*W*isconsin's *I*nterleaved *M*ultithreaded *P*rocessor

Architecture Manual

Bryan Berns
Jacob Petranak
Jordan Wenner
Parikshit Narkhede
Suman Mamidi

Table of Contents

Introduction 3
WIMP Architecture 5
Register Map 5
Memory Map 11
Instruction Set 17
Instruction Encoding 37
Interrupts 39
Exceptions 41
Writing to Output Ports 43
Reading from Input Ports 45
Initializing a New Thread 46
Thread Synchronization 47
Video RAM and the VGA 49
References 50

Introduction

WIMP is an interleaved multi-threaded processor designed to exploit the data parallelism inherent in many multi-media DSP applications. It handles four physical threads that are interleaved to share the hardware resources. Threads are processed in a round robin scheme, and each thread is given a single cycle for processing.

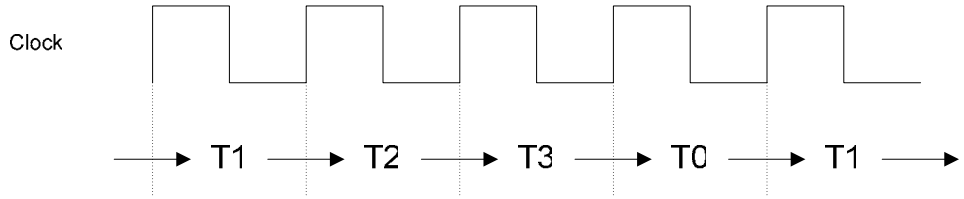

Figure 1: Round robin scheme for processing threads

The time elapsed before the same thread is processed again is called a machine cycle. In the case show in Figure 1, a machine cycle is equal to 4 clock cycles.

In addition, an instruction in each thread requires four clock cycles for completion as shown in Figure 2. The time elapsed from the instruction fetch to instruction commit for each thread constitutes a single thread cycle.

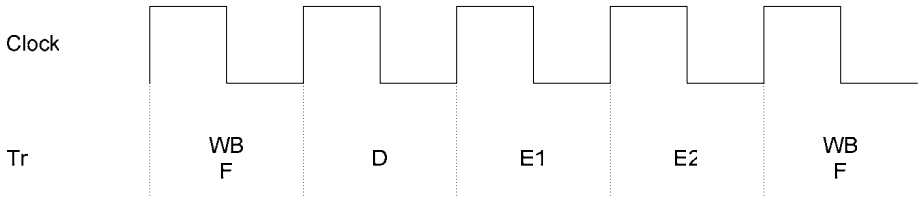

Figure 2: A single thread cycle

Figure 3 shows when each thread is processed and what stages each thread passes through. Having the number of hardware threads equal or greater than the latency eliminates any forms of data dependency, hence ensuring no stalls. This architecture is feasible when the target application set possesses enough parallelism to fill up the four physical threads. The round robin scheme ensures that all resources perform useful work as long as all the hardware threads are active.

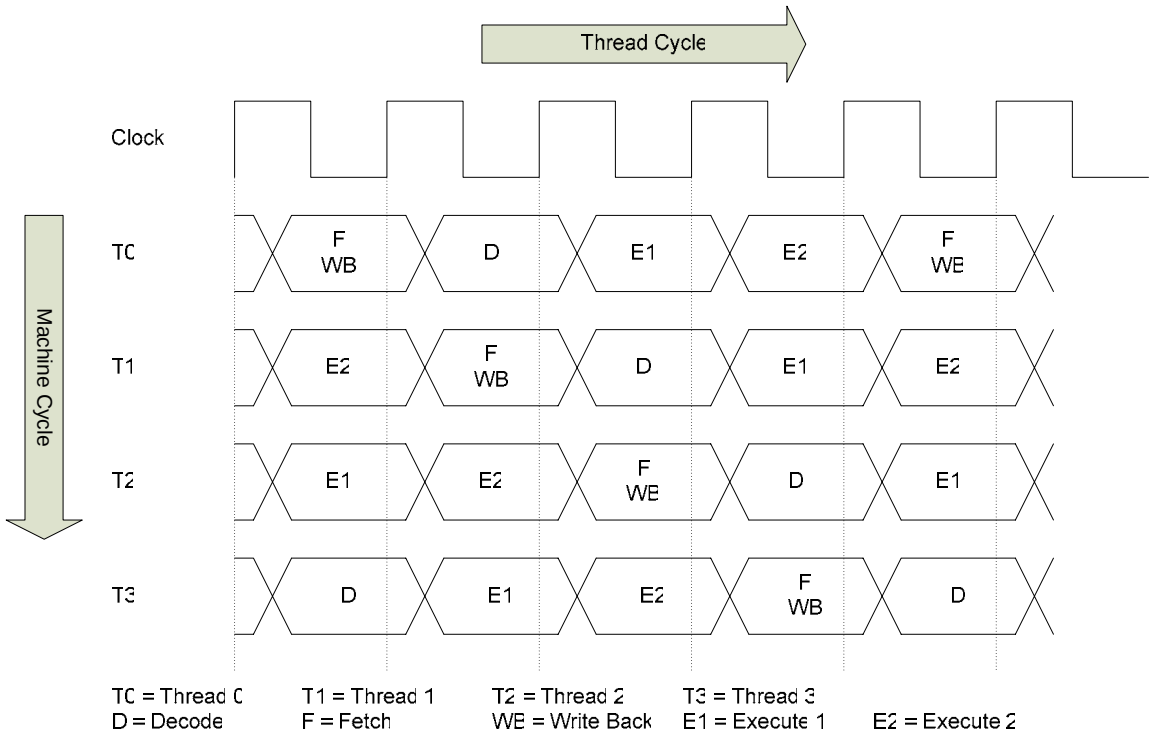

Figure 3: Interleaved multithreading

At reset, only thread 0 is active while all other threads are dead. At run time, any active thread can activate another dead thread. If the thread to be activated is already running, an exception occurs in the thread that is trying to activate the new thread. A thread cannot kill another thread; a thread can only kill itself. The behavior is undefined if two thread try to active a new thread in the same machine cycle.

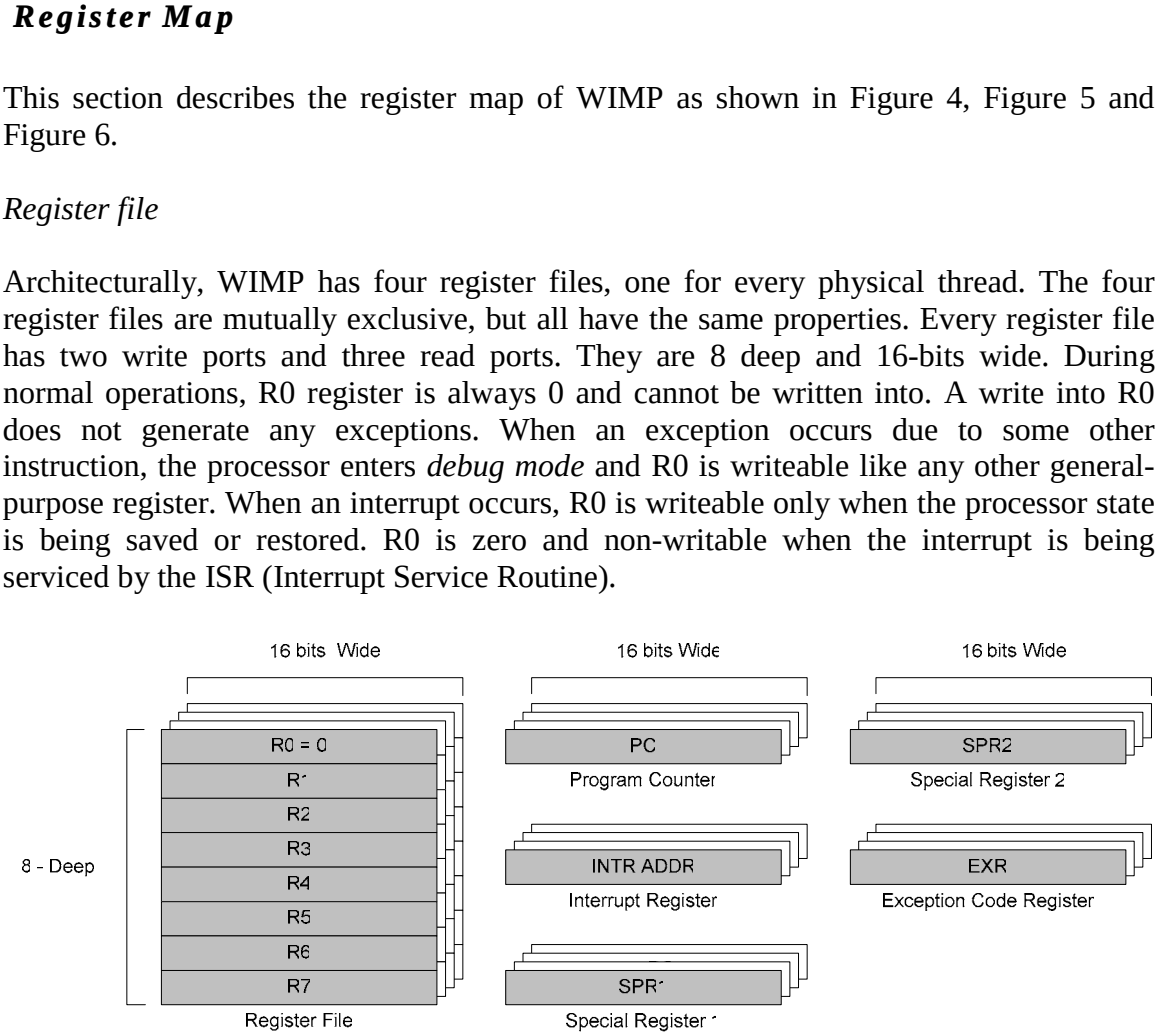

Figure 4: Register map

WIMP Architecture

Register Map

This section describes the register map of WIMP as shown in Figure 4, Figure 5 and Figure 6.

