Computer Architecture Project

Objective

To design and implement a simple 5-stage pipelined processor, **Harvard Architecture (Two memories)**. The design should conform to the ISA specifications described in the following sections.

Introduction

The processor in this project has a RISC-like instruction set architecture. There are eight 4-byte general purpose registers; R_0, till R7. Another two specific registers, One works as a program counter (PC). The other works as a stack pointer (SP); and hence; points to the top of the stack. The initial value of SP is (**2^12-1**). The memory address space is **4 K of 16-bit** width and is word addressable. (N.B. word = 2 bytes). The data bus between memory and the processor is **16-bit for instruction memory and 32-bit widths for data memory.**

When an interrupt occurs, the processor finishes the currently fetched instructions (instructions that have already entered the pipeline). The address of the next instruction (in PC) is saved on top of the stack, and the PC is loaded from address [2-3] of the memory (the address takes two words). To return from an interrupt, an RTI instruction loads PC from the top of the stack, and resumes the original program. **Take care of corner cases like Branching and Calling.**

ISA Specifications

1. **Registers**
   a. R[0:7]<31:0> ; Eight 32-bit general purpose registers
   b. PC<31:0>   ; 32-bit program counter
   c. SP<31:0>   ; 32-bit stack pointer
   d. CCR<3:0>   ; Condition code register
      i. Z<0>:=CCR<0> ; Zero flag, change after arithmetic, logical, or shift operations
      ii. N<0>:=CCR<1> ; Negative flag, change after arithmetic, logical, or shift operations
      iii. C<0>:=CCR<2> ; Carry flag, change after arithmetic, or shift operations. (logical and, or, xor ,...etc don't change this flag)
      iv. O<0>:=CCR<3> ; Overflow flag, changes after arithmetic operations. (logical and, or, xor ,...etc don't change this flag)

2. Input-Output

a. IN_PORT<31:0>   ; 32-bit data input port

b. OUT_PORT<31:0>   ; 32-bit data output port

c. INT_In<0>   ; Single, non-maskable interrupt signal

d. RESET_IN<0>   ; Reset signal

e. Exception_out<0>   ; exception is raised in two cases (overflow or accessing wrong address)

3. Instructions

Mnemonic	Function
ONE Operand	
NOP	PC <- PC+1
NOT Rdst	NOT value stored in register Rdst R[Rdst] ← 1's Complement(R[Rdst]); **Update the flags as appropriate**
NEG Rdst	Negate value stored in register Rdst R[Rdst] ← 0 – R[Rdst] **Update the flags as appropriate**
INC Rdst	Increment value stored in Rdst R[Rdst] ←R[Rdst] + 1; **Update the flags as appropriate**
DEC Rdst	Decrement value stored in Rdst R[Rdst] ←R[Rdst] – 1; **Update the flags as appropriate**
OUT Rdst	OUT.PORT ← R[Rdst]
IN Rdst	R[Rdst] ←IN.PORT
TWO Operands	
MOV Rdst, Rsrc	Move the value of Rsrc 1 in Rdst flag shouldn't change
SWAP Rdst, Rsrc1	Store the value of Rsrc 1 in Rdst and the value of Rdst in Rsc1 flag shouldn't change
ADD Rdst, Rsrc1, Rsrc2	Add the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst **Update the flags as appropriate**

Instruction	Description
ADDI Rdst, Rsrc1, Imm	Add the values stored in registers Rsrc1 to Immediate Value and store the result in Rdst **Update the flags as appropriate**
SUB Rdst, Rsrc1, Rsrc2	Subtract the value stored in registers Rsrc2 from value stored in registers Rsrc1and store the result in Rdst **Update the flags as appropriate**
SUBI Rdst, Rsrc1, Imm	Subtract the immediate value from Rsrc1and store the result in Rdst **Update the flags as appropriate**
AND Rdst, Rsrc1, Rsrc2	AND the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst **Update the flags as appropriate**
OR Rdst, Rsrc1, Rsrc2	**OR the values stored in registers Rsrc1, Rsrc2** **and store the result in Rdst** **Update the flags as appropriate**
XOR Rdst, Rsrc1, Rsrc2	XOR the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst **Update the flags as appropriate**
CMP Rsrc1, Rsrc2	Compare the values stored in registers by subtracting Rsrc2 from Rsrc1 and **Update Zero & Negative flag only**
MEMORY Operations	
PUSH Rdst	DataMemory[SP--] ← R[Rdst];
POP Rdst	R[Rdst] ← DataMemory[++SP];
LDM Rdst, Imm	Load immediate value (16 bit) to register Rdst R[Rdst] ← {0,Imm<15:0>}
LDD Rdst, EA(Rsrc1)	Load value from memory address EA to register Rdst R[Rdst] ← M[EA+R[Rsrc1]]; **EA is a 16 bit signed number**
STD Rdst, EA(Rsrc1) ~~Rdst~~ Rsrc1 ~~Rsrc1~~ Rsrc2	Store value in register Rsrc to memory location EA M[EA+R[Rsrc1]] ←R[Rsrc]; Rsrc2 Rsrc1 **EA is a 16 bit signed number** When Rsrc1 Contains Value to be stored in the memory, and Rsrc2 Contains value used in memory address calculation
PROTECT Rsrc	Protects memory location pointed at by Rsrc (won't be affected by later store operations and raise exception)
FREE Rsrc	Frees a protected memory location pointed at by Rsrc and resets its content
BRANCHING	
JZ Rdst	Jump if zero If (Z=1): PC ←R[Rdst];(Z=0)
JMP Rdst	**Jump** **PC <- R[Rdst]**
CALL Rdst	(DataMemory[SP] ← PC + 1; sp-=2; PC ← R[Rdst])

