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

Micro-Architecture Manual

Bryan Berns
Jacob Petranak
Jordan Wenner
Parikshit Narkhede
Suman Mamidi

Table of Contents

WIMP Micro-architecture

Pipeline Description

Figure 1 shows the block diagram of WIMP. The two physical banks on the prototype board are logically divided into instruction and data memory. The instruction memory ranges from 0000 to 7FFF. The data memory starts at 8000 and ends at FFFF.

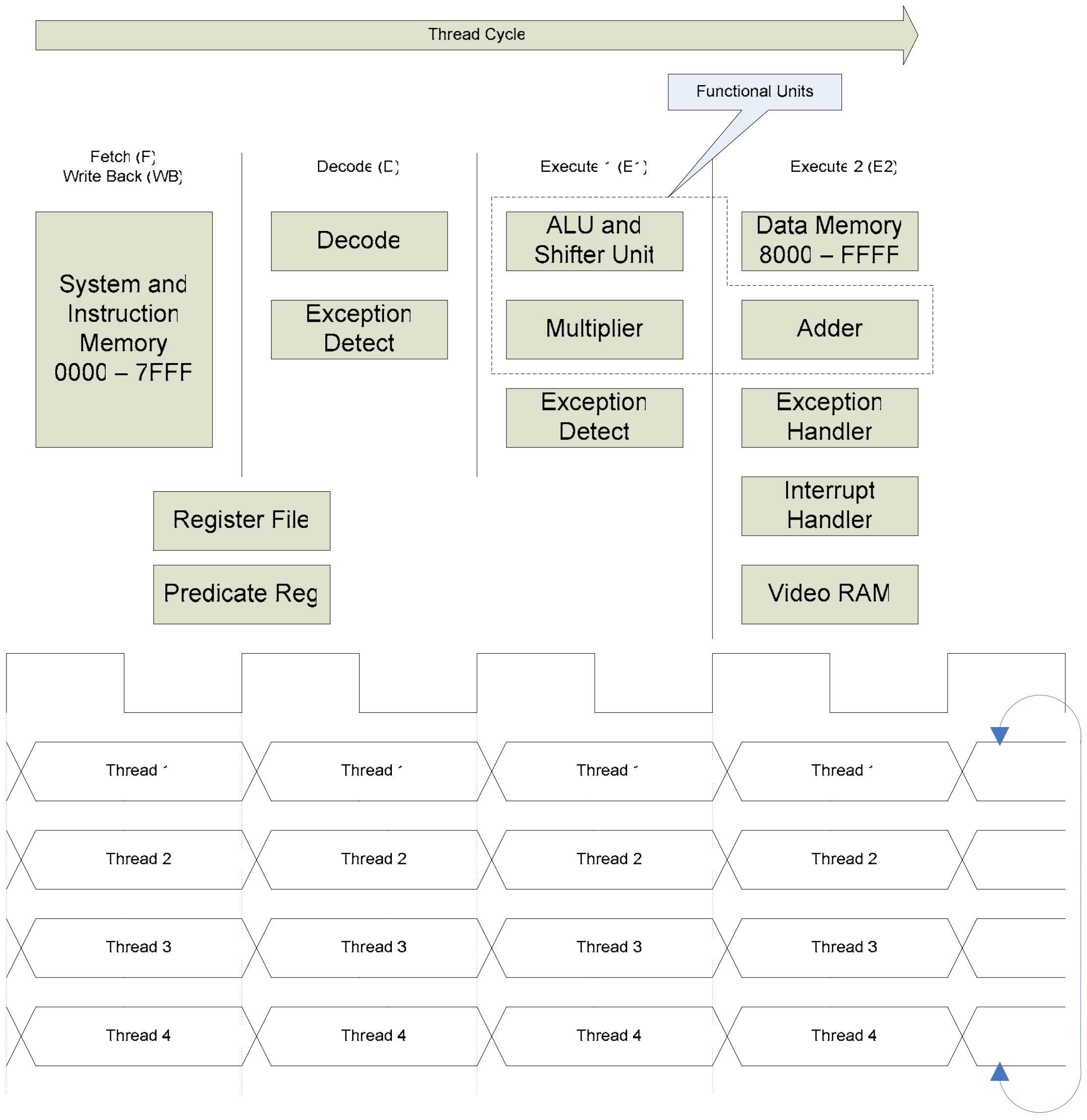


Figure 1: WIMP block diagram

Instructions are fetched in the Fetch stage. For any thread, a new instruction is being fetched, while the old instruction is being retired, implying that the fetch and the writeback stage are merged. The control signals for all the units are generated in the decode stage. In case, a thread is not active, the fetched instruction is converted to a NOP. In addition, the PC is also prevented from incrementing when the thread is not active. The register file and the flag register are read in the decode stage. The decode stage detects the illegal instruction, unaligned address for loads, stores and jump registers, and illegal init exceptions. The decode stage also flags if the thread is ready to acknowledge an interrupt. Two execute stages, namely E1 and E2 are dedicated to functional units. WIMP has a subword parallel ISA and the functional units supporting this ISA are shown in a dotted enclosure. E1 holds the ALU, shifter, permute unit, and the multiplier. In addition, unaligned and illegal addresses for jump immediate are detected in E1. The adder for the MAC instruction is present in E2. The data is fetched from memory in E2. Exceptions are detected in every stage but are handled only in E2. Interrupts are also sampled in E2. At the end of E2, the PC is made ready to fetch the next instruction and the current instruction is made ready for retirement.

Register File

Architecturally, every thread has it own register file. Each register file has two write ports and three read ports as shown in Figure 2. The three read ports support instructions like MAC that require three operands. The two write ports allow instructions like LWI (load increment) and SWI (store word increment) to be committed without stalls.

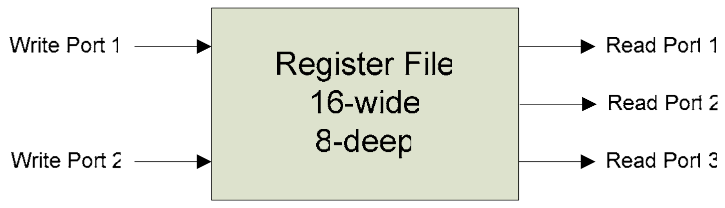


Figure 2: Register file port map

The register files are implemented in the block select RAM’s provided in the XILINX Virtex architecture. The block select RAM’s are configured to be 16-bits wide and 256 deep. Three block-select RAM’s are used to allow three read ports in the register file. Though the four register files are architecturally mutually exclusive, they reside in the same set of block-select RAM’s. Table 1 shows the address mapping of the block select RAM’s to the thread id.

Table 1: Using block select RAM for register files

Address Range	**Thread number**
0 – 7	Thread 1
8 – 15	Thread 2
16 – 23	Thread 3
24 – 31	Thread 4

The other locations in the block select RAM are unused. Writes are processed at twice the processor clock rate to simulate the two write ports.

Flag Register

WIMP uses a limited form of predication applied to jump instructions only. There is a flag register available for every thread. The bits in the flag register are modified by the compare and arithmetic instructions. The jump instructions read the bits in the flag register to resolve the jump. Flag(0) is always 1 to allow unconditional jumps. Figure 3 shows the interface for a flag register that has two write ports and a single read port.

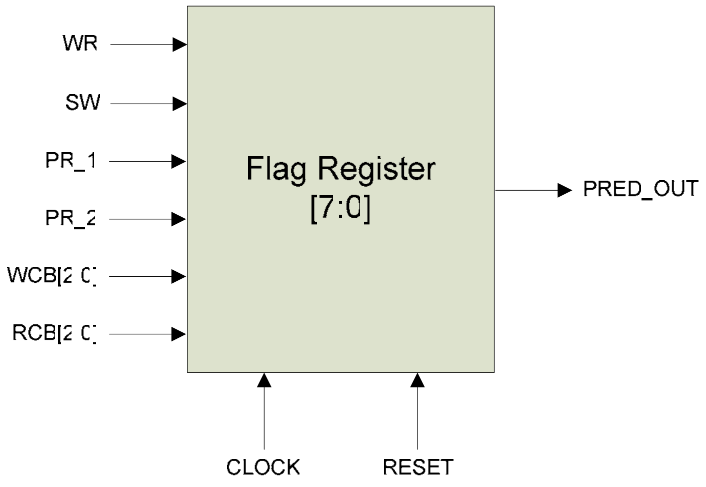


Figure 3: Flag register interface

The flag register clocks in PR_1 only if SW indicates a 16-bit operation. The flag register clocks in both PR_1 and PR_2 if the SW indicates an 8-bit operation. The address to be written into is indicated by WCB. The jump instructions read PRED_OUT pointed by RCB[2:0]. The processor jumps only if this value is '1'.

WIMP and Interrupts

Figure 4: Interrupt interface describes the interrupt interface between the processor and the device that interrupts. An interrupt cannot be processed if the thread is inactive. 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 an interrupt, it will not be serviced when an exception is being handled. An interrupt is not acknowledged when the current instruction is a jump.

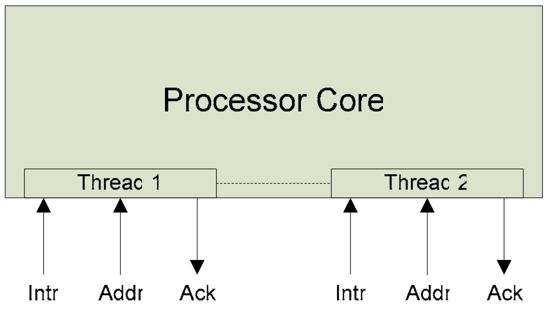


Figure 4: Interrupt interface

The timing relation between the processor and the device interrupting the thread is show in Figure 5.

