PIPELINED RISC PROCESSOR WITH INTERRUPT HANDLING

Project Report By:
Sameer Sondur

Project Requirements

The problem statement and requirements for the project are:

Design, Implement and Test a RISC (Reduced Instruction Set Computer) Processor with the following specifications:

- 16 bits Instruction Set
- 4 stage pipelining
- 4 external interrupts
- Non preemptive Interrupt Handler
- Sixteen 8-bit registers
- 8 bits wide data memory
- All address buses are 8 bits wide

The processor design and simulation should be done using Mentor Graphics ModelSim software. The Instruction Set for the processor is predefined.

The processor was designed according to the given problem statement with 16 bit instructions, 256 Bytes of Instruction memory, 256 Bytes of data memory. It is a RISC processor with four stage pipeline namely Fetch, Decode, Execute, WriteBack. It has got a non-preemptive interrupt handler with four external interrupt request query lines.

Instruction Set

The processor has an instruction set of 27 instructions that include arithmetical and logical instructions and load store instructions. The Instruction set is briefly given as follows:

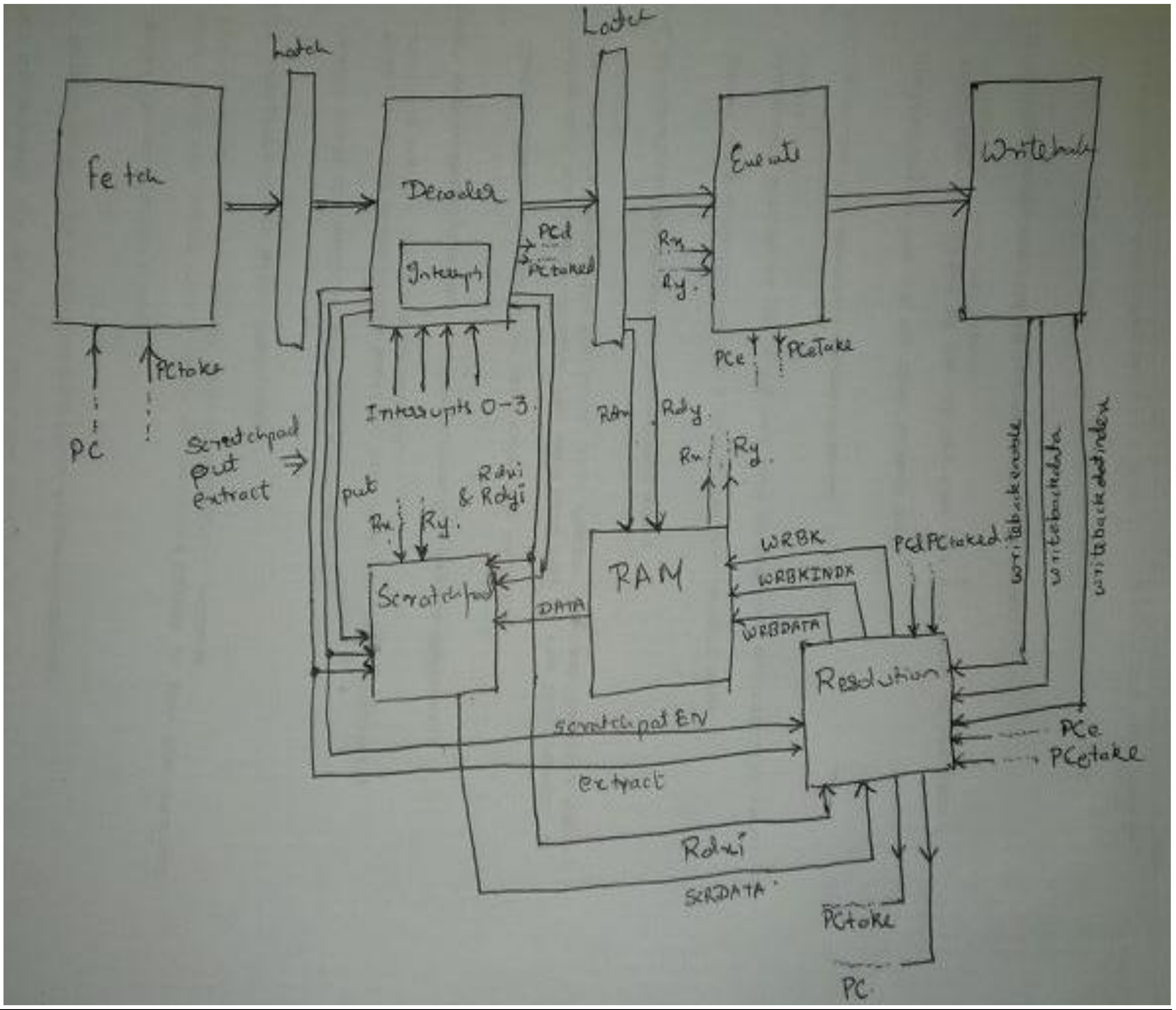
INSTRUCTION	TYPE	OPCODE/SEL [15 - 8]		MNEMONICS	OPERATION
NO OPERATION	R	0000	XXXX	NOP	PC<=PC+1
ADD IMMEDIATE	IM	0001	R[x] addr	ADD R[x],#data	R[x]<=R[x]+#data
ADD	R	0010	0000	ADD R[x],R[y]	R[x]<=R[x]+R[y]
SUBTRACT	R	0010	0001	SUB R[x],R[y]	R[x]<=R[x]-R[y]
INCREMENT	R	0011	0000	INC R[x]	R[x]<=R[x]+1
DECREMENT	R	0011	0001	DEC R[x]	R[x]<=R[x]-1
SHIFT LEFT	R	0100	0000	SHL R[x],R[y]	R[x]=R[x]<<R[y]
SHIFT RIGHT	R	0100	0001	SHR R[x],R[y]	R[x]=R[x]>>R[y]
LOGICAL NOT	R	0101	0000	NOT R[x]	R[x]<=NOT R[x]
LOGICAL NOR	R	0101	0001	NOR R[x],R[y]	R[x]<=R[x] NOR R[y]
LOGICAL NAND	R	0101	0010	NAND R[x],R[y]	R[x]<=R[x] NAND R[y]
LOGICAL XOR	R	0101	0011	XOR R[x],R[y]	R[x]<=R[x] XOR R[y]
LOGICAL AND	R	0101	0100	AND R[x],R[y]	R[x]<=R[x] AND R[y]
LOGICAL OR	R	0101	0101	OR R[x],R[y]	R[x]<=R[x] OR R[y]
CLEAR	R	0101	0110	CLR R[x]	R[x]<=0
SET	R	0101	X111	SET R[x]	R[x]<=1
SET IF LESS THAN	R	0101	1111	SLT R[x],R[y]	IF R[x]<R[y] THEN R[x]=1