Register file

Architecturally, WIMP has four register files, one for every physical thread. The four register files are mutually exclusive, but all have the same properties. Every register file has two write ports and three read ports. They are 8 deep and 16-bits wide. During normal operations, R0 register is always 0 and cannot be written into. A write into R0 does not generate any exceptions. When an exception occurs due to some other instruction, the processor enters *debug mode* and R0 is writeable like any other general-purpose register. When an interrupt occurs, R0 is writeable only when the processor state is being saved or restored. R0 is zero and non-writable when the interrupt is being serviced by the ISR (Interrupt Service Routine).

Program Counter

Every thread has an independent program counter that is 16-bits wide. At reset, all program counters fetch from address 0x0000; however, only the program counter for thread 0 increments since only thread 0 is active at system reset. The program counter is incremented by 2 to fetch the new instructions.

Interrupt Register

A device interrupting the processor provides a starting address of the ISR on the interrupt bus. This address is latched into the interrupt register when the interrupt is acknowledged. The interrupt register is accessed by get.intr instruction that copies the contents of the interrupt register into the register file. WIMP has four such interrupt registers for each thread.

Exception Register

The exception register is a 16-bit register that stores the exception type when an exception occurs. Each bit of an exception register is assigned to an exception type. Table 1 shows definitions for each bit in the exception register. Every thread has its own exception register, which is mutually exclusive with the other exception registers.

Table 1: EXR Definition

BIT NUMBERS IN EXR	EXCEPTION
0	ILLEGAL_INSTRUCTION
1	UNALIGNED_ADDRESS
2	ILLEGAL_ADDRESS
3	INIT_THREAD
4	ILLEGAL_WAIT
5-15	Reserved

Special Register 1

SPR1 is a special register that records the address of the next instruction when an interrupt occurs. This is the return address of the program once an interrupt is serviced. The special register 1 is read by the get.spr1 instruction that copies the contents of the special register into the register file. Every thread has access to its own SPR1 that is 16-bits wide.

Special Register 2

SPR2 is a special register that records the address of the current instruction when an exception occurs. This register can be accessed by the get.spr2 instruction that copies the contents of the SPR2 into the register file. Every thread has its own SPR2 that is 16-bits wide.

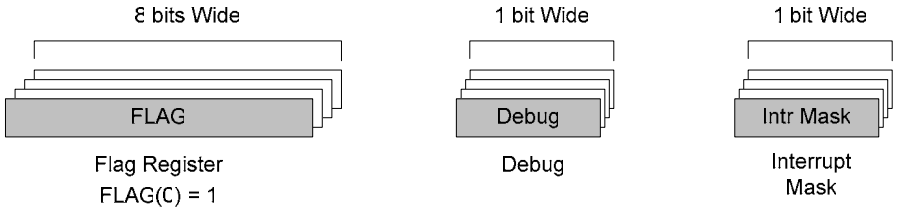

Figure 5: Register map (flag, debug, intrmask)

Flag Register

WIMP employs a limited form of predication applicable to the jump instructions. The 8-bit flag register is written by arithmetic instructions and compare instructions, and is read by the jump instructions. WIMP provides a flag register for every thread. In each of the flag registers, bit(0) is always 1. A write into the flag(0) does not cause any exceptions. The compare instructions can explicitly set the bits flag(1) through flag(7). Some arithmetic instructions listed in Table 2 implicitly set flag(7) and flag(6) when an overflow or underflow is detected.

Examples explaining the behavior of the flag register:

cmp.ne.16 $1, $2, 3 # compare the contents of
$1 and $2. If not equal, set flag(3) else reset flag(3)

cmp.ne.8 $1, $2, 3 # compare the contents of
$1 (lower byte) and $2 (lower byte) If not equal, set flag(3) else reset flag(3)
$1 (higher byte) and $2 (higher byte) If not equal, set flag(4) else reset flag(4)

add.16 $1, $2, $3 # Add $1 and $2 and store in $3
If result is > 2^{16}-1, set flag(6) else reset flag(6)

add.8 $1, $2, $3 # Add $1 (lower byte) and $2 (lower byte) and store in $3 (lower byte)
If result is > 2^{8}-1, set flag(6) else reset flag(6)
Add $1 (higher byte) and $2 (higher byte) and store in $3 (higher byte)
If result is > 2^{8}-1, set flag(7) else reset flag(7)

Table 2: Arithmetic instructions effecting the flag register

ARITHMETIC INSTRUCTION	WHEN
ADD.8	Result > 2^8 - 1
ADD.16	Result > 2^{16} - 1
ADDI.8	Result > 2^8 – 1
ADDI.16	Result > 2^{16} – 1
SUB.#	Result < 0
SUBI.#	Result < 0
AVG.#	Reset flags
SAVG.#	Result < 0
MUL	Reset flags
MAC	Result > 2^{16} - 1

Debug

The debug register is a one-bit register available independently for every thread. When the debug register is 1, register R0 of the register file is writable. When the debug register is 0, register R0 is not writable. The debug register can be explicitly set and reset using the set.debug instruction. In addition, debug is implicitly set when an interrupt or an exception occurs.

Interrupt Mask

The interrupt mask is one-bit register that cannot be accessed by any instruction. There is an interrupt mask register for every thread. External interrupts are not serviced when the interrupt mask is set. The interrupt mask is set implicitly when an interrupt or an exception occurs. This allows an exception to occur when an interrupt occurs, but not the other way round. The interrupt mask is implicitly reset after the system completes servicing the interrupt, restores the processor state, and returns to normal execution of the program.

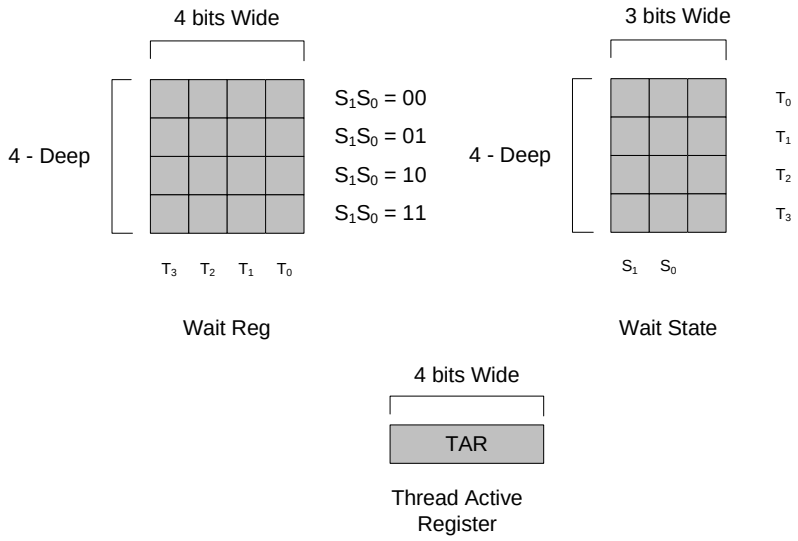

Figure 6: Register map (wait reg, wait state, and TAR)

Thread Active Register

The Thread Active Register (TAR) specifies which threads are active and which are dead. The bit corresponding to a thread is set when ever a thread is activated by the init instruction. A bit in TAR is reset whenever a threads corresponding bit dies, either due to an exception or a kill instruction. Table 3 shows the relation between the bits in the TAR and the physical threads.

Table 3: Relation between bits in TAR and physical threads

Bit	Thread
TAR[0]	Thread 0
TAR[1]	Thread 1
TAR[2]	Thread 2
TAR[3]	Thread 3

Wait Register

The wait register and the wait state work together to implement thread synchronization. The wait register is 4 deep; where each row corresponds to a synchronization point. Every row is 4-bits wide, corresponding to thread 0 through thread 3. The wait register is primarily addressed by the synchronizing point. The wait register is modified by wait instructions. The wait register can be read by the get.wait_reg instruction that copies the contents of the wait register into the register file.

Wait State

The wait state register is four deep; where each row corresponds to a thread. Every row is three bits wide. The first bit (ws) indicates if the thread is waiting to be synchronized. If so, the next two bits, S_0 and S_1, indicate the synchronizing point in the wait register that the thread is waiting on. The wait state is modified by the wait instruction and can be read using the get.ws instruction that copies the wait state register into the register file.

Memory Map

WIMP instructions address three sections of memory, which are: Instruction Memory, Data Memory and Video RAM. The instruction memory is accessed by the PC (program counter), loads, and stores; while only loads and stores access data memory. The Video RAM is a write only memory, accessible only to some special store instructions.

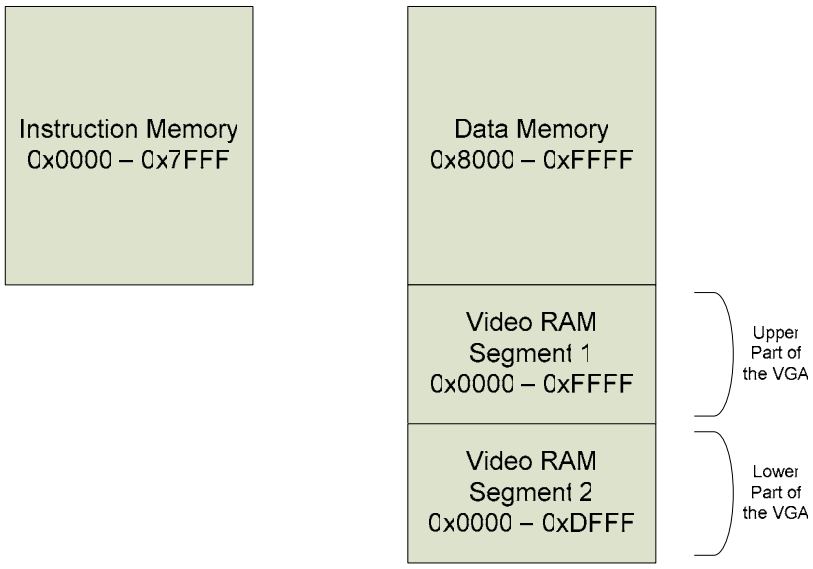

Figure 7: WIMP Memory map

WIMP has following restrictions that on the memory:

- 0x0000 – 0x3FFF cannot be written into (an exception will be generated otherwise)
- 0x4000 – 0x7FFF can be written into only by instructions that are present in the range 0x0000 – 0x3FFF (an exception will be generated otherwise)
- 0x8000 – 0xFFFF can be written and read by any instruction
- Video RAM is write only; accessed by special instructions only

The following table shows the memory map of the WIMP system.