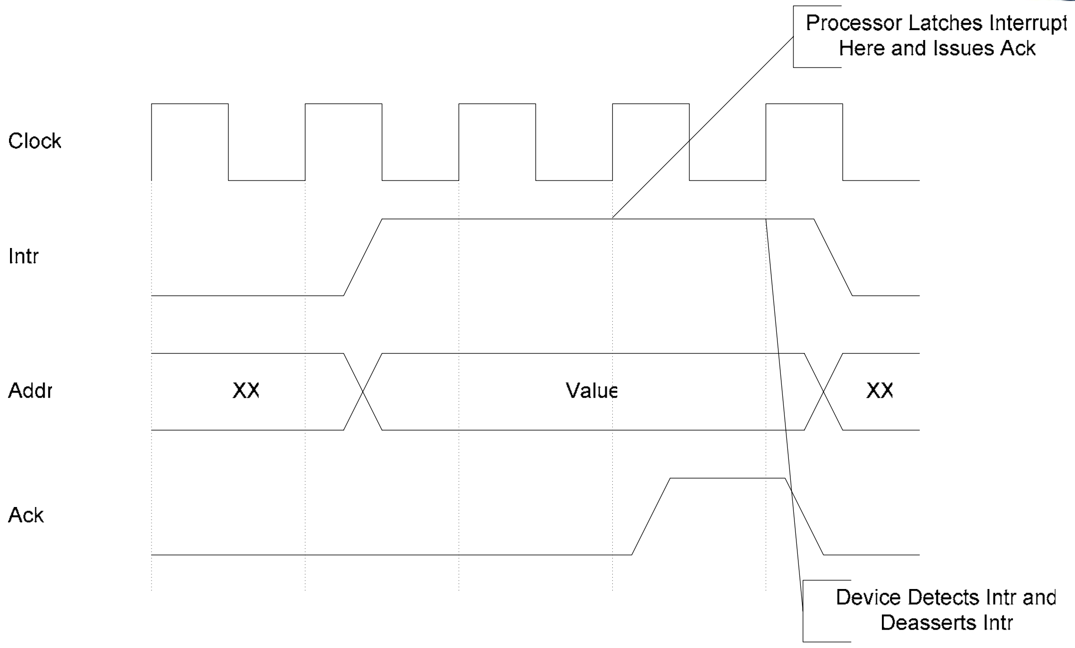

Processor Latches Interrupt Here and Issues Ack
Clock
Intr
Addr
XX
Value
XX
Ack
Device Detects Intr and Deasserts Intr

Figure 5: Interrupt timing

Implementing Interrupts in WIMP

Interrupts are handled by software rather than hardware. The ISA and the hardware provide support for handling interrupts, but do not handle them by its own.

The processor incorporates the following instructions that that the ISA provides for interrupt handling.

- 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.

- GET.SPR1 $RD
  Copies the SPR1 register (holds address to jump to after interrupts) into register RD.

- GET.SPR2 $RD
  Copies the SPR2 register (holds address to jump to after exceptions) to register RD.

- GET.EXR $RD
  Copies the EXR register (holds type of exception that occurred) to register RD. In addition the EXR register is cleared.

- 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.

- 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.

- 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.

- GET.INTR $RD
  Copies the INTR_ADDR register (holds address to jump to when interrupt occurs) to register RD.

- PUT.FLAG $RA
Loads the flag register with the lower byte of register RA.

- PUT.SPR1 $RA
Loads the SPR1 register with the contents of register RA.

- PUT.DEBUG IMM1
Writes the value specified in the IMM field into the 1-bit DEBUG register.

- 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.

This instruction is available to the user. No opcode is assigned to it though.
- INTRCOMPLETE
This is a pseudo instruction that translates to something like…
```
LI.HI $7, 0x06
LI.LO $7, 0x00
JMPR $7, $0
```

Consider the sequence of instructions…
```
0001          ADD
0002          SUB
0003          MUL
```

Say the interrupt with the destination address is detected between the SUB and the MUL instruction. The SUB instruction is completed and the PC is loaded with a predefined value 0500 (See memory map in the architecture manual). The Intr Mask (Interrupt Mask) is set to avoid an interrupt within an interrupt. The software routine in 0500 saves the thread state before jumping to the destination address. The interrupt handler indicates the completion of its routine by the INTRCOMPLETE instruction. The INTRCOMPLETE instruction loads the PC with the routine that restores the processor state. After the processor state is restored, the Intr Mask is reset, indicating that the processor is ready to accept another interrupt. In the final step, the processor loads the MUL instruction and continues normal execution.

#define Intr Mask = Interrupt Mask, when 1, interrupts not acknowledged
#define Intr = The interrupt pin at the external interface
#define ws[3:0] = A four bit register that indicates if the thread is in wait state
#define tar[3:0] = A four bit register that indicates if a thread is active
#define current_thread = A two bit value that indicates the current thread being processes by the stage
#define Intr_Addr[15:0] = 16 bit register, holds address to jump to when INT occurs

```
D:
    If ((instruction != jump) & (tar[current_thread] != 0) & (ws[current_thread] != 1)
& (!Intr_Mask))
        Intr_accept_ready = 1
    Else
        Intr_accept_ready = 0
E2:
    If ((Intr_accept_ready == 1) & (~exception_detect) & (INTR = active))
        Set Intr Mask
        Pc = OS_intr_address
        Set Ack
        Clock in ISR address into Intr Addr
    Else
        Reset Ack
    Endif
```

WIMP and 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. The thread is killed after the processor's state is put onto the output port. The thread can be restarted only by the 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

Implementing Exceptions in WIMP

Exceptions are handled by software rather than hardware. The ISA provides support for handling exceptions. An exception causes the processor to enter a debug mode where the register R0 can now be written into and is not read only. The destination for the processor store state is loaded into R0.

The following table lists the exception codes and their names…

Table 2: 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

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

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

In the above sequence of instructions, thread 1 initialized a new thread T2 to start from location Y. The INIT instruction is present in 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.

The following algorithm should ensure that the new thread that starts will begin its execution in the next machine cycle.

```
#define Tar[3:0] = A register that indicates if the thread is active
#define Imm2 = the thread to be started provided in the instruction
#define new_address = location from where the new thread starts off
#define current_thread = the thread being processed by E2
#define nt_addr[15:0] = a register that holds the possible PC
#define nt_reg[3:0] = indicates which thread needs to be activated
E2:
        If (Tar[current_thread] == active)
                If (Tar[Imm2] != 0)
                        Raise_exception
                Else
                        Set nt_reg[Imm2]
                        Write   nt_addr[Imm2] = new_address
                Endif
        Else
                If (nt_reg[current_thread] == active)
                        Set tar[current_thread]
                        Send new_address to memory from nt_addr
                        Reset EXR
                        Reset Intr_Mask
                        Reset nt_reg[current_thread]
                Endif
        Endif
```

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], SP

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 till it comes across a similar instruction in thread 3 that points to the synchronization point 0.

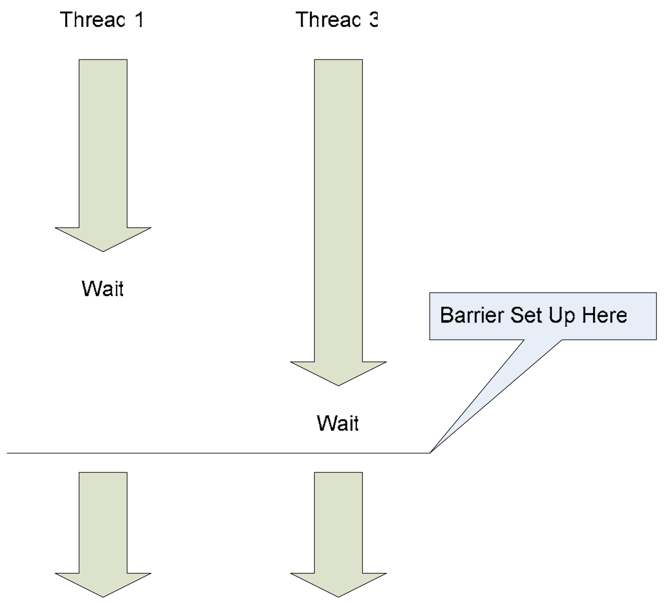


Figure 6: Illustrating thread synchronization

It wouldn’t matter if thread 3 had reached the barrier point before. Thread 3 will stall till 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

The implementation uses a wait register pointed to by the synchronization point provided in the instruction. The following changes need to be made to the pipeline…

```
#define ws[3:0] = indicates which threads are in wait state
#define Tar[3:0] = a register that indicates if the thread is active
#define Imm4 = indicates which threads need to be synchronized
#define wait_reg[3:0] = 4 bit register that implements wait instruction
#define current_thread = the thread being processed by that stage of the pipeline
```

```
/* All states change the next clock cycle, which implies that if a thread encounters a wait
instruction, it waits on the next instruction */
E2:
    if(thread is not in wait state)
        if(instruction  is a legal wait)
            set_wait_state <= 1'b1;
            if(curr thread is the first to arrive)
                compliment others bits wait reg
            else
                compliment bit corresponding to itself in wait reg
            endif
        endif
    else
        hold_pc
        annul_exceptions
    endif
```

Here is a step by step example of what happens to the wait reg and wait state registers when the wait instructions are encountered in different threads.

T0: wait 0,0,1,1,1   T1: wait 0,0,1,1,1

Figure 7 shows the state of the wait reg and the wait state registers before the wait instructions arrive. Let T0 arrive first instruction to be fetched. After the execution of the wait instruction, the wait reg and the wait state registers look like in Figure 8. Thread 0 enters the wait state where in it waits for thread 1 to arrive at the synchronizing point. Once T1 fetches the same instruction, the wait reg and the wait state registers look like in Figure 9. The threads stay in wait state for one more clock cycle after the execution of the wait instruction in T1 and then both instructions proceed with their new PC's. This is shown in Figure 10.

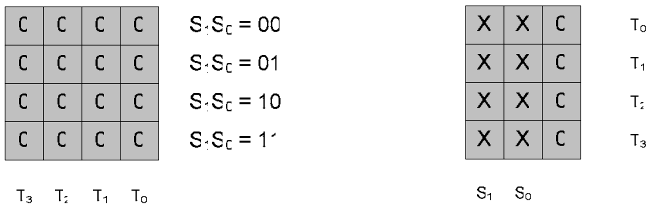


Figure 7: Wait reg and wait state registers before the wait instruction arrive

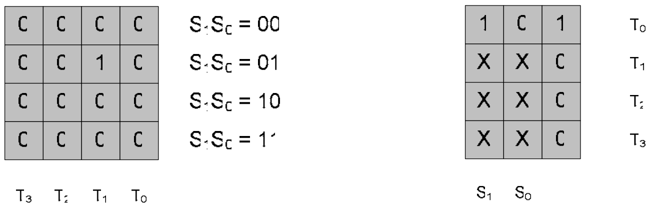


Figure 8: Wait reg and wait state registers after the instruction (wait 0,0,1,1,1) in T0 arrives

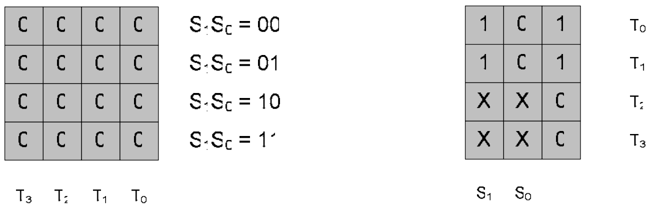


Figure 9: Wait reg and wait state registers after wait instruction (wait 0,0,1,1,1) in T1 arrives

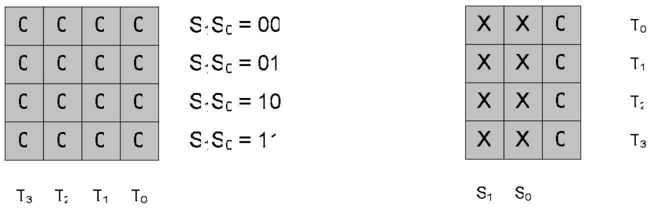


Figure 10: Wait reg and wait state register one clock cycle later. Now both the threads are synchronized

Deciding the Next PC

The program counter (PC) holds the address of the present instruction. The PC is loaded at the end of E2 phase for each thread. If the thread is inactive, the PC can be loaded only from the nt_reg register, which is loaded by another active thread. When the thread is active, the PC can be loaded from any of the following sources in the increasing order of priority.