MOVE	R	0101	1000	MOV R[x],R[y]	R[y]<=R[x]
ENABLE INTERRUPTS	R	0111	XXXX	EI intcode	lowest 4 bits enable IR3~IR0
LOAD INDIRECT	EM	1000	XXXX	LD R[x],R[y]	R[x]<=MEMR[y]
STORE INDIRECT	EM	1001	XXXX	LD R[x],R[y]	MEM[R[x]]<=R[y]
LOAD REGISTER	EM	1010	R[x] addr	LD R[x],#address	R[x]<=MEM[#address]
STORE REGISTER	EM	1011	R[x] addr	STR #address,R[x]	MEM[#address]<=R[x]
JUMP	BR	1100	XXXX	JMP #address	PC<=#address
BRANCH IF ZERO	BR	1101	XXXX	JZ #offset	IF R[x]==0, PC<=PC+offset
BRANCH IF NOT ZERO	BR	1110	XXXX	JNZ #offset	IF R[x]<>0, PC<=PC+offset
RETURN FROM INTERRUPT	BR	1111	XXXX	RETI	PC<=PC'+1

VHDL Files

The vhdl files in our project are organized in following files based on the pipeline architecture and the interrupts. The files are:

- Fetch Stage - fetch.vhd
  1. Instruction Memory - instruction_memory.vhd
  2. Program Counter - pc.vhd, pcresolution.vhd
- Fetch-Decode Latch - fetdec.vhd, lafetch.vhd
- Decoder Stage - decoder.vhd, combo.vhd
- Decode-Execute Latch - latchdec.vhd
- Execute Stage - execute.vhd
  1. Register Bank - ram.vhd
  2. ALU - aluvala.vhd
- Execute-Writeback Latch - laexec.vhd
- Writeback Stage - writeback.vhd
  1. Multiplexer - MUX2_1.vhd
  2. Data Memory - data_mem.vhd
- Interrupt handler - scrathpad.vhd
- Top Level Processor - interrupttrail1.vhd
- Pipelined Processor - pipelined1.vhd

Top Level Block Diagram



Fetch Stage

On every positive edge of an input clock cycle, the fetch stage increments the value of the program counter (which is set to zero during the processor bringup). This value of program counter is used to fetch the instructions that are stored in the 256*16 instruction memory. If the flow of execution is changed by either ALU stage or the Interrupt handler, the fetch stage is informed of this modified program counter with the control signal ctrl. When ctrl changes its state from low to high, the value of pc is overwritten with the modified value of program counter that is input (pc_in). The fetch stage thus fetches the new instruction from the instruction memory according to the modified value of PC.