Table 4: WIMP memory map

Table 4.1: WIMP Memory

Range (Hex)	Map
0000 – 7FFF	Program Memory
8000 – FFFF	Data Memory

Table 4.2: Program Memory

Range (Hex)	Map
0000 – 03FF	Thread 0 System Memory
0400 – 07FF	Thread 1 System Memory
0800 – 0BFF	Thread 2 System Memory
0C00 – 0FFF	Thread 3 System Memory
1000 – 3FFF	Operating System Memory
4000 – 7FFF	User Program Memory

Table 4.3: Thread0 System Memory

Range (Hex)	Map
0000 – 000F	Reserved
0010 – 00FF	Exception Handler
0110 – 01FF	Interrupt Handler – Init
0200 – 02FF	Interrupt Handler – Restore

Table 4.4: Thread 1 System Memory

Range (Hex)	Map
0400 – 040F	Reserved
0410 – 04FF	Exception Handler
0500 – 05FF	Interrupt Handler – Init
0600 – 06FF	Interrupt Handler – Restore

Table 4.5: Thread 2 System Memory

Range (Hex)	Map
0800 – 080F	Reserved
0810 – 08FF	Exception Handler
0900 – 09FF	Interrupt Handler – Init
0A00 – 0AFF	Interrupt Handler – Restore

Table 4.6: Thread 3 System Memory

Range (Hex)	Map
0C00 – 0C0F	Reserved
0C10 – 0CFF	Exception Handler
0D00 – 0DFF	Interrupt Handler – Init
0E00 – 0EFF	Interrupt Handler – Restore

Table 4.7: Operating System Memory

Range (Hex)	Map
1000 – 12FF	Monitor program
1300 – 13FF	Keyboard driver (ISR)
1400 – 14FF	SPAT driver
1500 – 2FFF	OS command routines

Table 4.8: Data Memory

Range (Hex)	Map
8000 – 8FFF	OS Data Memory
9000 – FFFF	User Data Memory

Table 4.9: OS Data Memory

Range (Hex)	Map
8000 – 803F	Keyboard buffer
8040 – 804F	Thread 0 command area
8050 – 805F	Thread 1 command area
8060 – 806F	Thread 2 command area
8070 – 807F	Thread 3 command area
8080 – 80FF	Thread 0 write buffer
8100 – 817F	Thread 1 write buffer
8180 – 81FF	Thread 2 write buffer
8200 – 827F	Thread 3 write buffer
8280 – 829F	Thread 0 dump memory
82A0 – 82BF	Thread 1 dump memory
82C0 – 82DF	Thread 2 dump memory
82E0 – 82FF	Thread 3 dump memory
8300 – 83FF	Thread 0 frame storage
8400 – 84FF	Thread 1 frame storage
8500 – 85FF	Thread 2 frame storage
8600 – 86FF	Thread 3 frame storage
8700 – 8FFF	System variable storage

Table 4.10: Keyboard Buffer

Range (Hex)	Map
8000 – 8001	Buffer ready
8002 – 803F	Buffer area

Table 4.11: Thread 0 command area

Range (Hex)	Map
8040 – 8041	Command
8042 – 8043	Status
8044 – 8045	Parameter 1
8046 – 8047	Parameter 2

8048 – 8049	Parameter 3

Table 4.12: Thread 1 command area

Range (Hex)	Map
8050 – 8051	Command
8052 – 8053	Status
8054 – 8055	Parameter 1
8056 – 8057	Parameter 2
8058 – 8059	Parameter 3

Table 4.13: Thread 2 command area

Range (Hex)	Map
8060 – 8061	Command
8062 – 8063	Status
8064 – 8065	Parameter 1
8066 – 8067	Parameter 2
8068 – 8069	Parameter 3

Table 4.14: Thread 3 command area

Range (Hex)	Map
8070 – 8071	Command
8072 – 8073	Status
8074 – 8075	Parameter 1
8076 – 8077	Parameter 2
8078 – 8079	Parameter 3

Table 4.15: Thread 0 Write Buffer

Range (Hex)	Map
8080 – 8081	Buffer ready
8082 – 80FF	Buffer area

Table 4.16: Thread 1 Write Buffer

Range (Hex)	Map
8100 – 8101	Buffer ready
8102 – 80FF	Buffer area

Table 4.17: Thread 2 Write Buffer

Range (Hex)	Map
8180 – 8181	Buffer ready
8182 – 81FF	Buffer area

Table 4.18: Thread 3 Write Buffer

Range (Hex)	Map
8200 – 8201	Buffer ready
8202 – 82FF	Buffer area

Table 4.19: Thread 0 Dump Memory

Range (Hex)	Map
8280 – 8281	PC/PC+2
8282	Predicate register
8284 – 8285	EXR
8286 – 8287	R1
8288 – 8289	R2
828A – 828B	R3
828C – 828D	R4
828E – 828F	R5
8290 – 8291	R6
8292 – 8293	R7
8294 – 8295	WAIT_REG
8296 – 8297	TAR
8298 – 8299	Thread id

Table 4.20: Thread 1 Dump Memory

Range (Hex)	Map
82A0 – 82A1	PC/PC+2
82A2	Predicate register
82A4 – 82A5	EXR
82A6 – 82A7	R1
82A8 – 82A9	R2
82AA – 82AB	R3
82AC – 82AD	R4
82AE – 82AF	R5
82B0 – 82B1	R6
82B2 – 82B3	R7
82B4 – 82B5	WAIT_REG
82B6 – 82B7	TAR
82B8 – 82B9	Thread id

Table 4.21: Thread 2 Dump Memory

Range (Hex)	Map
82C0 – 82C1	PC/PC+2
82C2	Predicate register
82C4 – 82C5	EXR
82C6 – 82C7	R1
82C8 – 82C9	R2
82CA – 82CB	R3
82CC – 82CD	R4
82CE – 82CF	R5
82D0 – 82D1	R6
82D2 – 82D3	R7
82D4 – 82D5	WAIT_REG

82D6 – 82D7	TAR
82D8 – 82D9	Thread id

Table 4.22: Thread 3 Dump Memory

Range (Hex)	Map
82E0 – 82E1	PC/PC+2
82E2	Predicate register
82E4 – 82E5	EXR
82E6 – 82E7	R1
82E8 – 82E9	R2
82EA – 82EB	R3
82EC – 82ED	R4
82EE – 82EF	R5
82F0 – 82F1	R6
82F2 – 82F3	R7
82F4 – 82F5	WAIT_REG
82F6 – 82F7	TAR
82F8 – 82F9	Thread id

Instruction Set

The processor incorporates a small subword parallel ISA aimed at multimedia DSP applications. Table 5 lists all the instructions present in the WIMP's ISA.

Table 5: WIMP instructions

Instruction	Function
ADD.#	RD ← RA + RB
SUB.#	RD ← RA - RB
AVG.#	RD ← (RA + RB) >> 1
SAVG.#	RD ← (RA - RB) >> 1
RSL.#	RD ← RA >> RB
RSA.#	RD ← {RA[MSB,...],RA} >> RB
LSL.#	RD ← RA << RB
ROT.#	RD ← ROTATE_LEFT(RA by RB)
CMP.EQ.#	FLAG[CB] ← (RA == RB) ? 1 : 0
CMP.GT.#	FLAG[CB] ← (RA > RB) ? 1 : 0
CMP.LT.#	FLAG[CB] ← (RA < RB) ? 1 : 0
CMP.NE.#	FLAG[CB] ← (RA != RB) ? 1 : 0
CMP.GE.#	FLAG[CB] ← (RA >= RB) ? 1 : 0
CMP.LE.#	FLAG[CB] ← (RA <= RB) ? 1 : 0
MAC.LO	RD[15:0] ←RD[15:0] + RA[7:0] x RB[7:0]
MAC.HI	RD[15:0] ← RD[15:0] + RA[15:8] x RB[15:8]
MUL.LO	RD[15:0] ←RA[7:0] x RB[7:0]
MUL.HI	RD[15:0] ← RA[15:8] x RB[15:8]
MIX.LO	INTERLEAVE
MIX.HI	INTERLEAVE
MUX.LO	INTERLEAVE
MUX.HI	INTERLEAVE
AND	RD ← RA AND RB
OR	RD ← RA OR RB
XOR	RD ← RA XOR RB
NOT	RD ←NOT RA

COPY.LO	RD[7:0] ←RA[7:0]
COPY.HI	RD[15:8] ← RA[15:8]
LW	RD ← M[RA]
LWI	RD ← M[RA], RA ←RA + 2
LWD	RD ←M[RA], RA ←RA – 2
SW	M[RB] ←RA
SWI	M[RB] ←RA, RA ←RA + 2
SWD	M[RB] ←RA, RA ←RA - 2
READ	READ DATA FROM PORT: IM
WRITE	OUTPUT RB TO PORT: IM
WAIT	SPECIFY SCHRONIZED THREADS
KILL	KILL CURRENTLY EXECUTING THREAD
INIT	INITIALIZE THREAD #IM AT MEMORY RA
JMPR	PC ←RA
JALR	R7 ←PC, PC ←RA
JMPI	PC ←PC + IMM
JALI	R7 ←PC, PC ←PC + IMM
LI.HI	LOAD IMM INTO RD[15:8]
LI.LO	LOAD IMM INTO RD[7:0]
JMP.SPR1	PC ← SPR1
GET.FLAG	RD[7:0] ← FLAG, RD[15:8] Cleared
GET.SPR1	RD ← SPR1
GET.SPR2	RD ← SPR2
GET.EXR	RD ← EXR, Clear EXR
GET.WAIT_REG	RD[7:0] ← WAIT_REG, RD[15:8] Cleared
GET.TAR	RD[3:0] ← TAR, RD[15:4] Cleared
GET.WS	RD[3:0] ← WS, RD[15:4] Cleared
GET.INTR	RD[3:0] ← INTR Register
GET.THREAD	RD[1:0] ← Thread ID
PUT.FLAG	FLAG ← RA[7:0]
PUT.SPR1	SPR1 ←RA
PUT.DEBUG	PUT IMM INTO DEBUG

PUT.INTRMASK	Masks/Unmasks the interrupt
SWFT	M[RB] ←RA, Store RA into an output port
SWSG	Store into Video RAM
SWSGI	Store into Video RAM, Increment address by 1
SWSGD	Store into Video RAM, Decrement address by 1
NOP	DOES ABSOLUTELY NOTHING

Notes: # can take either 8 or 16 to indicate the subword size. Example, ADD.8 will add the lower and higher bytes of RA and RB where as ADD.16 considers RA and RB as single 16-bit words.