- PC + 2
- Interrupt handler
- PC+IMM8 or the value stored in the register specifies in the jump instructions
- Exception handler
- PC (If in wait state)

The PC generated by the exception handler is given the highest priority if the thread is not in a wait state. If an interrupt occurs when a jump instruction is being executed, the interrupt is handled after the jump. The program counter is loaded with PC + 2 only if there are no exceptions, jumps or interrupts.

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 3 indicates the mapping of the output port number to the IMM3 value.

Table 3: Output port mapping

IMM3	PORT #
000	1
001	2
010	3
011	4
100	5
101	6
110	7
111	8

Figure 11 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 12. 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.

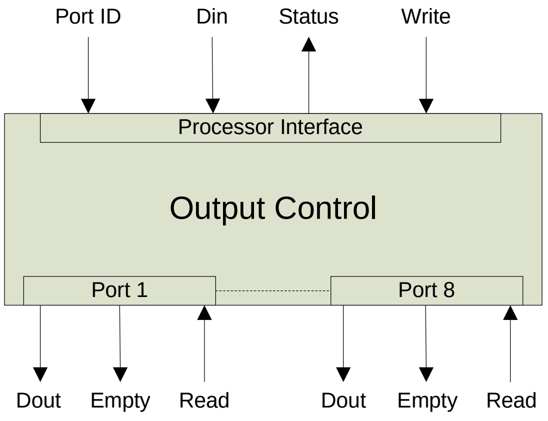


Figure 11: Output control interface

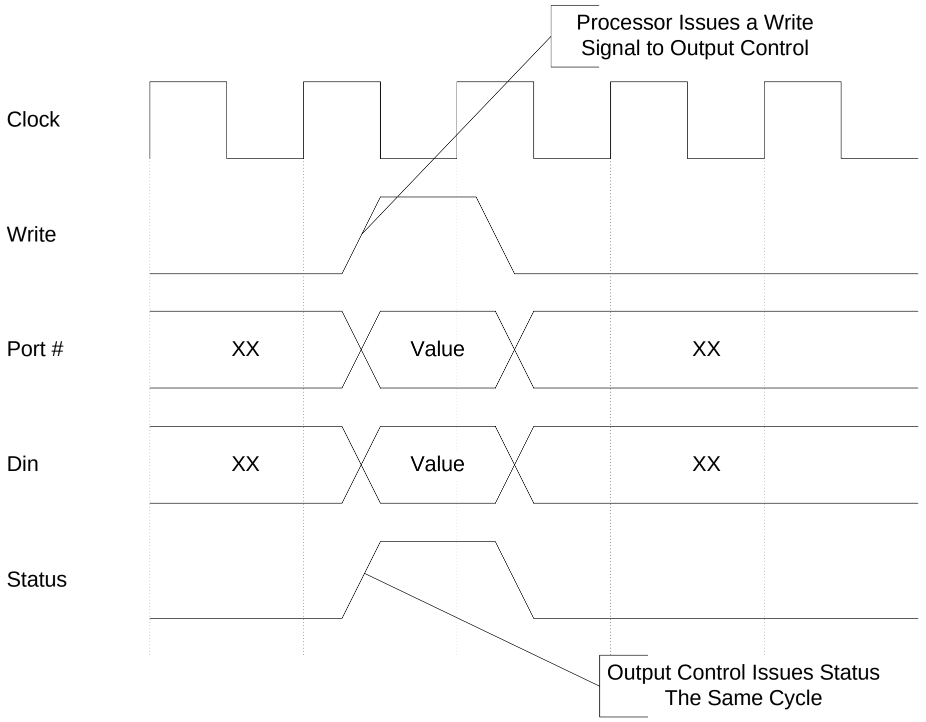

Processor Issues a Write Signal to Output Control
Clock
Write
Port #
XX
Value
XX
Din
XX
Value
XX
Status
Output Control Issues Status The Same Cycle

Figure 12: Timing relation between WIMP core and output control

Figure 13 shows a generic data and control flow used in WIMP that connects the output control to the output device through an interface. The same generic flow is used to connect the keyboard and the RS232 interface (SPART).

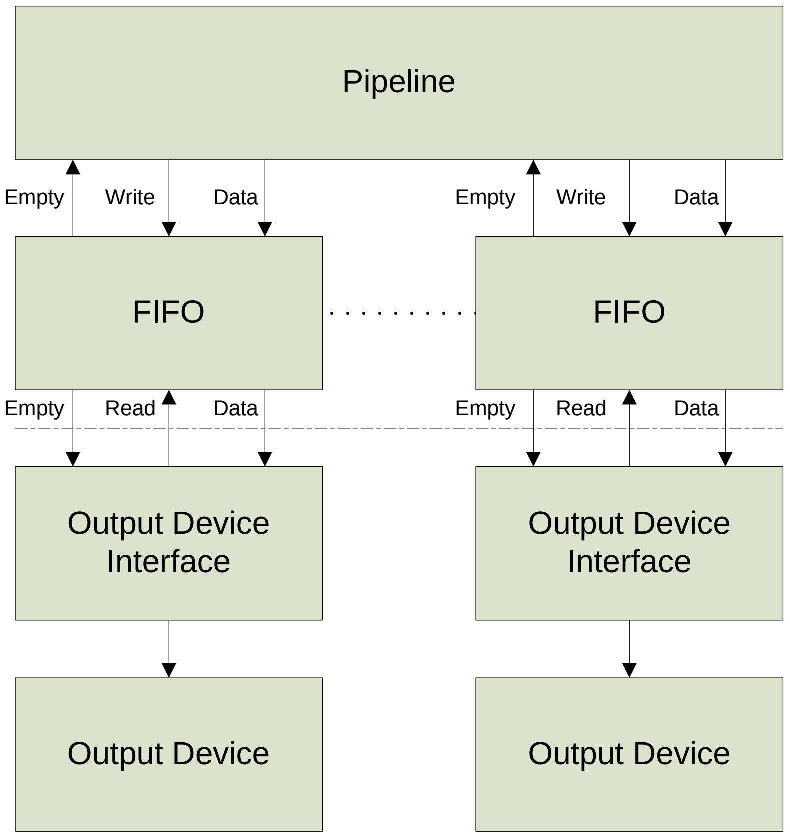


Figure 13: Connecting output devices to WIMP

Interfacing SPAT (Special Purpose Asynchronous Transmitter)

The output control interfaces with the SPAT through port 1 and port 2. Port 1 is assigned to the transmitter and port 2 is assigned to the baud rate generator. Figure 14 shows the SPAT interface to the output control while Figure 15 shows the timing relationship that is required from the interface and the transmitter.

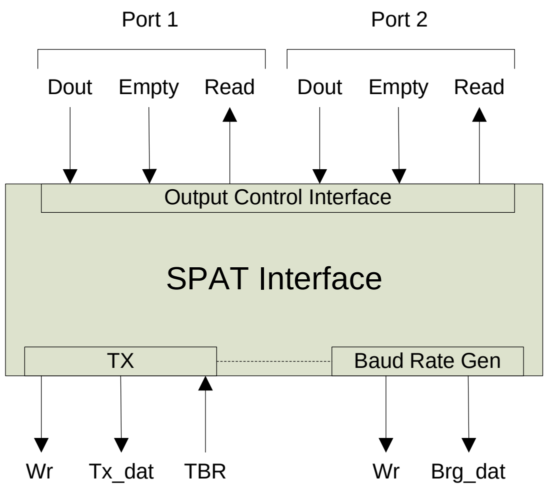


Figure 14: SPAT transmitter interface

The SPAT interface issues a read command to the output interface when the output queue is not empty and the transmitter is not busy (TBR = 1). On the other hand, the SPAT interface will unconditionally write into the divisor buffer of the baud rate generator when the output queue of port 2 has some data.

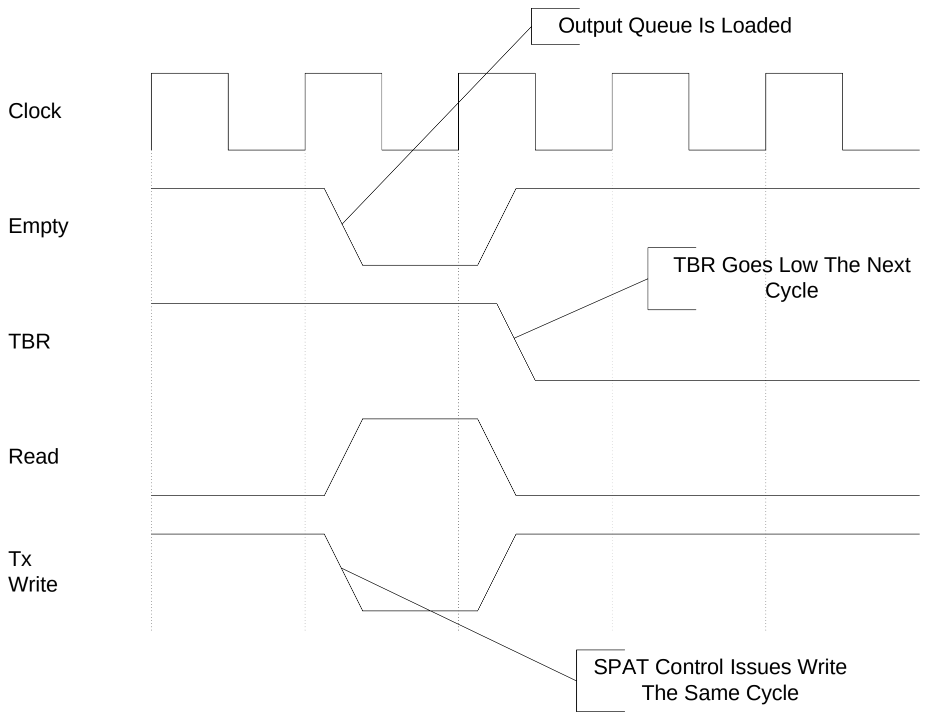

Output Queue Is Loaded
Clock
Empty
TBR Goes Low The Next Cycle
TBR
Read
Tx Write
SPAT Control Issues Write The Same Cycle

Figure 15: Timing relation between SPART transmitter 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 4 shows the mapping of the input ports to the IMM3 value.

Table 4: Input 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 the first execute 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 16 shows the Input Control interface on the processor core side and the input device side.