1) Program Counter - pc.vhd

Input Ports :

- **clk : in std_logic -** The input clock to system. In the fetch stage, the PC is incremented by 1 each time an upper edge of the clock is received. Thus in the first clock cycle of the fetch stage, the PC becomes "00000001" and hence points to the first instruction in the Instruction Memory

- **PC_in : in std_logic_vector(7 downto 0) -** The new/modified value of Program Counter. This value can either be input from ALU stage or from Interrupts when these blocks want to alter the flow of execution.

- **PC_ctrl : in std_logic -** The control bit that decides if the value in PC_in should be the value of new program counter.

Output Ports:

- **PC_out: out std_logic_vector(7 downto 0) -** The value to be sent to Instruction Memory which will then in turn fetch the instruction stored at the address of the Program Counter and send it to Decode stage.

2) Instruction Memory: instruction_memory.vhd

```
package arr is
        type array_rom is array (0 to 255) of std_logic_vector(15 downto 0); -- subtype for ROM
end package arr;
```

A two dimensional array for ROM which is 256 locations x 16 Bits. This package is used to define a signal ROM of type array_rom in instruction_memory.vhd which acts as the instruction memory for the processor.

Input Ports:

- **pc: in std_logic_vector(7 downto 0) -** Take the input program counter – the pointer to the current instruction in Instruction Memory.

Output Ports:

- **instruction: out std_logic_vector(15 downto 0) -**16 bit wide output port. Outputs the instruction word pointed by program counter to the decode stage.

3) Fetch : Fetch.vhd -- Top level entity for Fetch Stage

Input Ports:

- **clk : in std_logic -** The input clock to system.
- **ctrl : in std_logic -** Select Line for pc_in.
- **pc_in : in std_logic_vector(7 downto 0) -** Modified PC value from either ALU or Interrupt Handler.

Output Ports:

- **pc_out : out std_logic_vector(7 downto 0)-** Current value of PC forwarded to decode stage to resolve new PC if any change in execution sequence is to be made.
- **instruction : out std_logic_vector (15 downto 0) -** 16 bit instruction fetched from the location in Instruction Memory ROM indexed by the current value of Program Counter.
- **clk_out: out std_logic -** The clock signal output to next stage.

Decode Stage

The second cycle of the pipeline is the Decode stage where the Instruction fetched from the instruction memory is decoded to different signals and flags which is fed to the execute stage for the execution of the instruction. The input to the Decode stage is the 16 bit instruction that is fetched from the fetch stage.
The Decode stage decodes this 16 bit instruction based on the 'opcode' of the instruction which is the four MS bits in the instruction. Based on the instruction, different signal are assigned and different flag values are set. These flags and signals forms the output of the decoder which are the inputs for the execute stage. The decoder also handles the interrupts. The external interrupts to the processor are received in the decoder and based on the 'interrupt enable' signal received earlier by the decoder, the final interrupt values are set.

1) Decode - decode.vhd
Input Ports:

- **clk : in std_logic -** System Clock
- **zflg: in std_logic -** Zero flag, it is used for control signals in decoder stage, it is input from execute stage

- **irq0 : in std_logic** - Interrupt Request Query0, External interrupt request line, when this line goes high, the ISR for IRQ 0 is located at the index 128 in instruction memory. The execution flow jumps to this location in instruction memory ie. PC <= 128

- **irq1: in std_logic -** Interrupt Request Query1 , External interrupt request line, when this line goes high, the ISR for IRQ 1 is located at the index 153 in instruction memory. The execution flow jumps to this location in instruction memory ie. PC <= 153

- **irq2: in std_logic** - Interrupt Request Query2, External interrupt request line, when this line goes high, the ISR for IRQ 2 is located at the index 178 in instruction memory. The execution flow jumps to this location in instruction memory ie. PC <= 178

- **irq3: in std_logic** - Interrupt Request Query3 External interrupt request line, when this line goes high, the ISR for IRQ 1 is located at the index 203 in instruction memory. The execution flow jumps to this location in instruction memory ie. PC <= 203

