. This registers are read only registers, and must be read 32-bit simultaneously.

30.12.5 AC’97 Modem Line output port

This registers are Modem Line codec output write registers, from Internal Peripheral Bus to AC-link. This registers are connected to FIFO input. Bit 31:16 are 16-bit data (sample 1). Bit 15:0 are 16-bit data sample2. These values are used for Audio Output Frame Slot 5, bit 19:4. First, sample 1 is send to AC-link, after sample 2 is send in next Audio Output Frame. This registers are write only registers, and must be written 32-bit simultaneously.

30.12.6 AC’97 Modem Line input port

This registers are Modem Line codec input read registers, from AC-link to Internal Peripheral Bus. This registers are connected to FIFO output. Bit 31:16 are 16-bit data (sample 1). Bit 15:0 are 16-bit data sample2. These values are read from Audio Input Frame Slot 5, bit 19:4. First, sample 1 is read from AC-link, after sample 2 is read from next Audio Output Frame. This registers are read only registers, and must be read 32-bit simultaneously.

30.12.7 AC’97 Microphone input port

This registers are Microphone input read registers, from AC-link to Internal Peripheral Bus. This registers are connected to FIFO output. Bit 31:16 are 16-bit data (sample 1). Bit 15:0 are 16-bit data sample2. These values are read from Audio Input Frame Slot 6, bit 19:4. First, sample 1 is read from AC-link, after sample 2 is read from next Audio Output Frame. This registers are read only registers and must be read 32-bit simultaneously.