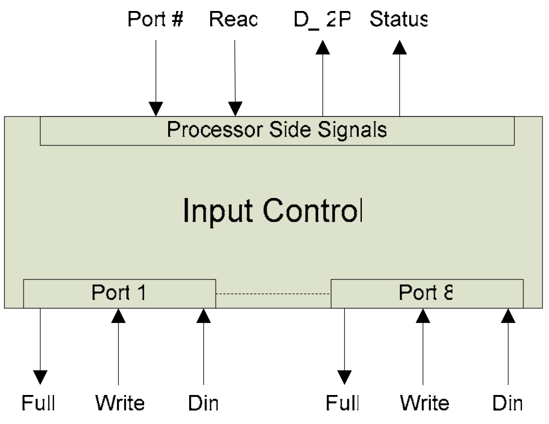


Figure 16: Input control interface

Keyboard Controller

PS2 Interface IP ref: http://opencores.org

The keyboard interface use Interrupt Port 1, Output Port 3 and Input Port 1 of the processor. The output port is used to send the HOST commands to the keyboard and the keystroke is read via the input port. Figure 17 shows the keyboard interface.

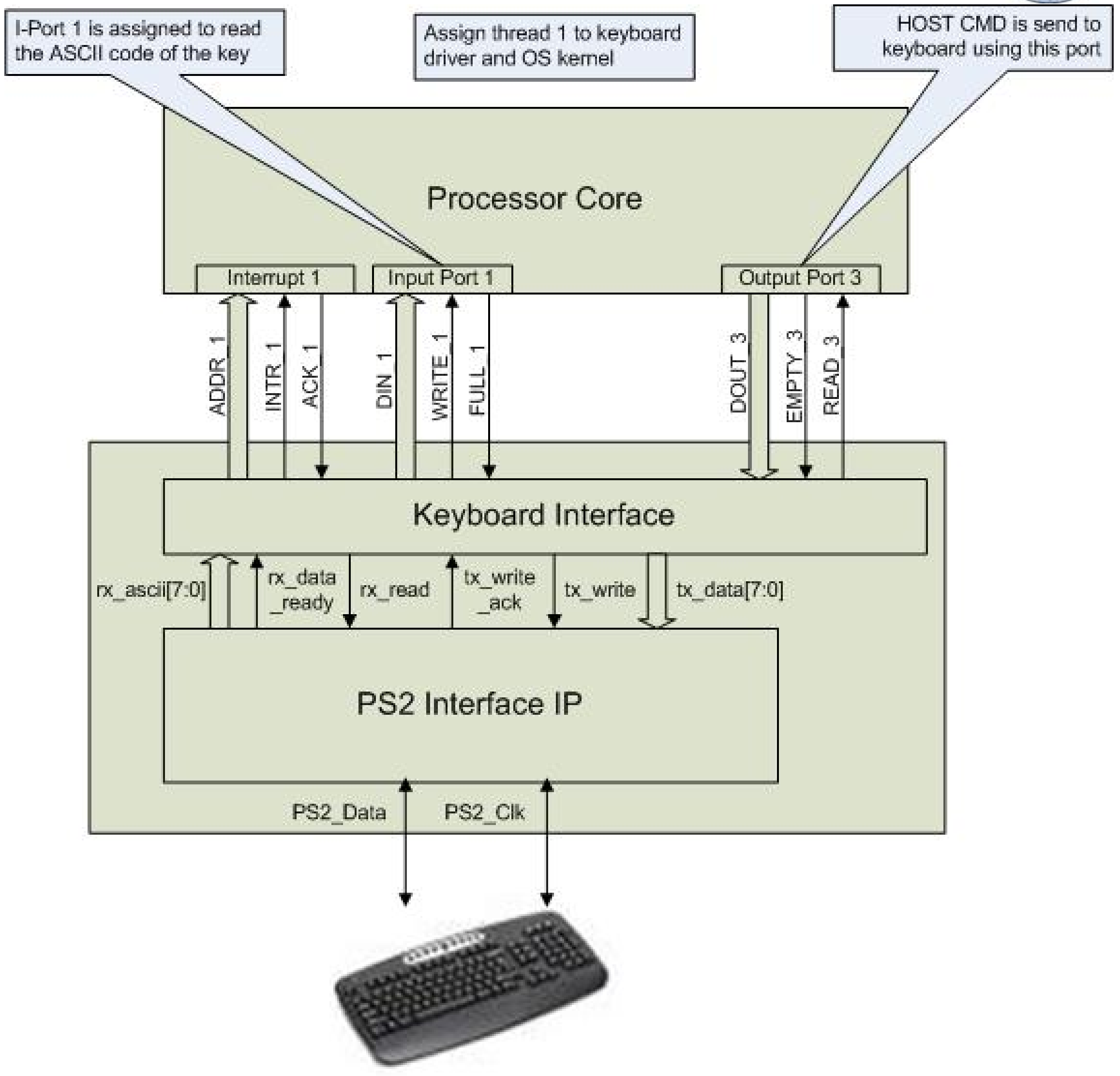


Figure 17: Keyboard interface

The keyboard interface state machine keeps track of the output port and issues a READ command to the processor whenever the output queue is not empty. Then the data read is written into the keyboard interface IP. The keyboard Interface IP is designed such that any write operation to the keyboard (HOST commands) will have precedence over the read commands, and the rest of the keystrokes will be stored into the keyboard buffer.

Whenever a keystroke is available at the keyboard, the Keyboard Interface checks whether the Input Queue at Input Port is full. If the Queue is not full, then an interrupt is generated. After receiving an ACK from the processor the data is written into the input Queue.

Hardware Support for Fault Tolerance

WIMP has some basic hardware support for time redundant fault tolerant computing that allows:

- Track data values stored in the memory
- Input-Output port loopback
- Threads to interact using interrupts

SWFT: An instruction to track stores
The semantics of the SWFT instruction are as follows:

SWFT $RA, $RB, $RD

The contents of register RB are stored into the address pointed by register RA. In addition, the data (contents of $RB) are also posted into the output port indicated by the contents of register RD. If the write into the port is unsuccessful, flag(6) is cleared; if the write into the port is successful, flag(6) is set. The write into the store is unaffected with this status.

Input-Output port loopback

In addition, WIMP provides a path between the output ports 4 and 5 and the input ports port 4 and 5. Data values written into output port 4 can be read from the input port 4. Data values written into output port 5 can be read from the input port 5. This facilitates tracking data values written into the memory.

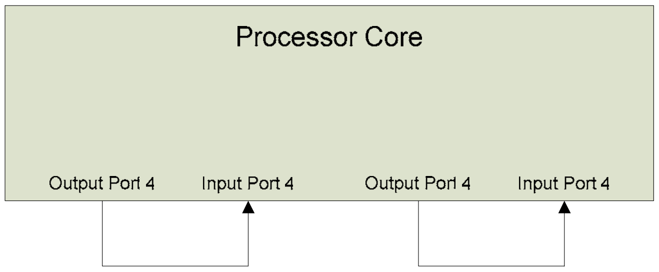


Figure 18: Port loopback

Threads interrupting threads

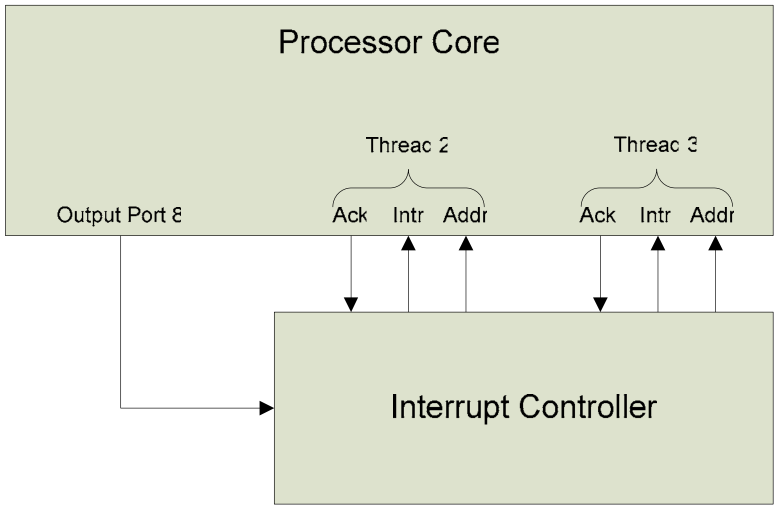


Figure 19: Threads interrupting threads

The output port 8 is dedicated for threads to interrupt other threads. Bits 0 through 3 interrupt threads 0 to 3. The interrupt controller is state machine shown in Figure 20. This decouples the data values at the output port and the actual interrupt protocol.

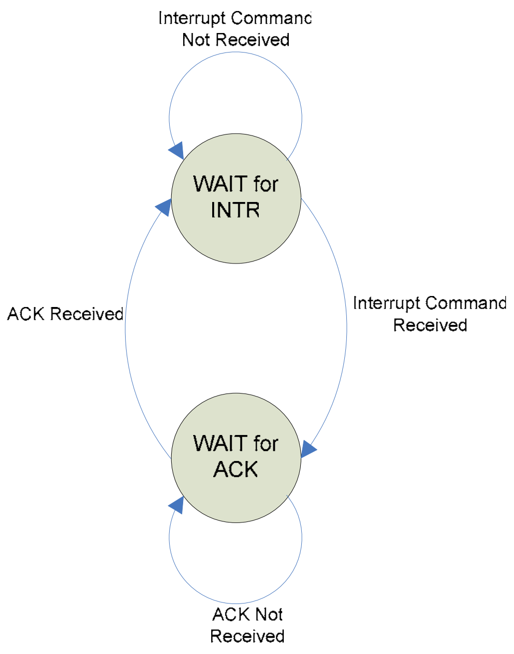


Figure 20: Interrupt controller state machine

WIMP Memory Interface and VGA

The prototype board comes along with two memory banks, one on the left and right bank of the FPGA. WIMP uses the right bank for instructions and the left bank for data. The processor talks to each of the memories through the memory interface provided at the course website. One interface is instantiated for each memory bank. The interface [written by Matt King] has the following signals on the memory and the processor side:

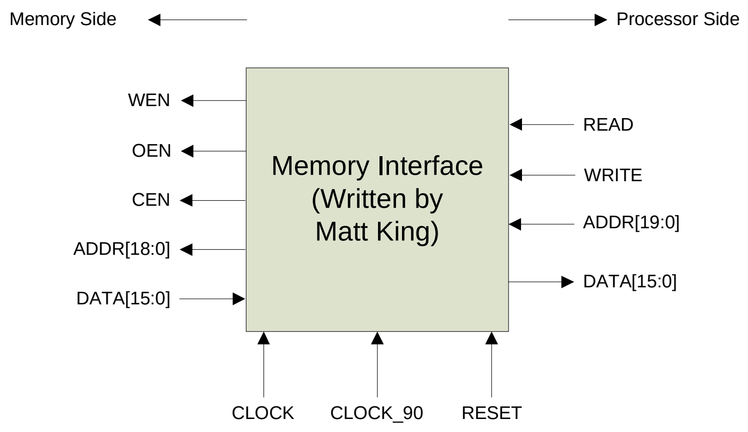


Fig 21: Memory interface on processor side and memory side

The interface allows either a read or a write in a single cycle, not both. The processor requires a read and a write simultaneously in the instruction memory. This allows programs to be loaded into the memory from the PC while instructions for other threads are being fetched. On the other hand, the left bank holds the video RAM as well as the data memory. Both need to be accessed in a single clock cycle; for data access and VGA refresh. For this reason, the memory interface is clocked at twice the processor clock rate that simulates a two port memory.