1. ADD.#

ADD.8 $RA, $RB, $RD
ADD.16 $RA, $RB, $RD

The contents of register RA and register RB are added and written into register RD. If the 16-bit addition results in a value greater than 2^{16}-1, the result wraps around and flag(6) is set, other wise flag(6) is reset. If any of the two 8-bit additions result in a value greater than 2^{8}-1, the corresponding numbers wrap around; flag(6) is set if the lower byte addition overflows, reset otherwise. Flag(7) is set if the higher byte addition overflows, reset otherwise.

2. SUB.#

SUB.8 $RA, $RB, $RD
SUB.16 $RA, $RB, $RD

The contents of register RB are subtracted from the contents of register RA and written into register RD. If the 16-bit subtraction results in a value less than 0, the result wraps around and flag(6) is set, otherwise flag(6) is reset. If any of the two 8-bit subtractions result in a value less than 0, the corresponding numbers wrap around. Flag(6) is set if the lower byte subtraction underflows, reset otherwise; flag(7) is set if the higher byte subtraction underflows, reset otherwise.

3. AVG.#

AVG.8 $RA, $RB, $RD
AVG.16 $RA, $RB, $RD

The contents of register RA and register RB are added; the result is right shifted and written into register RD. Flag(6) is cleared if the instruction is AVG.16. Flag(7) and flag(6) are cleared if the instruction is AVG.8

4. SAVG.#

SAVG.8 $RA, $RB, $RD
SAVG.16 $RA, $RB, $RD

The contents of register RB are subtracted from the contents of register RA; the result is right shifted and written into register RD. If the 16-bit SAVG results in a value less than 0, the result wraps around and flag(6) is set, other wise flag(6) is reset. If any of the two 8-bit additions result in a value less than 0, then the corresponding numbers wrap around. Flag(6) is set if the lower byte SAVG underflows, reset otherwise; flag(7) is set if the higher byte SAVG underflows, reset otherwise.

5. RSL.#

RSL.8 $RA, $RB, $RD
RSL.16 $RA, $RB, $RD

The contents of register RA are right shifted with zero padding by the amount indicated in the lower order bits of register RB and the result is written into register RD. The flag register is unaffected.

6. RSA.#

RSA.8 $RA, $RB, $RD
RSA.16 $RA, $RB, $RD

The contents of register RA are right shifted with MSB extension by the amount indicated in the lower order bits of register RB and the result is written into register RD. The flag register is unaffected.

7. LSL.#

LSL.8 $RA, $RB, $RD
LSL.16 $RA, $RB, $RD

The contents of register RA are left shifted with zero padding by the amount indicated in the lower order bits of register RB and the result is written into register RD. The flag register is unaffected.

8. ROT.#

ROT.8 $RA, $RB, $RD
ROT.16 $RA, $RB, $RD

The contents of register RA are rotated left by the amount indicated in the lower order bits of register RB and the result is written into register RD. The flag register is unaffected.

9. CMP.EQ.#

CMP.EQ.8 $RA, $RB, $CB
CMP.EQ.16 $RA, $RB, $CB

The contents of register RA and register RB are compared for equality. If the result is true, the predicate bit indicated by $CB is set to 1, otherwise set to 0. The contents of RA and RB are unaffected.

10. CMP.GT.#

CMP.GT.8 $RA, $RB, $CB
CMP.GT.16 $RA, $RB, $CB

The predicate bit indicated by $CB in the instruction is set to 1 if the contents of register RA are greater than the contents of register RB, otherwise set to 0.

11. CMP.LT.#

CMP.LT.8 $RA, $RB, $CB
CMP.LT.16 $RA, $RB, $CB

The predicate bit indicated by $CB in the instruction is set to 1 if the contents of register RA are less than the contents of register RB, otherwise set to 0.

12. CMP.NE.#

CMP.NE.8 $RA, $RB, $CB
CMP.NE.16 $RA, $RB, $CB

The contents of register RA and register RB are compared for inequality. If the result is true, the predicate bit indicated by $CB is set to 1, otherwise set to 0. The contents of RA and RB are unaffected.

13. CMP.GE.#

CMP.GE.8 $RA, $RB, $CB
CMP.GE.16 $RA, $RB, $CB

The predicate bit indicated by $CB in the instruction is set to 1 if the contents of register RA are greater than or equal to the contents of register RB, otherwise set to 0.

14. CMP.LE.#

CMP.LE.8 $RA, $RB, $CB
CMP.LE.16 $RA, $RB, $CB

The predicate bit indicated by $CB in the instruction is set to 1 if the contents of register RA are less than or equal to the contents of register RB, otherwise set to 0.

15. MAC.LO

MAC.LO $RA, $RB, $RD
MAC.LO $RA, $RB, $RD

The lower bytes of register RA and RB are multiplied and added to the contents of RD. The result is written back into RD. If the MAC results in a value greater than 2^{16}-1, the result wraps around and flag(6) is set, otherwise flag(6) is reset.

16. MAC.HI

MAC.HI $RA, $RB, $RD
MAC.HI $RA, $RB, $RD

The higher bytes of register RA and RB are multiplied and added to the contents of RD. The result is written back into RD. If the MAC results in a value greater than 2^{16}-1, the result wraps around and flag(6) is set, otherwise flag(6) is reset.

17. MUL.LO

MUL.LO $RA, $RB, $RD
MUL.LO $RA, $RB, $RD

The lower bytes of register RA and RB are multiplied and written into RD. Flag(6) is reset at the end of this instruction.

18. MUL.HI

MUL.HI $RA, $RB, $RD
MUL.HI $RA, $RB, $RD

The higher bytes of register RA and RB are multiplied and written back into RD. Flag(6) is reset at the end of this instruction.

19. MIX.LO

MIX.LO $RA, $RB, $RD

The lower byte of register RA and the higher byte in register RB are interleaved and written into RD. The flag register is unaffected.

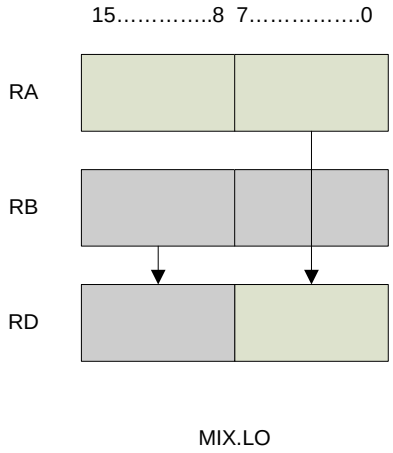

Figure 8: MIX.LO

20. MIX.HI

MIX.HI $RA, $RB, $RD

The higher byte of register RA and the lower byte of register RB are interleaved and written into RD. The flag register is unaffected.

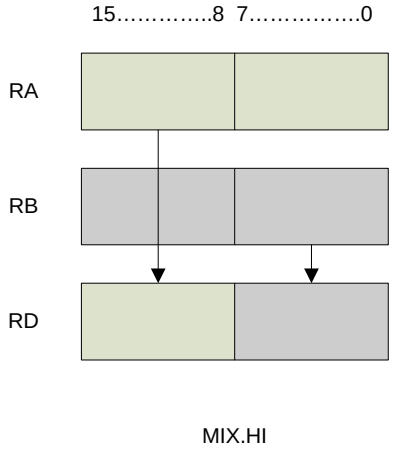

Figure 9: MIX.HI

21. MUX.LO

MUX.LO $RA, $RB, $RD

The lower byte of register RA and the higher byte in register RB are interleaved and swapped and then written into RD. The flag register is unaffected.

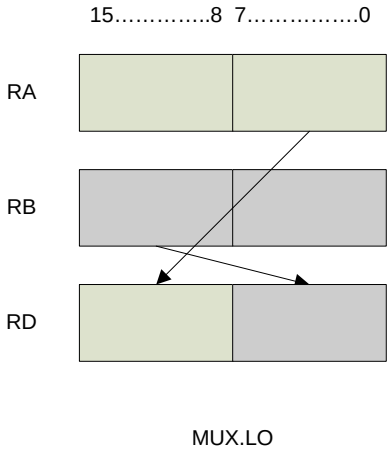

Figure 10: MUX.LO

22. MUX.HI

MUX.HI $RA, $RB, $RD

The higher byte of register RA and the lower byte in register RB are interleaved and swapped and then written into RD. The flag register is unaffected.

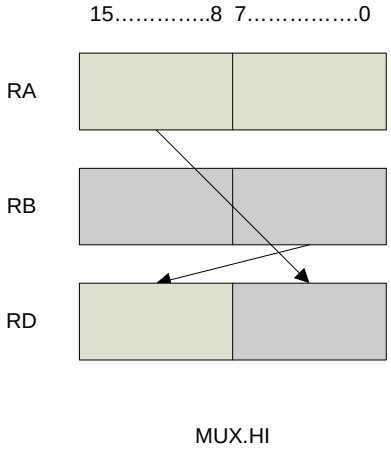

Figure 11: MUX.HI

23. AND

AND $RA, $RB, $RD

The contents of register RA and register RB are ANDed and written into register RD. The flag register is unaffected.