- **instruction: in std_logic_vector(15 downto 0)** - This is the current instruction to be decoded and executed which is input from the fetch stage

- **PCi: in std_logic_vector(7 downto 0)** - This is the current value of PC which is input from the fetch stage

Output Ports:

- **Immed : out std_logic_vector(7 downto 0)** - This is the 8 bit immediate address in the instruction

- **rlsaddress : out std_logic_vector(7 downto 0) -** This is the address generated from decoder for load or store instructions which is fed to write back stage

- **Rdx: out std_logic_vector(3 downto 0) -** Index to register bank generated from decode stage

- **rdy : out std_logic_vector(3 downto 0)** - Index to register bank generated from decode stage

- **PC : out std_logic_vector(7 downto 0)** - The modified value of Program counter which is sent back to the fetch stage to change the execution flow of the program.

- **offset: out std_logic_vector(3 downto 0) -** 3 bit Offset for the jump if zero and jump if not zero instructions

- **aluoptodram: out std_logic_vector(3 downto 0)- T**his is the output signal which tells write back stage if the operation is load indirect, store indirect, load register or store register

- **aluop: out std_logic_vector(4 downto 0) -** This is the opcode for the ALU operations which is sent to the execute stage to decide the type of operation

- **aluopse, wr, ryimmed,regindir,put,extract, scratchenable: out std_logic -** They are the control signals which are used to select the particular functionality

2) **Scratchpad : scratchpad.vhd**

Input Port:

- **clk: in std_logic -** System Clock
- **scratchenable :in std_logic -** This signal enables the scratchpad, this is input from the decode.vhd
- **extract :in std_logic** - When extract is 0 , the data is pushed into scratchpad. When extract is 1, the data is popped from the scratchpad
- **put:in std_logic :** When put is 1, data is pushed into scratchpad, when put is 0 the data is popped from the scratchpad
- **r0i: in std_logic_vector(7 downto 0);**
- **r1i: in std_logic_vector(7 downto 0) :**Register bank values to the scratchpad when pushed
- **pci: in std_logic_vector(7 downto 0):** The current value of program counter that is the state of cpu that needs to be preserved before the interrupt is serviced
- **rout:out std_logic_vector(7 downto 0):**Register bank output values when scratchpad is popped
- **pco:out std_logic_vector(7 downto 0):**The value of pc that needs to be restored when the interrupt is serviced and the scratchpad is popped
- **pctake: out std_logic:** This is the control signal for PC when scratchpad is popped. When this signal is hi, the value in pco is written into the current state of cpu.

Execute Stage

Execute Stage receives the output from the decode stage and it performs the execution of the Instruction.
The execute stage performs various operations like ADD, SUBTRACT, INCREMENT, DECREMENT, SHIFT, LOGICAL OPERATIONS, CLEAR, SET, LOAD, STORE, JUMP etc. All these operations are done with the help of the signals set from the decoder in the decode stage, based on the

'OpCode'. After the execution of the instruction, the result is send to the WriteBack Stage for further operation. The value is written back to the first operand of the operation, in case of an ALU operation.

Input Ports:

- **Rx,Ry: in std_logic_vector(7 downto 0) -** These are the operands to ALU, They are input from the decode stage
- **Immed: in std_logic_vector(7 downto 0) -** Immediate value input from decode stage
- **PCi: in std_logic_vector(7 downto 0) -** Current program counter input from decode stage
- **aluop: in std_logic_vector(4 downto 0) -** Opcode from the decode stage used to select the operation in ALU
- **aluoptodrami :in std_logic_vector(3 downto 0)** - This is the input signal from decode stage which specifies if the operation is load indirect, store indirect, load register or store register
- **ryimm : in std_logic** - This signal specifies if the value on ry is immediate or not
- **regindiri: in std_logic** - This signal specifies if this is an register indirect operation or not
- **offset: in std_logic_vector(3 downto 0)** -The offset for the jump instruction

Output Ports:

- **rwrite : out std_logic_vector(3 downto 0) -**This is the register index to writeback stage. That is the location in the register file at which the writeback data is stored.
- **aluoptodramo : out std_logic_vector(3 downto 0)** - This is the output signal from execute stage to writeback stage which specifies if the operation is load indirect, store indirect, load register or store register
- **aluout: out std_logic_vector(7 downto 0)** - The result from the ALU operation
- **PCo: out std_logic_vector(7 downto 0)** - The modified value of PC for Jump instructions, Sent to the fetch stage to alter the flow of execution
- **rxo, ryo: out std_logic_vector(7 downto 0)** - Rx, Ry outputs sent to writeback stage
- **zflag, wren,pctake,regindiro: out std_logic** - Various control signals