WIMP uses only the lower order 16-bits out of the 19-bits provided to address the memory since WIMP is a 16-bit processor. Figure 22 shows the data flow between the processor and the memory through the memory interface.

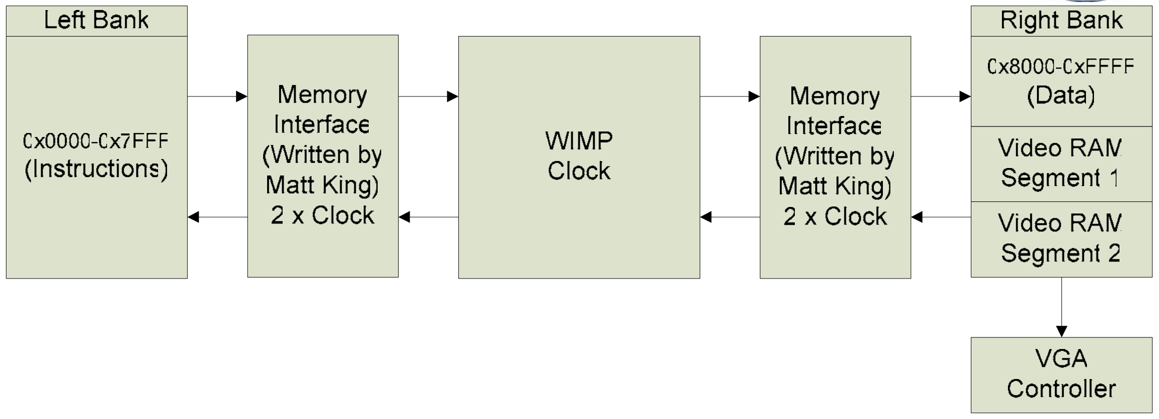


Figure 22: Data flow between memory and WIMP

VGA refresh is controlled by the VGA controller that was downloaded from the course web-page. This interface supports a resolution of 256x480 pixels at a frequency of 12.5MHz. Figure 23 shows a high level data flow between the processor and the VGA.

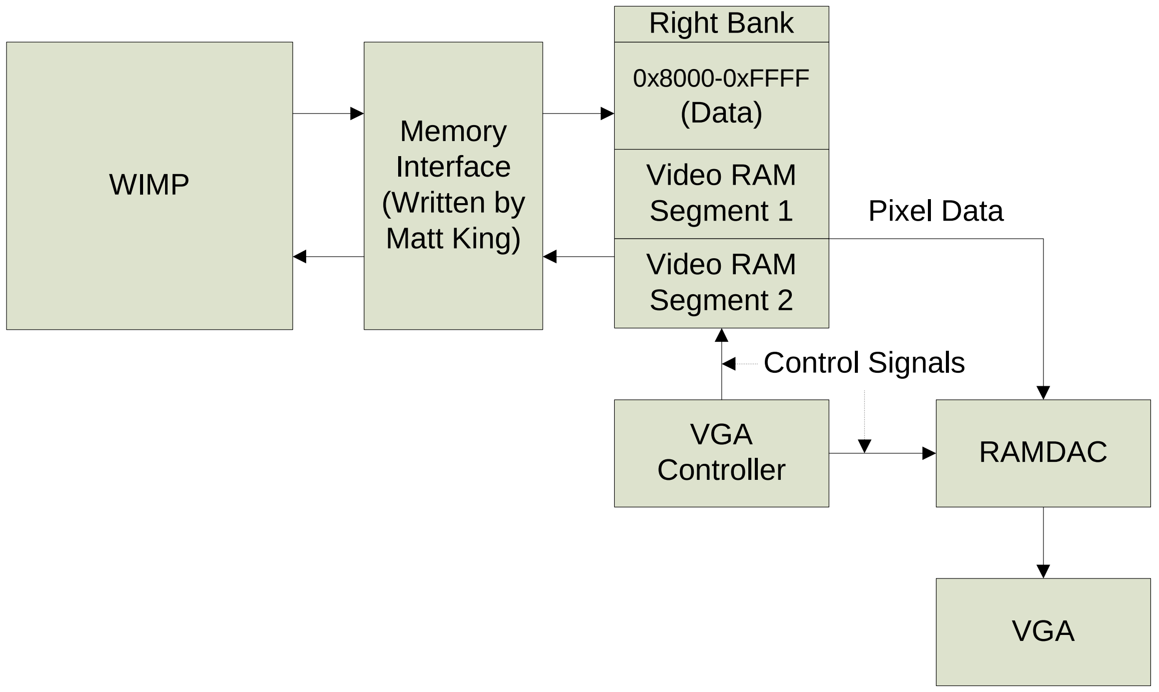


Figure 23: WIMP and VGA

The VGA controller generates the address corresponding to the scan lines of the VGA. The VGA controller also generates the HSYNC and VSYNC signals that control the VGA scanning. Additionally, the VGA controller initializes the RAMDAC with the 8-bit grey scale and color map.

Hardware Counter

WIMP has a 16-bit hardware counter that is accessible through the input and output ports 6. The commands to start/stop/reset/load the counter are given through the output port and counter value is read through the input ports.

Table 5: Controlling the hardware counter

VALUE	**COMMAND**
10xx	Clear counter
0800	Start counter
0400	Stop counter
02VV	Load VV into lower byte
03VV	Load VV into higher byte
	VV = 8-bit hex value

Example:

To clear the counter:

```
li        0x1000, $1              // Load clear command
write   5, 1, $1                  // Write into counter
```

To start the counter:

```
li        0x0800, $1              // Load start command
write   5, 1, $1                  // Write into counter
```

To read the counter:

```
read   5, 1, $1                   // Write into counter into register 1
```

The counter increments once every machine cycle, not every clock cycle, implying that the counter increments once for every four clock cycles. The counter value is inclusive of the start and the stop commands.

Example

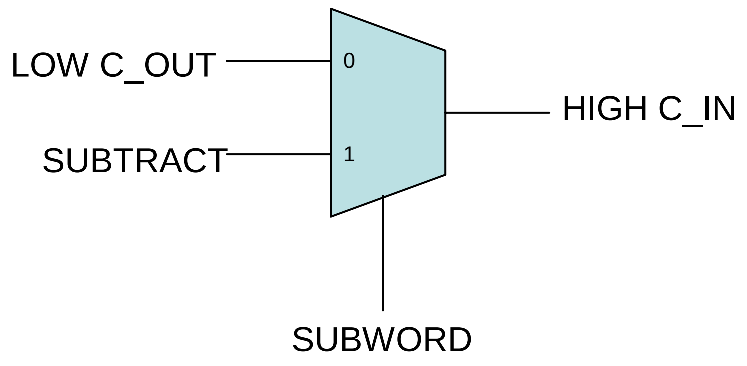
```
li        0x1000, $1              // Load clear command

write   5, 1, $1                  // Write into counter
li        0x0800, $1              // Load start command
write   5, 1, $1                  // Write into counter
nop
nop
li        0x0400, $1              // Load stop command
write   5, 1, $1                  // Write into counter
read   5, 1, $1                   // Write into counter value into register 1
```

The contents of register 1 at the end of this instruction sequence are 0x0008.

ALU

ALU Addition, Subtraction

The ALU for WIMP is specialized for subword instructions. To avoid using three adders (8-bit low subword, 8-bit high subword, 16-bit entire), we use two 8-bit adders and use some condition logic for propagating the carry bit. The following illustration depicts the simple logic involved to do this.



Subtraction is implemented by complementing the input of the second operand and setting the LSB carry in to high. For the adder, we use the synthesized adder generated from the verilog construct '+'. When the entire ALU is optimized for speed, it has a maximum combinational delay of around ~20 ns. Obviously using a core generated version of an adder wouldn't be much of an improvement as described in the multiply and accumulate section.

ALU Comparison

The comparison instructions have two output values; only one is relevant for 16-bit operating instructions. If the condition to be tested is true for both subwords, '11' will be outputted on the FLAG bus. The datapath carries these bits onto a section of a larger register file defined by other parts of the instruction.

All comparisons are computed in the typical manner, both normal and subword instructions. For example, to prove a number is greater than or equal to another, the most significant bit (or bits) is checked for its sign after subtracting the two operands.

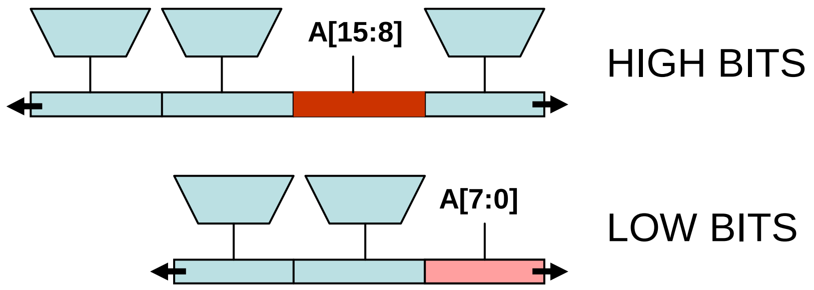
ALU Logical

WIMP implements four logical instructions: AND, OR, XOR, NOT. Since these operations are purely bitwise, the implementation is straightforward and subword support is intrinsic.

ALU Shifter & Rotator

WIMP uses both variable shifting and includes shifting for subword instructions. Variable shifting can be implemented with stacks of multiplexers. Given subword support, there are 18 possibilities for values that would need to be shifted in on the end of the two eight bit sections.

To eliminate this hassle, the data can be arranged in such a way that that we don't need to chain shifters. Instead, we can use one large shifter with multiplexed inputs to handle the various data that would need to be shifted over (as depicted in the illustration below).



WIMP does not use a core-generated version of these shifters; it needs to be pipelined and registered with a very fast clock. Moreover, it does not need to be fast or space efficient to a large degree since the multiplier is considerably slower.

Multiplier, MAC, and Permute Unit

The Multiplier will take one cycle to execute and will be reused for the multiply and accumulate instruction. In the second execute cycle, the results of the multiply will added to the register the result will be loaded into. All of the permute instructions will take one cycle to complete and will be forwarded to the second cycle. A MUX with forwarded control signals will determine the output of the whole unit. The multiply and MAC instructions are in Figure 24 below and the MIX, MUX, and copy instructions are in Figure 25 below.