RET	sp+2, PC ←DataMemory[SP]
RTI	sp+2; PC ← DataMemory[SP]; Flags restored **from stack as well**
Input signals	
Reset	PC<-{M[1],M[0]}

Interrupt	DataMemory[SP]<-PC;
	SP-=2;
	DataMemory[SP]<-CCR;
	SP-2 ;
	PC<-{M[3],M[2]};
	//you are free to make M as data memory or instruction memory
	//you might want to make interrupt and instructions with several steps with FSM

Phase1 (40%)

- **Phase1 Due Date:** Week 11

- **Requirement: Design and Implement:**
  - **Printed Report Containing: (20%)**
    - Instruction format of your design
      - Opcode of each instruction
      - Instruction bits details
    - Control Signal Table
      - Create a table that contains **all** the control signals vs Instructions. For each instruction mark the corresponding control signals.
    - Schematic diagram of the processor with data flow details.
      - ALU / Registers / Memory Blocks / control block / Muxes / Adders outside ALU / etc…
      - Dataflow Interconnections between Blocks & its sizes. (show control buses, data buses, address buses)
      - PC and SP update circuits
    - Pipeline stages design
    - Pipeline registers details (Size, Input, Connection, …)
    - Pipeline hazards and your solution including
      - Data Forwarding
      - One-bit global Branch Prediction
  - **Code : (20%)**
    - Implement and integrate your architecture without hazards for the given

instructions only

- VHDL Implementation of each component of the processor
- VHDL file that integrates the different components in a single module
- Instructions:

a.	NOT	d.	LDM
b.	DEC	e.	MOV
c.	OR	f.	CMP

- Simulation Test code that reads a program file and executes it on the processor.
  - Setup the simulation wave using a do file, and show the following signals ONLY: Clk,Rst,PC, Registers Values, In port, Out port, MemoryOutput, Flags
  - Load Memory File & Run the given test program

Phase2 (5%)

- **Phase2 Due Date:** Week 12.
- **Requirement:** Assembler
  - Assembler code that converts assembly program (Text File) into machine code according to your design (Memory File)

Phase3 (55%)

- **Project Due Date:** Week 14
- **Requirement:** Full processor, Implement and integrate your architecture
  - VHDL Implementation of each component of the processor
  - VHDL file that integrates the different components in a single module
  - Report that contains any design changes after phase1

Project Testing

- You will be given different test programs. You are required to compile and load it onto the RAM and **reset** your processor to start executing from the right memory location. Each program would test some instructions (you should notify the TA if you haven't implemented or have logical errors concerning some of the instruction set).
- **You MUST prepare a waveform using do files with the main signals showing that your processor is working correctly (R0-R7, PC,SP,Flags,CLK,Reset,Interrupt, IN.port,Out.port, Exception_out).**

Evaluation Criteria

1. Each project will be evaluated according to the number of instructions that are implemented, and Pipelining hazards handled in the design.
2. Failing to implement a working processor will nullify your project grade. <u>No credits will be given to individual modules or a non-working processor.</u>
3. Your processor should be **synthesizable**.
4. Unnecessary latching or very poor understanding of underlying hardware will be penalized.

Team Members

- **Each team shall consist of a maximum of four members**

General Advice

1. **Compile your design regularly** (after each modification) so that you can figure out new errors early. Accumulated errors are harder to track.
2. Use engineering sense to backtrace the error source.
3. As much as you can, **don't ignore warnings.**
4. Read the **transcript window messages** in Modelsim carefully.
5. **After each major step**, and if you have a **working processor**, **save the design** before you modify it (use a versioning tool if you can as git & svn).
6. Always save the RAM files to easily export and import them.
7. **Start early** and give yourself enough time for testing.
8. Integrate your components incrementally (i.e: Integrate the RAM with the Registers, then integrate with them the ALU …).
9. Use coding conventions to know the functionality of each signal easily.
10. Try to simulate your control signals sequence for an instruction (i.e: Add) to know if your timing design is correct.
11. There is no problem in changing the design after phase 1, but justify your changes.
12. Always reset all components at the start of the simulation.
13. Don't leave any input signal float "U", set it with 0 or 1.
14. Remember that your VHDL code is a HW system (logic gates, Flipflops and wires).
15. Use Do files instead of re-forcing all inputs each time.
16. Give the Design phase a suitable amount of time and thinking. This will save you great time during implementation.
17. Document your design well. This will prevent any confusion during implementation.