WriteBack Stage

WriteBack stage stores the values the output of the execute stage in register. The value could be either from ALU or Data Memory, depending on Register Type or Load/Store Instruction.
The multiplexer in writeback stage gets the values Register address and Indirect address and the select line for the multiplexer decides if register addressing or indirect addressing is used. This address is then input to the Data memory which inturn fetches correct data from the index provided by the output of the multiplexer.
The other input for the data memory is Register address ‘rx’, if the operation is load or store is decided by the 4 bit signal alu opcode which comes from decode stage.
If it is a load instruction, then the data is fetched from the data memory and put on the bus, if not the output is selected from ALU.
Either of these two outputs, whichever selected will be written back to register bank if the write enable line is high. The index of the location in register bank at which this output is to be written back is generated in the writeback stage.

1) 2:1 Multiplexer : MUX2_1.vhd

Input Port:

- **Ry : in std_logic_vector(7 downto 0) := (others => '0') -** One of the inputs of the 2 :1 multiplexer. Either of these two get selected as the address to Data Memory. Ry is the 8 bit direct address.

- **IM : in std_logic_vector(7 downto 0) := (others => '0') -** The other input to the mux is IM. This 8 bit address to Data Memory comes if it is an indirect address. This address comes from the ALU

- **SEL : in std_logic := '0'** - Select line for the 2:1 multiplexer.

Output Port:

- **To_ALU : out std_logic_vector(7 downto 0) := (others => '0') -** This is the output of the 2_1 multiplexer. Depending on either direct/indirect addressing mode being used, one of the Ry or IM are selected as the output of the multiplexer which outputs the address to Data memory.

2) Data Memory – data_mem.vhd

Input Port:

- **Rls : in std_logic_vector(7 downto 0)-** RLS is the 8 bit direct address that goes as one of the inputs in the 2_1 multiplexer instantiation in data memory.

- **Ry : in std_logic_vector(7 downto 0) -** Ry is the 8 bit indirect address that goes as one of the inputs in the 2_1 multiplexer instantiation in data memory. .

- **Reg_Ind : in std_logic -** Reg_Ind is the select line for the 2_1 multiplexer instantiated in data memory.

- **ALUop : in std_logic_vector(3 downto 0) -** ALUop is the 4 bit signal form Decode stage. ALUop stands for ALU opcode which decides if the operation is Lode/Store, Direct/Indirect.

- **memd : in std_logic_vector(7 downto 0) -** This is the 8 bit input to data memory, it's output from register bank

Output Port:

- **MEM_op : out std_logic_vector(7 downto 0)) -** This is the 8 bit output of data memory.

3) Write Back – writeback.vhd – High Level Entity of Write Back Block

Input Port:

- **clk : in std_logic -** This is the system clock input signal.

- **ALUop_wr : in std_logic_vector(3 downto 0) := (others => '0') -** ALUop is the 4 bit signal form Decode stage. ALUop stands for ALU opcode which decides if the operation is Lode/Store, Direct/Indirect.

- **Reg_Ind_Wr : in std_logic := '0' -** Reg_Ind_Wr is the select line for the 2_1 multiplexer instantiated in data memory.

- **Rls_wr : in std_logic_vector(7 downto 0) := (others => '0') -** RLS is the 8 bit direct address that goes as one of the inputs in the 2_1 multiplexer instantiation in data memory

- **Ry_wr : in std_logic_vector(7 downto 0) := (others => '0'); -** Ry is the 8 bit indirect address that goes as one of the inputs in the 2_1 multiplexer instantiation in data memory.

- **Rx_wr : in std_logic_vector(7 downto 0) := (others => '0') -** This is the 8 bit input to data memory, it's output from register bank

- **ALUout_wr : in std_logic_vector(7 downto 0) := (others => '0') -** This is the 8 bit output of the ALU unit.

Output Port:

- **WrBk_out : out std_logic_vector(7 downto 0) := (others => '0') -**This is the 8 bit data output from Data memory to be written back to Register bank.

- **Wen_out : out std_logic := '0' -** This is the control signal for WrBk_out, when hi, the data from data memory will be written to the register bank

- **Wr_out : out std_logic_vector(3 downto 0) := (others => '0'))** - Index to the location in register bank where the data from Data memory is to written

Conclusion

The VHDL model of 16 bit RISC processor was successfully implemented. The above waveforms match the expected output. **Each and every project requirement was met.**