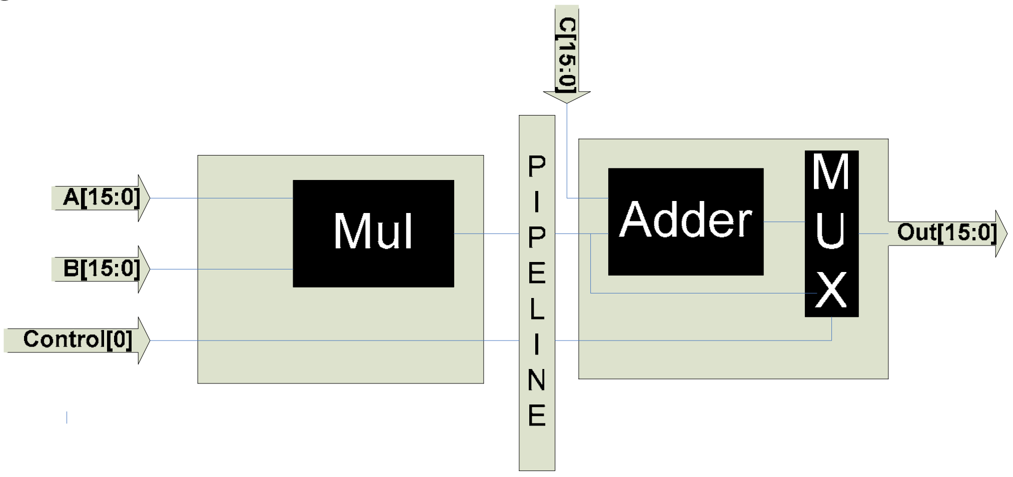


Figure 24: Multiply and MAC instruction

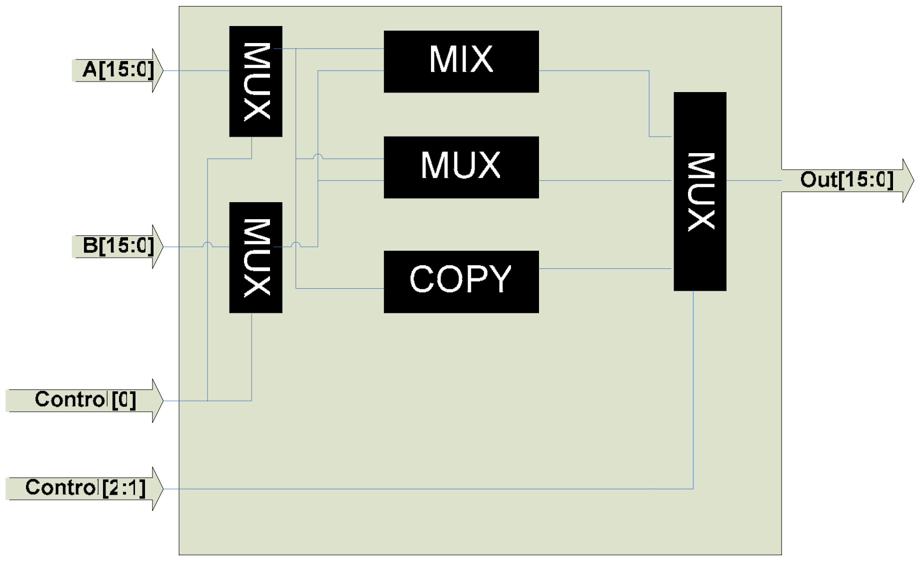


Figure 25: Mix, Mux and Copy instructions

Clocking

WIMP requires three different clocks; clock, 2 x clock and a 2 x clock phase shifted by 90 degrees. The processor runs at clock, while the register file in the processor runs at twice the clock rate (clock_2x = 2 x clock). The memory interface requires clock_2x and clock_2x shifted by 90 degrees. The board has only one external clock input at pin 89. All the required clocks are generated from this clock input using the clock DLL's present in the FPGA. Figure 26 shows how the clock DLL's and clock buffers are connected to get the three required clocks.

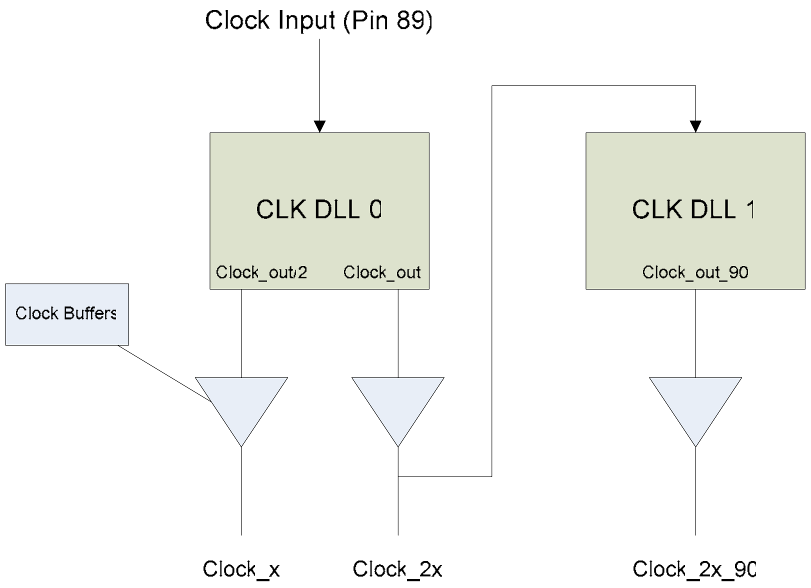


Figure 26: Clock generation for WIMP

The clock DLL's present in the FPGA has a clock input; the clock out follows the input clock, clock out/2 (that is half the input clock frequency and in phase with the input clock), and clock_90 (that is 90 degrees out of phase with the input clock). The clock_2x and clock_2x_90 are derived from the clock input through the clock DLL's. The clock_x that is used by the processor is derived from the clock input (pin 89) through clock out/2 of the clock DLL's.

Synthesis and Implementation

Initial Setup

WIMP is coded in verilog, the HDL (Hardware Description Language) recommended for the course. FPGA express, a synthesis tool provided by Synopsys was used to synthesize the processor. The following options were initially used for the synthesis (no optimizations)

Input clock	:	
Clock	:	30 ns
Clock_2x	:	15 ns
Clock_2x_90	:	15 ns
Area/Speed	:	Synthesis for area
Max effort	:	Medium

The resulting edf file was mapped to XCV800-HQ240-4 using the XILINX project manager tool. The only constraints provided during implementation were the pin constraints.

This resulting critical paths after implementation had the following delay:

Clock	:	34.4 ns (~29MHz)
Clock_2x	:	18.5 ns (~54MHz)
Clock_2x_90	:	15.5 ns (~64MHz)

On an average, 30% of the delay was due to logic, setup the rest was due to routing. This limits the clock to 27MHz and the clock_2x to 54MHz. This does not include the memory access time of 15ns and the FPGA pin delay of 5ns (total 20ns). Once these values are plugged in, clock is limited to 12.98MHz and clock_2x to 25.97MHz.

There were no issues with the resource utilization of the FPGA. Only 25% of the slices were used for logic. This implied the goal of the future optimizations was to improve speed.

A Note on Optimizations

WIMP was optimized in different phases. In the first phase, only the synthesis and implementation options were changed. The max fanout was made 8 and register duplication was enabled. In addition, the following changes were made:

Input clock	:	
Clock	:	20 ns
Clock_2x	:	10 ns
Clock_2x_90	:	10 ns
Area/Speed	:	Synthesis for speed
Max effort	:	High

The resulting critical paths after implementation had the following delay:

Clock	:	30.3 ns (~33MHz)
Clock_2x	:	16.2 ns (~62MHz)
Clock_2x_90	:	14.8 ns (~68MHz)

This limits clock to 31 MHz and clock_2x to 62 MHz without the memory access time of 20 ns. Once the memory access time is included, clock is limited to 14 MHz and clock_2x is restricted to 28 MHz.

In the second phase, the processor code itself was modified to increase clock frequency. The second phase consisted of

- Redundant code analysis
- Critical path analysis
    - Retiming – Moving logic around
    - Changing coding style
    - Changing data structures

Code Coverage Analysis

Synopsys VCS was used to find redundant code in the system. Benchmarks are executed on the processor while this tool monitors every line of the verilog code. The tool highlights:

- Lines in the code not used in the verilog code (Line Coverage)
- If-then-else structures not used in the system (Conditional Coverage)
- State not entered in the state machine (Functional Coverage)

This data is useful to reduce the code size to some extent so that the synthesis tool has less to work with.

Critical Path Analysis

The timing analyzer that comes along with the XILINX implementation tool was used to find the critical path of the system. The critical path was optimized using one or more of the following ways:

- Retiming – Moving logic around the pipes
- Changing coding style
- Changing data structures

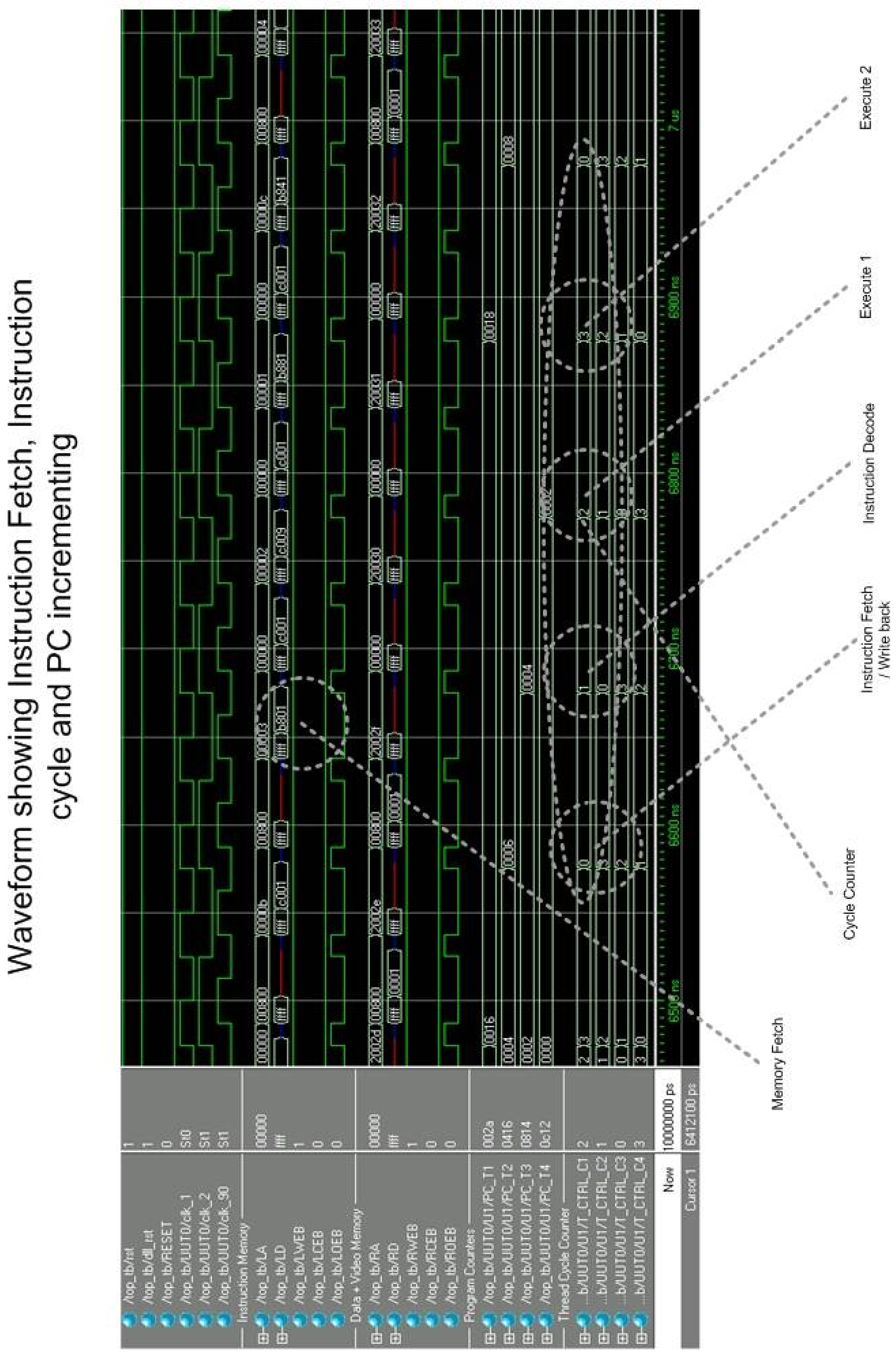
The top-level file was re-synthesized to find the next critical path. This process was continued till the improvements in the clock cycle time were negligible.