24. OR

AND $RA, $RB, $RD

The contents of register RA and register RB are ORed and written into register RD. The flag register is unaffected.

25. XOR

XOR $RA, $RB, $RD

The contents of register RA and register RB are XORed and written into register RD. The flag register is unaffected.

26. NOT

NOT $RA, $RD

The contents of register RA are complemented and written into register RD. The flag register is unaffected.

27. COPY.LO

COPY.LO $RA, $RD

The lower byte of the register RA is copied to the lower and higher byte of the register RD. The flag register is unaffected.

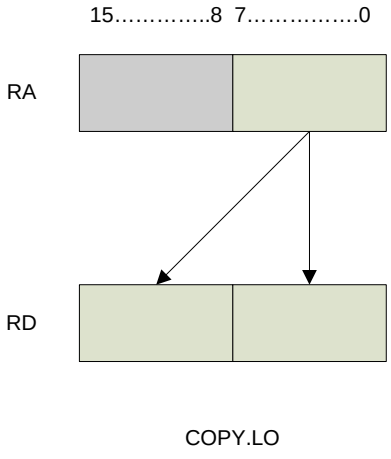

Figure 12: COPY.LO

28. COPY.HI

COPY.HI $RA, $RD

The higher byte of the register RA is copied to the lower and higher byte of the register RD. The flag register is unaffected.

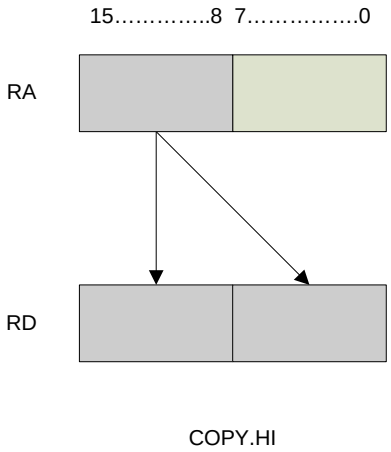

Figure 13: COPY.HI

29. LW

LW $RA, $RD

The processor loads the register RD with the data present in the location pointed to by the contents of register RA. If the memory address present in the register RA is not aligned to words, an UNALIGNED_ADDRESS exception occurs.

30. LWI

LWI $RA, $RD

The processor loads the register RD with the data present in the location, which is pointed to by the contents of register RA. If the memory address present in the register RA is not aligned to words, an UNALIGNED_ADDRESS exception occurs. In addition, the processor will increment the contents of RA by 2.

31. LWD

LWD $RA, $RD

The processor loads the register RD with the data present in the location, which is pointed to by the contents of register RA. If the memory address present in the register RA is not aligned to words, an UNALIGNED_ADDRESS exception occurs. In addition, the contents of RA are decremented by 2.

32. SW

SW $RA, $RB

The processor stores the contents of RB into the memory location specified by register RA. If the memory address present in the register RA is not aligned to words, an UNALIGNED_ADDRESS exception occurs. If the address in RA points to the system or program memory, an ILLEGAL_ADDRESS exception occurs.

33. SWI

SWI $RA, $RB

The processor stores the contents of RB into the memory location specified by register RA. If the memory address present in the register RA is not aligned to words, an UNALIGNED_ADDRESS exception occurs. If the address in RA points to the system or program memory, an ILLEGAL_ADDRESS exception occurs. In addition, the contents of RA are incremented by 2.

34. SWD

SWD $RA, $RB

The processor stores the contents of RB into the memory location specified by register RA. If the memory address present in the register RA is not aligned to words, an UNALIGNED_ADDRESS exception occurs. If the address in RA points to the system or program memory, an ILLEGAL_ADDRESS exception occurs. In addition, the contents of RA are decremented by 2.

35. LI.HI

LI.HI IMM8, $RD

The higher byte of the register RD is loaded with IMM8.

36. LI.LO

LI.LO IMM8, $RD

The lower byte of the register RD is loaded with IMM8.

37. READ

READ IMM3, $CB, $RD

The thread reads the input port indicated by IMM3. If the READ is successful, the predicate bit indicated by $CB is set and the value is written into RD. If the READ is unsuccessful, RD is left untouched and predicate bit indicated by $CB cleared.

38. WRITE

WRITE IMM3, $CB, $RD

The thread writes the contents of RD into the port indicated by IMM3. If the WRITE is successful, the predicate bit indicated by $CB is set, otherwise $CB is cleared.

39. WAIT

WAIT IMM[3], IMM[2], IMM[1], IMM[0], S_{10}

The processor supports a very simple form of thread synchronization using the WAIT instruction.

IMM[0] ➔ Thread 0
IMM[1] ➔ Thread 1
IMM[2] ➔ Thread 2
IMM[3] ➔ Thread 3

Say the instruction in thread 0 is
WAIT 1, 0, 1, 0, 0
When the processor comes across this instruction, it will stall thread 1 until it comes across a similar instruction in thread 3 that points to the synchronization point 0.

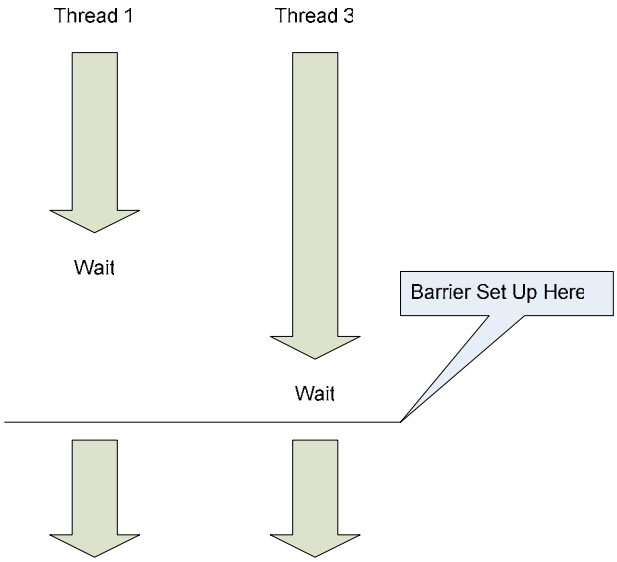

Figure 14: Illustrating thread synchronization

It wouldn't matter if thread 3 had reached the barrier point before. Thread 3 will stall until thread 1 reaches the same instruction. Both the WAIT instructions can be in different places in the memory.

40. KILL

KILL

The KILL instruction unconditionally kills the hardware thread. The KILL instruction does not have any parameters. A thread can be killed only from within itself using the KILL instruction. Once the thread is killed, the PC does not increment and the hardware thread will not respond to any interrupts. The thread can be brought up only from another thread or system reset.

41. INIT

INIT $RA, IMM2

During normal working, a thread can initiate another hardware thread with the INIT instruction.

Example,
Thread 2 has the instruction,

INIT R5, 3

The processor checks if thread 3 is already active. If so, an INIT_THREAD exception occurs in thread 2 while thread 3 continues as is. If thread 3 is not active, the PC for thread 3 is loaded with the contents of R5. Thread 3 will start in the next machine cycle. Once thread 3 is active, it can be killed only be the KILL instruction present in its instruction stream. Thread 2 will have no more control over thread 3.

Initializing a thread comes into effect in the next machine cycle. During this period, if another thread tries to initialize the same thread, the behavior is unpredictable.

If all the threads are killed, the processor halts. The processor can be activated only with the system reset. There should be at least one active thread at all times.

42. JMPR

JMPR $RA, CB

The program control jumps to the location present in the register RA if the bit defined by CB is 1, otherwise normal program execution proceeds. The jump is unconditional if CB is specified to 0. If the contents of register RA are not word aligned, an UNALIGNED_ADDRESS exception occurs. If an attempt is made to jump to the data area of the memory, an ILLEGAL_ADDRESS exception occurs.

43. JALR
JALR $RA, CB

The program control jumps to the location present in the register RA if the bit defined by CB is 1, otherwise normal program execution proceeds. The program counter + 2 is stored in R7. The jump is unconditional if CB is specified to 0. If the contents of register RA are not word aligned, an UNALIGNED_ADDRESS exception occurs. If an attempt is made to jump to the data area of the memory, an ILLEGAL_ADDRESS exception occurs.

44. JMPI

JMPI IMM8, CB

The program control jumps to the location indicated by the sum of sign-extended IMM8 and the PC if the bit defined by CB is 1, otherwise normal program execution proceeds. The jump is unconditional if CB is specified to 0. If the contents of register RA are not word aligned, an UNALIGNED_ADDRESS exception occurs. If an attempt is made to jump to the data area of the memory, an ILLEGAL_ADDRESS exception occurs.

45. JALI

JALI IMM8, CB

The program control jumps to the location indicated by the sum of sign-extended IMM8 and the PC if the bit defined by CB is 1, otherwise normal program execution proceeds. The program counter + 2 is stored in R7. The jump is unconditional if CB is specified to 0. If the contents of register RA are not word aligned, an UNALIGNED_ADDRESS exception occurs. If an attempt is made to jump to the data area of the memory, an ILLEGAL_ADDRESS exception occurs.

46. GET.FLAG

GET.FLAG $RD

Copies the 8-bit flag register (predicate register) into the lower byte of register RD. The higher byte of register RD is cleared.

47. GET.SPR1

GET.SPR1 $RD

Copies the SPR1 register (that holds the address to jump to after an interrupt is serviced) into register RD.

48. GET.SPR2

GET.SPR2 $RD

Copies the contents of SPR2 register (that holds the address of the instruction which has caused an exception) to the register indicated by RD.

49. GET.EXR

GET.EXR $RD

Copies the EXR register (that holds all the exceptions that have occurred for the instruction) to register RD. In addition the EXR register is cleared.

50. GET.WAIT_REG

GET.WAIT_REG $RD

Copies the 8-bit WAIT_REG register (synchronization register) into the lower byte of register RD. The higher byte of register RD is cleared.

