Architecture Project

Objective

To design and implement a **simple 6-stage pipelined processor, Harvard**.
The design should conform to the ISA specification described in the following sections.

Introduction

The processor in this project has a RISC-like instruction set architecture. There are eight 2-byte general purpose registers; R_0, till R7. Another two general purpose registers, One works as a program counter (PC). And the other, works as a stack pointer (SP); and hence; points to the top of the stack. The initial value of SP is (2^10-2). The data memory address space is 1 KB of 16-bit width and is word addressable. (N.B. word = 2 bytes). You are allowed to make the data bus 32 bits to access two consecutive words.

The data memory in our processor is slow, so we split the memory stage into two stages and the new pipeline should be Fetch, Decode, Execute, Memory1,Memory2, writeback. Because we have only one Data Memory, no two consecutive memory instructions could be performed (structural hazard). The first stage of the memory should read the address and the data, the result won't be available until the second stage of the memory finishes.

When an interrupt occurs, the processor finishes the currently fetched instructions (instructions that have already entered the pipeline), then the address of the next instruction (in PC) is saved on top of the stack as well as the flags, and PC is loaded from address 1 of the instruction memory (Address 1 of the instruction memory contains the **address** of the interrupt service routine). To return from an interrupt, an RTI instruction loads the PC from the top of stack as well as the flags, and the flow of the program resumes from the instruction after the interrupted instruction. **Take care of corner cases like Branching.**

ISA Specifications

A) Registers

R[0:7]<15:0> ; Eight 16-bit general purpose register
PC<15:0> ; 16-bit program counter
SP<15:0>; 16-bit stack pointer
CCR<2:0> ; condition code register
Z<0>:=CCR<0> ; zero flag, change after arithmetic, logical, or shift operations
N<0>:=CCR<1> ; negative flag, change after arithmetic, logical, or shift operations
C<0>:=CCR<2> ; carry flag, change after arithmetic or shift operations.

B) Input-Output

IN.PORT<15:0> ; 16-bit data input port
OUT.PORT<15:0> ; 16-bit data output port
INTR.IN<0> ; a single, non-maskable interrupt
RESET.IN<0> ; reset signal

C) Mnemonics convention

Rsrc1 ; 1st operand register
Rsrc2 ; 2nd operand register
Rdst ; result register field
Imm ; Immediate Value 16-bits unless stated otherwise.

Take Care that Some instructions will Occupy more than one memory location

Mnemonic	Function
NOP	PC ← PC + 1
SETC	C ←1, make sure other flags don't get affected
CLRC	C ←0, make sure other flags don't get affected

NOT Rdst, Rsrc1	NOT value stored in register Rsrc and store it in Rdst
	R[Rdst] ← 1's Complement(R[Rsrc1]);
	If (1's Complement(R[Rsrc1]) = 0): Z ←1; else: Z ←0;
	If (1's Complement(R[Rsrc1]) < 0): N ←1; else: N ←0
	Don't change carry flag

INC Rdst, Rsrc1	Increment value stored in Rsrc1
	R[Rdst] ←R[Rsrc1] + 1;
	If ((R[Rsrc1] + 1) = 0): Z ←1; else: Z ←0;
	If ((R[Rsrc1] + 1) < 0): N ←1; else: N ←0
	Update carry flag as appropriate

DEC Rdst,Rsrc1	Decrement value stored in Rsrc1
	R[Rdst] ←R[Rsrc1] - 1;
	If ((R[Rsrc1] - 1) = 0): Z ←1; else: Z ←0;
	If ((R[Rsrc1] - 1) < 0): N ←1; else: N ←0
	Update carry flag as appropriate

OUT Rsrc1	OUT.PORT ← R[Rsrc1]
IN Rdst	R[Rdst] ←IN.PORT
MOV Rdst, Rsrc1	Move value from register Rsrc1 to register Rdst **DON'T change flags**
ADD Rdst, Rsrc1,Rsrc2	Add the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst and updates carry If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0 **Update carry flag as appropriate**
IADD Rdst, Rsrc1, Imm	Add the values stored in registers Rsrc1 to Immediate Value and store the result in Rdst and updates carry If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0 **Update carry flag as appropriate**
SUB Rdst, Rsrc1,Rsrc2	Subtract the values stored in registers **Rsrc1-Rsrc2** and store the result in Rdst and updates carry If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0 **Update carry flag as appropriate**
AND Rdst, Rsrc1,Rsrc2	AND the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0 **Don't change carry flag**
OR Rdst, Rsrc1,Rsrc2	OR the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst

	If the result =0 then Z ←1; else: Z ←0; If the result <0 then N ←1; else: N ←0 **Don't change carry flag**
PUSH Rsrc1	DataMemory[SP] ← R[Rsrc]; SP-=1
POP Rdst	SP+=1; R[Rdst] ← DataMemory[SP];
LDM Rdst, Imm	Load immediate value (16 bit) to register Rdst R[Rdst] ← Imm<15:0>
LDD Rdst, Rsrc1	Load value from memory address Rsrc1 R[Rdst] ← DataMemory [R[Rsrc1]];
STD Rsrc2, Rsrc1	Store value that is in register Rsrc1 to memory location Rsrc2 + offset DataMemory [R[Rsrc2]] ←R[Rsrc1];
JZ Rdst	Jump if zero If (Z=1): PC ←R[Rdst]; (Z=0)
JC Rdst	Jump if negative If (C=1): PC ←R[Rdst]; (C=0)
JMP Rdst	Jump PC ←R[Rdst]
CALL Rdst	(DataMemory[SP] ← PC + 1; sp-=1; PC ← R[Rdst])
RET	sp+=1, PC ← DataMemory [SP]
RTI	sp+=1; PC ← DataMemory [SP]; **Then restore Flags**

4 + 2

Stall

Input Signals	
Reset	PC ← M[0] //memory location of zero
Interrupt	DataMemory[Sp]←PC; sp-=1;PC ← M[1]; **Store Flags ins stack**

Phase1 Requirement: Design and Implement (50%)

- **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. → edit code to match 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) → not precisely correct needs modifications
    - PC and SP update circuits
  - Pipeline stages design
    - Pipeline registers details (Size, Input, Connection, …) → didn't start in it
    - Pipeline hazards and your solution including → Control hazards solutions
      i. Data Forwarding
      ii. Static Branch Prediction
  - **Code** : **(30%)**
    - Implement and integrate your architecture without any hazards for the given instructions only
      i. VHDL Implementation of each component of the processor
      ii. VHDL file that integrates the different components in a single module
      iii. Instructions:
        1. NOP
        2. INC
        3. AND
        4. IN
        5. LDD
        6. STD
    - 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: Clk,Rst,PC, Registers Values, Inport, MemoryOutput, Flags → didn't perform a final one on the new test cases
      - Load Memory File & Run the given test program

Phase2 Requirement: Assembler (5%)

- Assembler code that converts assembly program (Text File) into machine code according to your design (Memory File)
- **The assembler can shuffle or rename the Rsrc/Rdst register but It can't split one instruction into multiple instructions.**

Phase3 Requirement: Full processor (45%)

- 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 Instruction and Data Memories and **reset** your processor to start executing from the memory location written in address 0000h (PC = M[0] not 0). 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).

not implemented in code

same point in previous page didn't finalize a wave form

Evaluation Criteria

- Each project will be evaluated according to the number of instructions that are implemented, and Pipelining hazards handled in the design. Check Table 2 for the distribution of grades.
- Failing to implement a working processor will nullify your project grade. No credits will be given to individual modules or a non-working processor.
- Unnecessary latching or very poor understanding of underlying hardware will be penalized.
- Your processor should be **synthesizable**, else it will be strongly penalized.
- Individual Members of the same team can have different grades, you can get a zero grade if you didn't work while the rest of the team can get fullmark, Make sure you balance your Work distribution.

Table 2: Evaluation Criteria

Marks Distribution	Design	20 %
	Instructions without hazard	50 %
	Handling Hazards (Data + Control + Structural Hazard)	30 %

Team Members

- Each team shall consist of a maximum of four members

Phase 1 Due Date

- Sunday of Week 10 (16th of April), Delivery of hardcopy report to TAs or Dr and Code should be uploaded to elearning.
- Week 10, Discussion will be on the Lab slots.

Phase 2 Due Date

- **Sunday of** Week 12 (30th of April), upload to elearn with a readme explaining how it works

Project Due Date

- **Sunday of** Week 14(14th of May), Delivery a softcopy on elearn and discussion on the same day.

General Advice

1. Compile your design on regular basis (after each modification) so that you can figure out new errors early. Accumulated errors are harder to track.
2. Synthesize your processor on regular basis as well, accumulation of non-synthesizable code could me it hard to fix.

3. Start by finishing a working processor that does all one operands only. Integrating early will help you find a lot of errors. You can then add each type of instructions and integrate them into the working processor.
4. Use the engineering sense to back trace the error source.
5. As much as you can, don't ignore warnings.
6. Read the transcript window messages in Modelsim/Quartus carefully.
7. 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).
8. Always save the ram files to easily export and import them.
9. Start early and give yourself enough time for testing.
10. Integrate your components incrementally (i.e: Integrate the RAM with the Registers, then integrate with them the ALU …).
11. Use coding conventions to know each signal functionality easily.
12. Try to simulate your control signals sequence for an instruction (i.e: Add) to know if your timing design is correct.
13. There is no problem in changing the design after phase1, but justify your changes.
14. Always reset all components at the start of the simulation.
15. Don't leave any input signal float "U", set it with 0 or 1.
16. Remember that your VHDL code is a HW system (logic gates, Flipflops and wires).
17. Use Do files instead of re-forcing all inputs each time.