The resulting critical paths after implementation and some optimizations had the following delay:

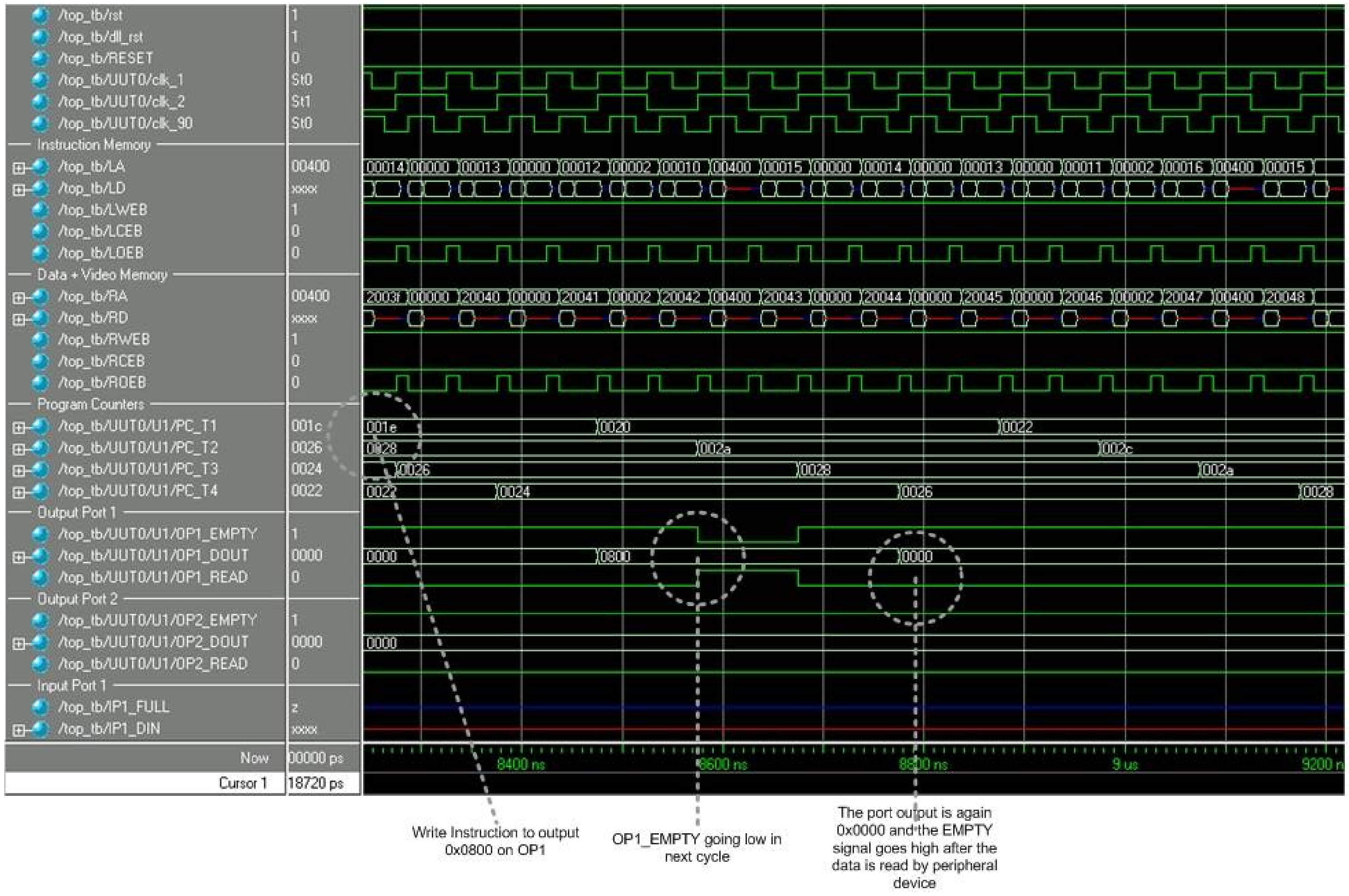
Clock	:	22.2 ns (~45MHz)
Clock_2x	:	12.2 ns (~83MHz)
Clock_2x_90	:	10.1 ns (~99MHz)

Again, these values do not include the memory access time. They are an indication of what the possible frequencies will be if there was perfect memory. Once the memory access time (20ns) are added, the clock frequency is limited to 15MHz and clock_2x frequency is limited to 30MHz.