51. GET.TAR

GET.TAR $RD

Copies the 4-bit TAR register (thread active register) into the lower four bits of register RD. The upper twelve bits of register RD are cleared.

52. GET.WS

GET.WS $RD

Copies the 4-bit WS register (holds which threads are currently waiting) into the lower four bits of register RD. The upper twelve bits of register RD are cleared.

53. GET.INTR

GET.INTR $RD

Copies the contents of INTR_ADDR register (holds address to jump to when interrupt occurs) to register RD.

54. GET.THREAD

GET.THREAD $RD

Gets the current thread id and places it in $RD. The upper fourteen bits of register RD are cleared.

55. PUT.FLAG

PUT.FLAG $RA

Loads the flag register with the lower byte of register RA.

56. PUT.SPR1

PUT.SPR1 $RA

Loads the SPR1 register with the contents of register RA.

57. PUT.DEBUG

PUT.DEBUG IMM1

Writes the value specified in the IMM field into the 1-bit DEBUG register. When the IMM field is 1, R0 becomes writable. When the immediate field is 0, R0 is cleared and cannot be written into.

58. PUT.INTRMASK

PUT.INTRMASK IMM1

Writes the specified value into the interrupt mask. If the IMM1 value is 1, the mask is set at the end of the instruction. Interrupts will not be acknowledged till the interrupt mask is cleared. If the IMM1 value is 0, the interrupt mask is cleared at the end of the instruction, which allows interrupts to be acknowledged. Interrupts are not acknowledged when this instruction is being executed, irrespective of whether the interrupt mask is being set or reset.

59. JMP.SPR1

JMP.SPR1

Loads the PC with the contents of the SPR1 register. In addition, the intr_mask register is cleared signifying that the interrupt has been serviced and the processor is ready to accept another interrupt.

60. NOP

NOP

Does absolutely nothing

61. ADDI.#

ADD.8 $RA, IMM3, $RD
ADD.16 $RA, IMM3, $RD

The contents of register RA are added to 3-bit IMM3 and written into register RD. If the 16-bit addition results in a value greater than 2^{16}-1, the result wraps around and flag(6) is set, otherwise flag(6) is reset. If any of the two 8-bit additions result in a value greater than 2^{8}-1, then the corresponding numbers wrap around. Flag(6) is set if the lower byte

addition overflows, reset otherwise; flag(7) is set if the higher byte addition overflows, reset otherwise.

62. SUBI.#

SUBI.8 $RA, IMM3, $RD
SUBI.16 $RA, IMM3, $RD

3-bit IMM3 is subtracted from the contents of register RA and written into register RD. If the 16-bit subtraction results in a value less than 0, the result wraps around and flag(6) is set, otherwise flag(6) is reset. If any of the two 8-bit subtractions result in a value less than 0, then the corresponding numbers wrap around. Flag(6) is set if the lower byte subtraction underflows, reset otherwise; flag(7) is set if the higher byte subtraction underflows, reset otherwise.

63. SWFT

SWFT $RA, $RB, $RD

The contents of register RB are stored into the address pointed by register RA. Additionally, the contents of register RB are stored into the output port indicated by RD. If the memory address present in the register RA is not aligned to words, an UNALIGNED_ADDRESS exception occurs. If the address in RA points to the system or program memory, an ILLEGAL_ADDRESS exception occurs. . If the write into the port is successful, the predicate bit 6 is set, otherwise predicate bit 6 is cleared. The store is unaffected by the status of the write into the output port.

64. SWSG

SWSG $RA, $RB, $RD

This special instruction allows writes into the video RAM. Concatenating the two lower order bits of the contents of register RD and the contents of register RB compute the effective address for the store. The contents of register RA are stored into this effective address. The contents of register RD can be 1 or 2. The store is cancelled if RD is 0. No exceptions are generated.

65. SWSGI

SWSGI $RA, $RB, $RD

This special instruction allows writes into the video RAM. Concatenating the two lower order bits of the contents of register RD and the contents of register RB compute the effective address for the store. The contents of register RA are stored into this effective

address. The contents of register RD can be 1 or 2. The store is cancelled if RD is 0. No exceptions are generated. Additionally, the contents of register RA are incremented by 1.

66. SWSGD

SWSGI $RA, $RB, $RD

This special instruction allows writes into the video RAM. Concatenating the two lower order bits of the contents of register RD and the contents of register RB compute the effective address for the store. The contents of register RA are stored into this effective address. The contents of register RD can be 1 or 2. The store is cancelled if RD is 0. No exceptions are generated. Additionally, the contents of register RA are decremented by 1.

67. SWAP

SWAP $RA, $RD

The lower byte and the higher byte of register RA and swapped and placed in RD.

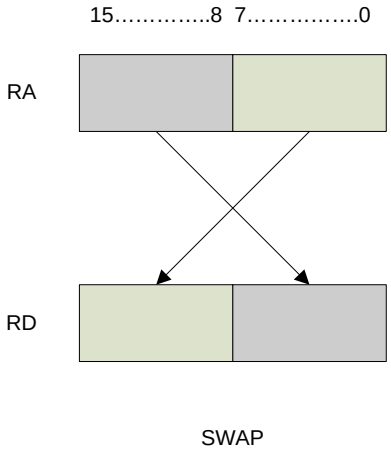

Figure 15: SWAP

Instruction Encoding

Table 6 shows the bit encoding for all the instructions supported by WIMP. All instructions have a 5-bit opcode. Instructions that do not have an 8-bit immediate field have a 2-bit sub-op. The instructions shaded in yellow are system instructions that help in interrupt and exception handling.

Table 6: WIMP instruction encoding

ADD.#	0	0	0	0	0	0	0/1	RA	RA	RA	RB	RB	RB	RD	RD	RD
SUB.#	0	0	0	0	0	1	0/1	RA	RA	RA	RB	RB	RB	RD	RD	RD
AVG.#	0	0	0	0	1	0	0/1	RA	RA	RA	RB	RB	RB	RD	RD	RD
SAVG.#	0	0	0	0	1	1	0/1	RA	RA	RA	RB	RB	RB	RD	RD	RD
RSL.#	0	0	0	1	0	X	0/1	RA	RA	RA	RB	RB	RB	RD	RD	RD
RSA.#	0	0	0	1	1	X	0/1	RA	RA	RA	RB	RB	RB	RD	RD	RD
LSL.#	0	0	1	0	0	X	0/1	RA	RA	RA	RB	RB	RB	RD	RD	RD
ROT.#	0	0	1	0	1	X	0/1	RA	RA	RA	RB	RB	RB	RD	RD	RD
CMP.EQ.#	0	0	1	1	0	0	0/1	RA	RA	RA	RB	RB	RB	CB	CB	CB
CMP.NE.#	0	0	1	1	0	1	0/1	RA	RA	RA	RB	RB	RB	CB	CB	CB
CMP.GT.#	0	0	1	1	1	0	0/1	RA	RA	RA	RB	RB	RB	CB	CB	CB
CMP.GE.#	0	0	1	1	1	1	0/1	RA	RA	RA	RB	RB	RB	CB	CB	CB
CMP.LT.#	0	1	0	0	0	0	0/1	RA	RA	RA	RB	RB	RB	CB	CB	CB
CMP.LE.#	0	1	0	0	0	1	0/1	RA	RA	RA	RB	RB	RB	CB	CB	CB
MAC.LO	0	1	0	0	1	X	0	RA	RA	RA	RB	RB	RB	RD	RD	RD
MAC.HI	0	1	0	0	1	X	1	RA	RA	RA	RB	RB	RB	RD	RD	RD
MUL.LO	0	1	0	1	0	X	0	RA	RA	RA	RB	RB	RB	RD	RD	RD
MUL.HI	0	1	0	1	0	X	1	RA	RA	RA	RB	RB	RB	RD	RD	RD
MIX.LO	0	1	0	1	1	X	0	RA	RA	RA	RB	RB	RB	RD	RD	RD
MIX.HI	0	1	0	1	1	X	1	RA	RA	RA	RB	RB	RB	RD	RD	RD
MUX.LO	0	1	1	0	0	X	0	RA	RA	RA	RB	RB	RB	RD	RD	RD
MUX.HI	0	1	1	0	0	X	1	RA	RA	RA	RB	RB	RB	RD	RD	RD
AND	0	1	1	0	1	0	0	RA	RA	RA	RB	RB	RB	RD	RD	RD
OR	0	1	1	0	1	0	1	RA	RA	RA	RB	RB	RB	RD	RD	RD
XOR	0	1	1	0	1	1	0	RA	RA	RA	RB	RB	RB	RD	RD	RD
NOT	0	1	1	0	1	1	1	RA	RA	RA	X	X	X	RD	RD	RD
COPY.LO	0	1	1	1	0	0	0	RA	RA	RA	X	X	X	RD	RD	RD
COPY.HI	0	1	1	1	0	0	1	RA	RA	RA	X	X	X	RD	RD	RD
SWAP	0	1	1	1	0	1	X	RA	RA	RA	X	X	X	RD	RD	RD
LW	0	1	1	1	1	0	0	RA	RA	RA	X	X	X	RD	RD	RD
LWI	0	1	1	1	1	0	1	RA	RA	RA	X	X	X	RD	RD	RD
LWD	0	1	1	1	1	1	0	RA	RA	RA	X	X	X	RD	RD	RD
SW	1	0	0	0	0	0	0	RA	RA	RA	RB	RB	RB	X	X	X

SWI	1	0	0	0	0	0	1	RA	RA	RA	RB	RB	RB	X	X	X
SWD	1	0	0	0	0	1	0	RA	RA	RA	RB	RB	RB	X	X	X
READ	1	0	0	0	1	X	0	IM	IM	IM	CB	CB	CB	RD	RD	RD
WRITE	1	0	0	0	1	X	1	IM	IM	IM	CB	CB	CB	RD	RD	RD
WAIT	1	0	0	1	0	0	0	IM	IM	IM	IM	S	S	X	X	X
KILL	1	0	0	1	0	0	1	X	X	X	X	X	X	X	X	X
INIT	1	0	0	1	0	1	0	RA	RA	RA	X	IM	IM	X	X	X
JMPR	1	0	0	1	1	X	0	RA	RA	RA	X	X	X	CB	CB	CB
JMP.SPR1	1	0	0	1	1	X	1	X	X	X	X	X	X	X	X	X
JALR	1	0	1	0	0	X	X	RA	RA	RA	X	X	X	CB	CB	CB
JMPI	1	0	1	0	1	IM	IM	IM	IM	IM	IM	IM	IM	CB	CB	CB
JALI	1	0	1	1	0	IM	IM	IM	IM	IM	IM	IM	IM	CB	CB	CB
LI.HI	1	0	1	1	1	IM	IM	IM	IM	IM	IM	IM	IM	RD	RD	RD
LI.LO	1	1	0	0	0	IM	IM	IM	IM	IM	IM	IM	IM	RD	RD	RD
GET.FLAG	1	1	0	0	1	0	0	X	X	X	X	X	X	RD	RD	RD
GET.SPR1	1	1	0	0	1	0	1	X	X	X	X	X	X	RD	RD	RD
GET.SPR2	1	1	0	0	1	1	0	X	X	X	X	X	X	RD	RD	RD
GET.EXR	1	1	0	0	1	1	1	X	X	X	X	X	X	RD	RD	RD
GET.WAIT_REG	1	1	0	1	0	0	0	X	X	X	X	X	X	RD	RD	RD
GET.TAR	1	1	0	1	0	0	1	X	X	X	X	X	X	RD	RD	RD
GET.WS	1	1	0	1	0	1	0	X	X	X	X	X	X	RD	RD	RD
GET.INTR	1	1	0	1	0	1	1	X	X	X	X	X	X	RD	RD	RD
PUT.FLAG	1	1	0	1	1	0	0	RA	RA	RA	X	X	X	X	X	X
PUT.SPR1	1	1	0	1	1	0	1	RA	RA	RA	X	X	X	X	X	X
PUT.DEBUG	1	1	0	1	1	1	0	X	X	X	X	X	IM	X	X	X
PUT.INTRMASK	1	1	0	1	1	1	1	X	X	X	X	X	IM	X	X	X
ADDI.#	1	1	1	0	0	0	0/1	RA	RA	RA	IM	IM	IM	RD	RD	RD
SUBI.#	1	1	1	0	0	1	0/1	RA	RA	RA	IM	IM	IM	RD	RD	RD
GET.THREAD	1	1	1	0	1	0	0	X	X	X	X	X	X	RD	RD	RD
SWFT	1	1	1	1	0	0	0	RA	RA	RA	RB	RB	RB	RD	RD	RD
SWSG	1	1	1	1	0	0	1	RA	RA	RA	RB	RB	RB	RD	RD	RD
SWSGI	1	1	1	1	0	1	0	RA	RA	RA	RB	RB	RB	RD	RD	RD
SWSGD	1	1	1	1	0	1	1	RA	RA	RA	RB	RB	RB	RD	RD	RD
NOP	1	1	1	1	1	X	X	X	X	X	X	X	X	X	X	X

Interrupts

Figure 16 describes the interrupt interface between the processor and the device that interrupts. An interrupt cannot be processed if the thread is inactive or if the thread is waiting to be synchronized. Once the processor latches the address and acknowledges the interrupt, the following events take place:

1. Complete the current instruction.
2. Saves the processor state (PC+2, condition codes and R0 thru R7)
3. Service the interrupt
4. Restore processor state
5. Continue with the execution of the current program

If an interrupt is being serviced, another interrupt will not be acknowledged. Although an exception can occur within interrupt, an interrupt will not be serviced when an exception is being handled. An interrupt is not acknowledged when the current instruction is a jump.

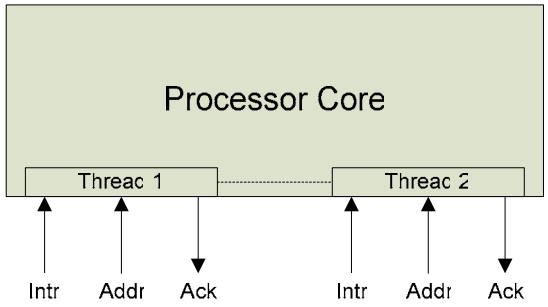

Figure 16: Interrupt interface

The timing relation between the processor and the device interrupting the thread is show in Figure 17.

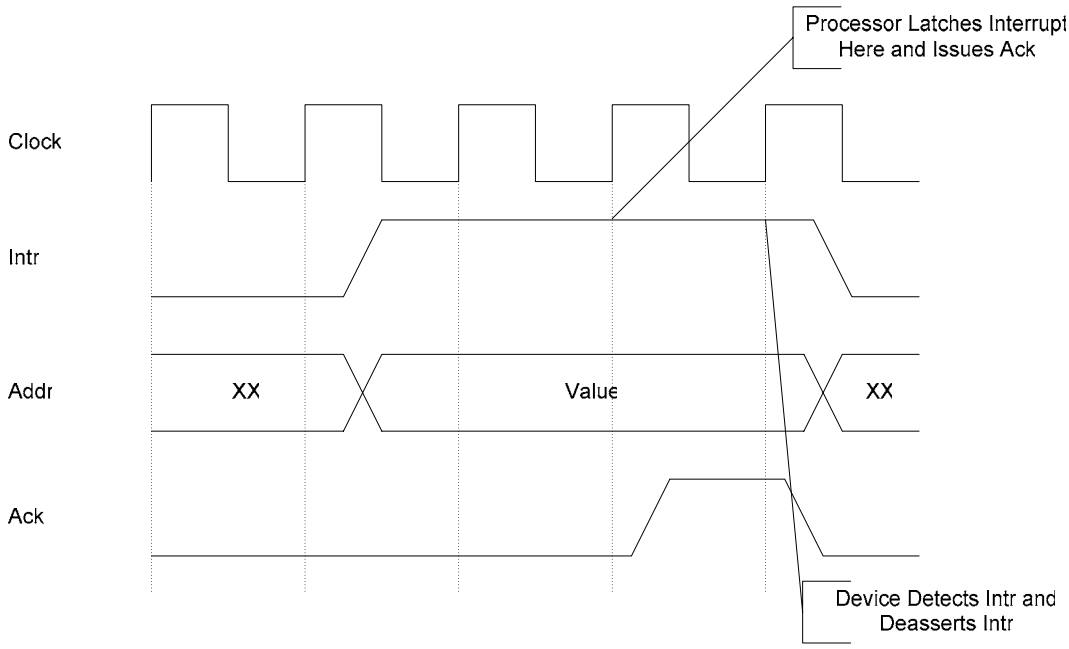

Figure 17: Interrupt timing

Exceptions

Exceptions are internal to the processor. Exceptions can be caused due to:

- Illegal instruction
- Unaligned address for load
- Starting hardware thread that is already busy
- Killing a thread that is waiting to be synchronized
- Store to a location from 0000 – 7FFF (Program Memory)

The exception raised in one thread usually affects only that thread, all other threads will continue to run without any issues. The instruction is not committed into the register file. The thread state (PC, condition codes, instruction and R0 through R7) is saved in a predefined location. An exception routine is serviced before the thread is killed. The thread can be restarted only by system reset or through some other thread. Exceptions mask interrupts, implying that an interrupt will not be serviced if an exception is being handled.

ILLEGAL_INSTRUCTION
JMP.SPR1 when not servicing an interrupt
Unrecognized opcode

UNALIGNED_ADDRESS
Loads and stores from an even address
Jump to an even address

ILLEGAL_ADDRESS
Jump to data memory
Store to program memory

ILLEGAL_INIT
Start a new thread that is already active

ILLEGAL_WAIT
Waiting for a thread which is already dead

The following table lists the exception codes and their names…

Table 7: EXR definition

BIT NUMBERS IN EXR	EXCEPTION
0	ILLEGAL_INSTRUCTION
1	UNALIGNED_ADDRESS
2	ILLEGAL_ADDRESS
3	INIT_THREAD
4	ILLEGAL_WAIT
5-15	Reserved

Writing to Output Ports

The write instruction has the following semantic…

WRITE IMM3, $CB, $RD

The contents of R2 are written into the port number that is mapped by IMM3. If the write is successful, $CB in the flag register is set. If the write to the port is unsuccessful, a value 0 is written into the location in the flag register pointed by $CB. Table 8 indicates the mapping of the output port number to the IMM3 value.

Table 8: Port mapping

IMM3	PORT #
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Figure 18 shows the output control interface of WIMP and the devices. Each port in the output controller is associated with an output queue. Each queue is 256 deep and 16-bits wide. If the queue associated with the port to be written into is not full, the output control writes the data into the queue and returns a 1 in the same clock cycle. On the other hand, if the queue is full, the output control prevents the writing into the queue and returns the status as 0. The timing relationship is shown in Figure 19. A similar protocol is followed on the device side where a read command is expected from the output device when the queue is not empty.