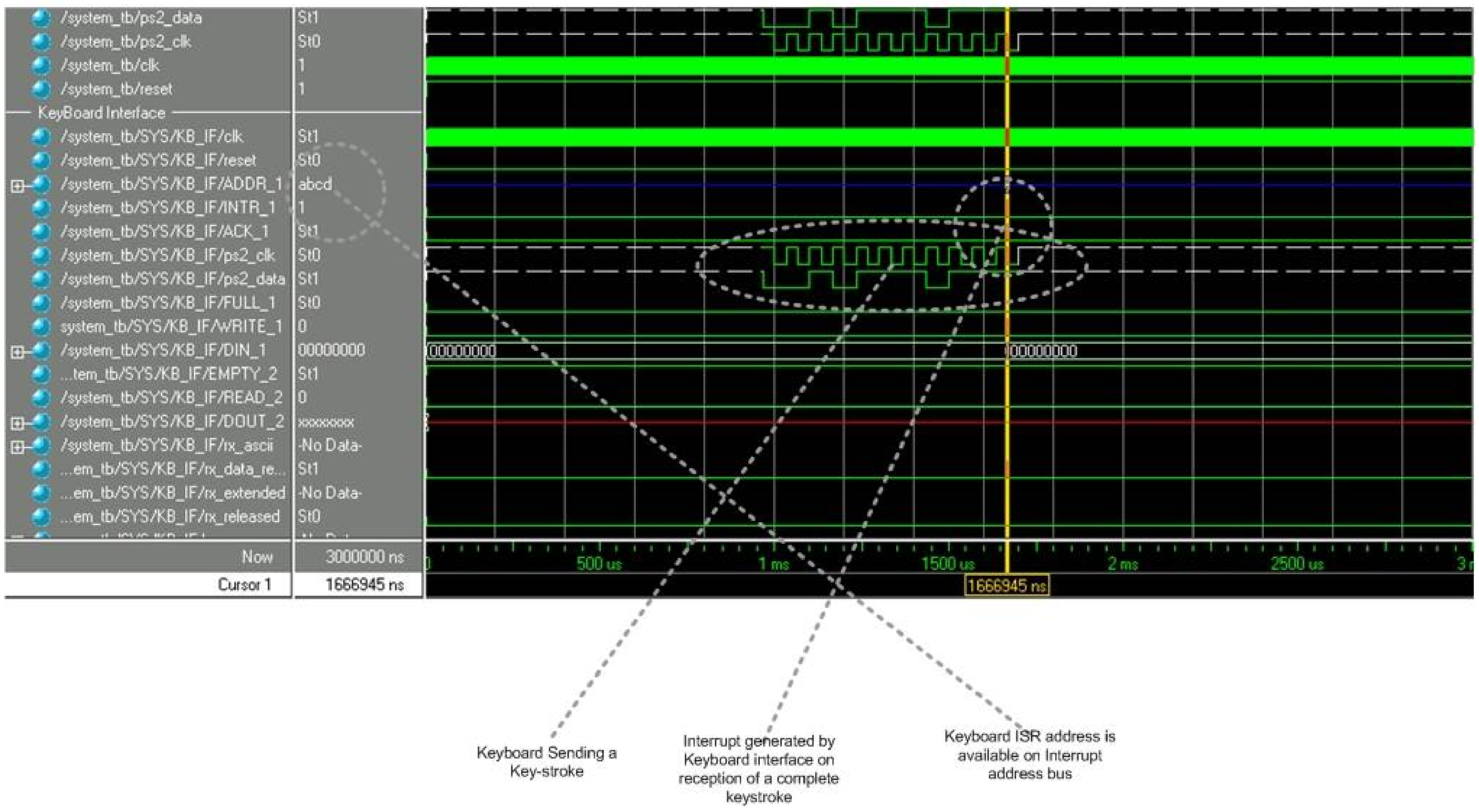
Waveforms showing important timings



Waveform showing Output Port timings



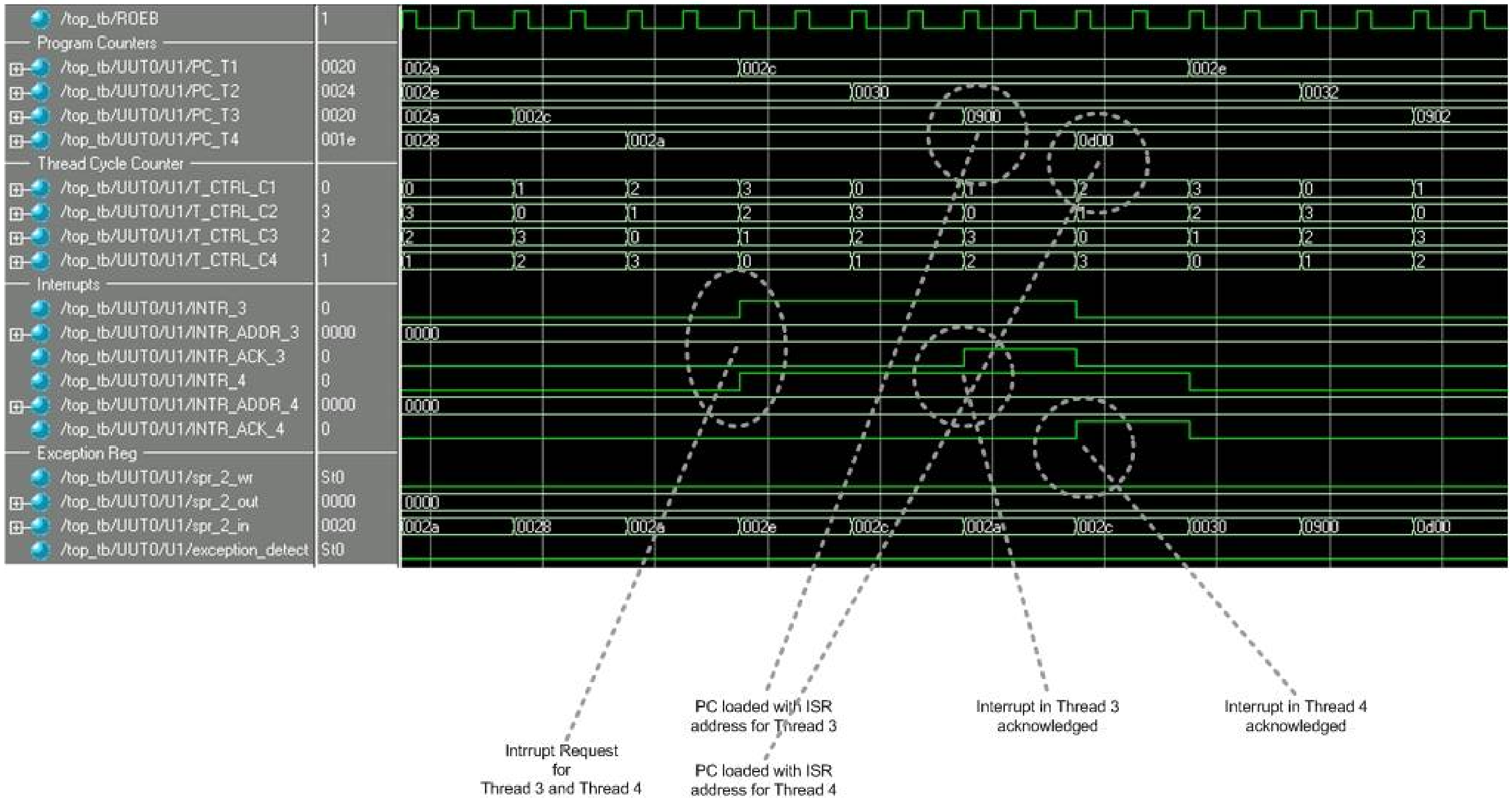
Waveform showing Keyboard Interface timings and Interrupt generation for key-stroke



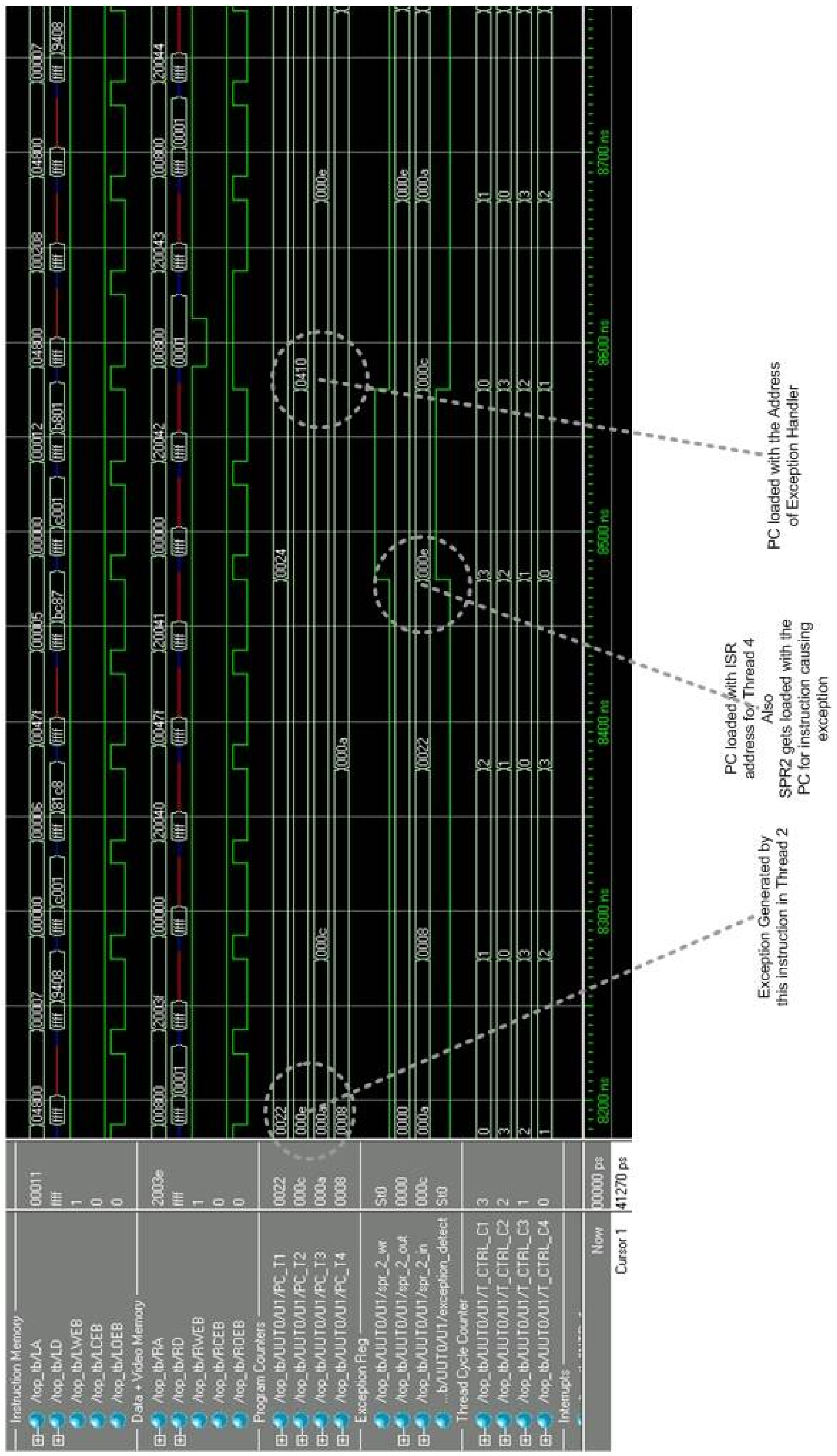
```
/system_tb/ps2_data              St1
/system_tb/ps2_clk               St0
/system_tb/clk                   1
/system_tb/reset                 1
KeyBoard Interface
/system_tb/SYS/KB_IF/clk         St1
/system_tb/SYS/KB_IF/reset       St0
/system_tb/SYS/KB_IF/ADDR_1      abcd
/system_tb/SYS/KB_IF/INTR_1      1
/system_tb/SYS/KB_IF/ACK_1       St1
/system_tb/SYS/KB_IF/ps2_clk     St0
/system_tb/SYS/KB_IF/ps2_data    St1
/system_tb/SYS/KB_IF/FULL_1      St0
system_tb/SYS/KB_IF/WRITE_1      0
/system_tb/SYS/KB_IF/DIN_1       00000000
...tem_tb/SYS/KB_IF/EMPTY_2      St1
/system_tb/SYS/KB_IF/READ_2      0
/system_tb/SYS/KB_IF/DOUT_2      xxxxxxxx
/system_tb/SYS/KB_IF/rx_ascii    -No Data-
...em_tb/SYS/KB_IF/rx_data_re... St1
...em_tb/SYS/KB_IF/rx_extended   -No Data-
...em_tb/SYS/KB_IF/rx_released   St0

Now                              3000000 ns
Cursor 1                         1666945 ns
```

00000000
00000000
500 us
1 ms
1500 us
2 ms
2500 us
1666945 ns
Keyboard Sending a Key-stroke
Interrupt generated by Keyboard interface on reception of a complete keystroke
Keyboard ISR address is available on Interrupt address bus

Waveform showing Interrupt Requests and Acknowledge timings


/top_tb/ROEB
Program Counters
/top_tb/UUT0/U1/PC_T1
/top_tb/UUT0/U1/PC_T2
/top_tb/UUT0/U1/PC_T3
/top_tb/UUT0/U1/PC_T4
Thread Cycle Counter
/top_tb/UUT0/U1/T_CTRL_C1
/top_tb/UUT0/U1/T_CTRL_C2
/top_tb/UUT0/U1/T_CTRL_C3
/top_tb/UUT0/U1/T_CTRL_C4
Interrupts
/top_tb/UUT0/U1/INTR_3
/top_tb/UUT0/U1/INTR_ADDR_3
/top_tb/UUT0/U1/INTR_ACK_3
/top_tb/UUT0/U1/INTR_4
/top_tb/UUT0/U1/INTR_ADDR_4
/top_tb/UUT0/U1/INTR_ACK_4
Exception Reg
/top_tb/UUT0/U1/spr_2_wr
/top_tb/UUT0/U1/spr_2_out
/top_tb/UUT0/U1/spr_2_in
/top_tb/UUT0/U1/exception_detect
Intrrupt Request for Thread 3 and Thread 4
PC loaded with ISR address for Thread 3
PC loaded with ISR address for Thread 4
Interrupt in Thread 3 acknowledged
Interrupt in Thread 4 acknowledged

Waveform showing Exception Handling in WIMP


Instruction Memory
/top_tb/LA
/top_tb/LD
/top_tb/LWEB
/top_tb/LCEB
/top_tb/LOEB
Data + Video Memory
/top_tb/RA
/top_tb/RD
/top_tb/RWEB
/top_tb/RCEB
/top_tb/ROEB
Program Counters
/top_tb/UUT0/U1/PC_T1
/top_tb/UUT0/U1/PC_T2
/top_tb/UUT0/U1/PC_T3
/top_tb/UUT0/U1/PC_T4
Exception Reg
/top_tb/UUT0/U1/spr_2_wr
/top_tb/UUT0/U1/spr_2_out
/top_tb/UUT0/U1/spr_2_in
...b/UUT0/U1/exception_detect
Thread Cycle Counter
/top_tb/UUT0/U1/T_CTRL_C1
/top_tb/UUT0/U1/T_CTRL_C2
/top_tb/UUT0/U1/T_CTRL_C3
/top_tb/UUT0/U1/T_CTRL_C4
Interrupts
Now
Cursor 1
8200 ns
8300 ns
8400 ns
8500 ns
8600 ns
8700 ns
Exception Generated by this instruction in Thread 2
PC loaded with ISR address for Thread 4 Also SPR2 gets loaded with the PC for instruction causing exception
PC loaded with the Address of Exception Handler