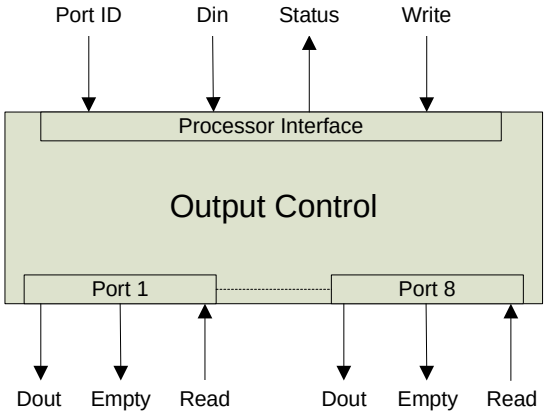

Figure 18: Output control interface

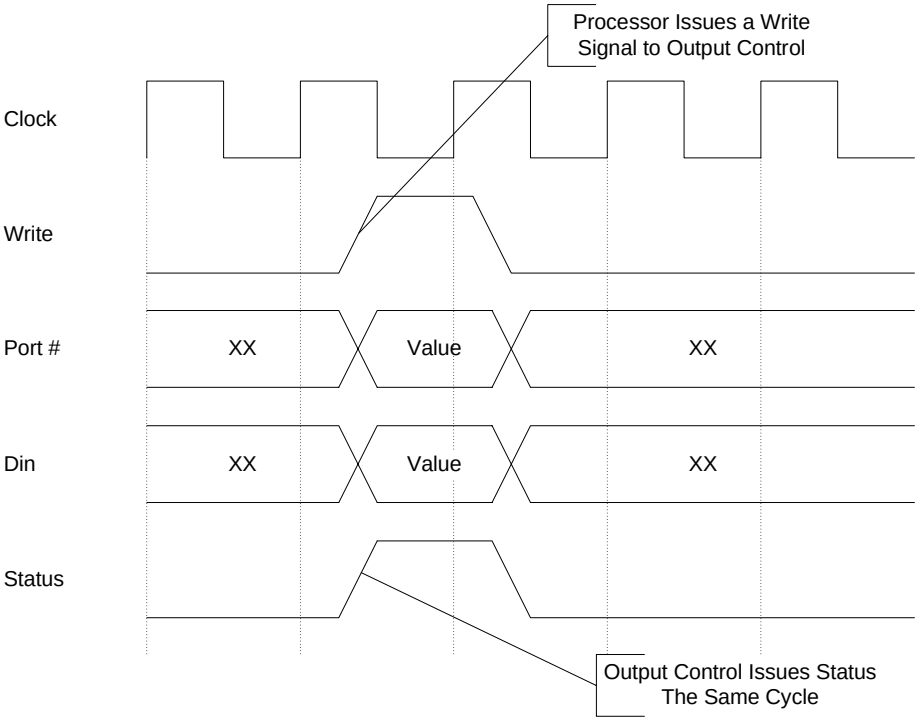

Figure 19: Timing relation between WIMP core and output control

Reading from Input Ports

The read instruction has the following semantics…
READ IMM3, $CB, $RD

The contents from port specified by IMM3 are copied into RD. If the read operation from the port was successful, a 1 is written into $CB; otherwise, a 0 is written into $CB. The processor can read from 8 different ports specified by the IMM3 value. Table 9 shows the mapping of the input ports to the IMM3 value.

Table 9: Port mapping

IMM3	PORT #
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

The read instruction communicates with the input control in execute 1 stage of the pipeline. The input control can interface 8 input ports to the processor core. Each port is associated with an input queue. An input device connected to an input port writes into the FIFO as long it is not full. On the other hand, the processor can read from an input port only when the queue is not empty. The status signal returns a 0 when the processor attempts to read from an empty input queue and a 1 when the queue is not empty. Figure 20 shows the Input Control interface on the processor core side and the input device side.

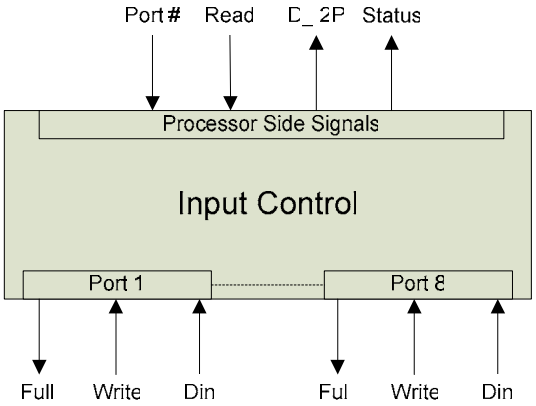

Figure 20: Input control interface

Initializing a New Thread

- During system reset
  At reset, only hardware thread 1 is active. This is the default thread and is hardwired into the system.
- Normal working
  During normal working, a thread can initiate a new hardware thread with the INIT instruction.

The semantics of the INIT instructions are as follows…

INIT $RA, IMM2

Example,
Thread 2 has the instruction,

INIT $5, 0x03

The processor checks if thread 3 is already active. If so, an INIT THREAD EXCEPTION occurs in thread 2 while thread 3 continues as is. If thread 3 is not active, the PC for thread 3 is loaded with the contents of R5. Thread 3 will start in the next machine cycle. Once thread 3 is active, it can be killed only by the KILL instruction present in its instruction stream. Thread 2 will have no more control over thread 3.

If all the threads are killed, the processor halts. The processor can be activated only with the system reset. There should be at least one active thread at all times.

The INIT instruction takes any where from 4 to 7 clock cycles to come into effect. In other words, the new thread is brought into effect in the next machine cycle. The following example makes the timing of the INIT instruction clear.

Consider the following instruction in the memory

Table 10: Initializing Threads

T1		T2		T3		T4	
ADDR	**INSTR**	**ADDR**	**INSTR**	**ADDR**	**INSTR**	**ADDR**	**INSTR**
Z	INIT T2, Y	x	x	x	x	x	x
Z+2	ADD	Y	LOAD	x	x	x	x

x : don’t care

Table 10 shows thread 1 initialing a new thread T2 to start from location Y. The instruction INIT instruction is present in the location Z. The processor guarantees that the instruction in thread T2 (which is LOAD) WILL be executed after the instruction in Z+2 (which is an ADD), implying that the new thread will always start in the next machine cycle.

Thread Synchronization

The processor supports a very simple form of a thread synchronization using the WAIT instruction. The semantics of the WAIT instruction are as follows:

WAIT IMM[3], IMM[2], IMM[1], IMM[0], S_{10}

IMM[0] ➔ Thread 0
IMM[1] ➔ Thread 1
IMM[2] ➔ Thread 2
IMM[3] ➔ Thread 3

Say the instruction in thread 1 is
WAIT 1, 0, 1, 0, 2

When the processor comes across this instruction, it will stall thread 1 till it comes across a similar instruction in thread 3 that points to the synchronization point 2.

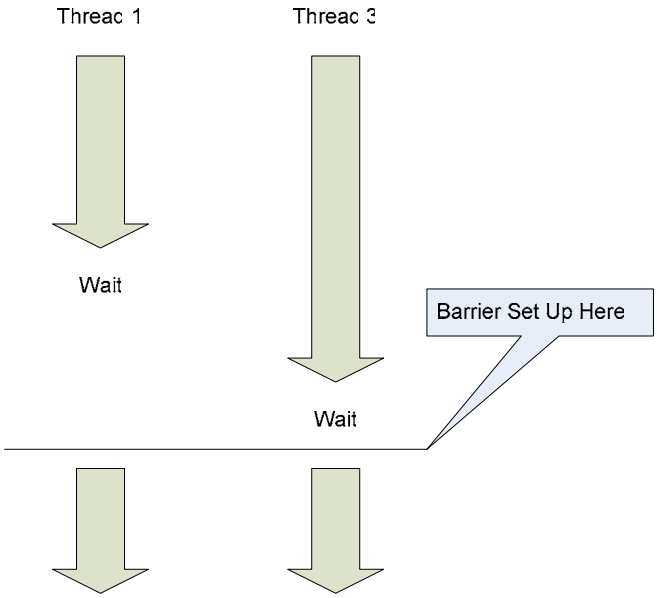

Figure 21: Thread synchronization using barrier

It wouldn't matter if thread 3 had reached the barrier point before. Thread 3 will stall until thread 1 reaches the same instruction. Both the WAIT instructions can be in different places in the memory. An ILLEGAL_WAIT exception will occur in thread-n if thread-n is waiting for thread-m and thread-m is killed.

Rules to be followed for wait instruction

- The thread that reaches the barrier first should start first
- A thread cannot wait for a dead thread

Video RAM and the VGA

The video RAM has a byte for every pixel on the VGA. The byte corresponds to the grey scale value of the pixel that needs to be displayed by the VGA. Figure 22 shows the position on the VGA corresponding to a location in the video RAM.

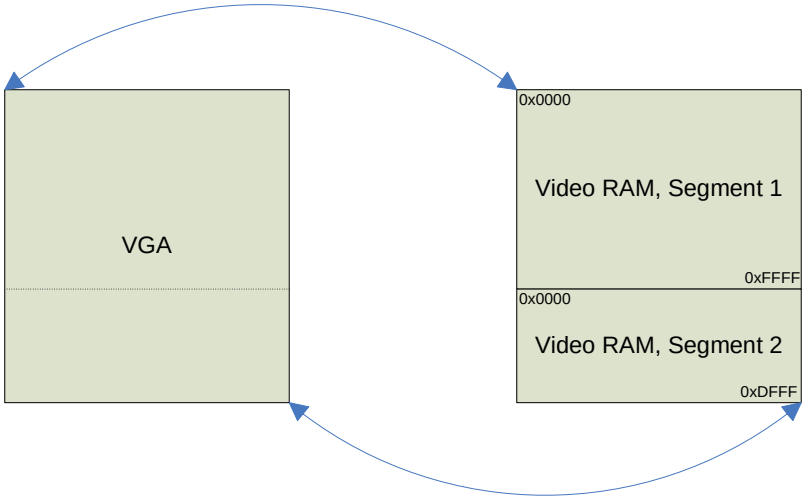

Figure 22: Video RAM and the VGA

Writing into the Video RAM is decoupled from refreshing aspect of the VGA. Applications can write into the Video RAM using the SWSG, SWSGI and SWSGD instructions.

References

Ruby B. Lee and A. Murat Fiskiran, “PLX: A Fully Subword-Parallel Instruction Set Architecture for Fast Scalable Multimedia Processing”, *Proceedings of the 2002 IEEE International Conference on Multimedia and Expo (ICME 2002)*, pp. 117-120, August 2002

John Glossner, Michael Schulte, Mayan Moudgill, Suman Mamidi, “A Low-Power Multithreaded Processor for Baseband Communication Systems”, *Springer-Verlag (2004)*, in print

Theo Ungerer, Borut Robic, Jurij Silc, "A survey of processors with explicit multi-threading", *ACM computer survey*, 2003, vol. 35, pp. 29-63