

Five-stages pipeline processor

Objective

By the end of this project, you will be able to design, implement and test a **Harvard (separate memories for data and instructions), RISC-like, five-stages pipeline** processor, with the specifications as described in the following sections.

Memory units and registers description

In this project, we apply a Harvard architecture with two memory units; **Instructions' memory** and **Data memory**.

The processor in this project has a RISC-like instruction set architecture. There are **eight 2-bytes general purpose registers [R0 to R7]**. These registers are **separate** from the **program counter** and the **stack pointer** registers.

The program counter (PC) spans the **instructions memory address** space that has a total size of **2 Megabytes**. Each memory address has a **16-bit width** (i.e., is word addressable). The instructions memory starts with the **interrupts area** (the very first address space from [0 down to 2^5-1]), followed by the **instructions area** (starting from [2^5 and down to 2^{20}]) as shown in Figure.1. By default, the PC is initialized with a value of (2^5) where the program code starts.

The other memory unit is the data memory, which has a total size of **4 Kilobytes** for its own, 16-bit in width (i.e., is word addressable). The processor can access both memory units at the same time without having a memory access hazard.

The data memory starts with the **data area** (the very first address space and down), followed by the **stack area** (starting from [$2^{11} - 1$ and up]) as shown in Figure.1. By default, the stack pointer (SP) pointer points to the top of the stack (the next free address available in the stack), and is initialized by a value of ($2^{11}-1$).

When an **interrupt occurs**, the processor **finishes the currently fetched instructions** (instructions that have already entered the pipeline), **save the processor state** (Flags), then the **address of the next instruction** (in PC) is saved on top of the stack, and **PC is loaded from address 0 of the memory where the interrupt code resides**.

For simplicity reasons, we will have only one **interrupt program**, the one which starts at the top of the instruction's memory, but be aware of possible **nested interrupts** i.e., an interrupt might be raised while executing an interrupt, and your processor should handle all of them successfully.

To **return from an interrupt**, an **RTI** instruction **loads the PC** from the top of stack, **restores the processor state** (Flags), and the **flow of the program resumes** from the instruction that was supposed to be fetched in-order before handling the interrupted instruction. **Take care of corner cases like Branching.**

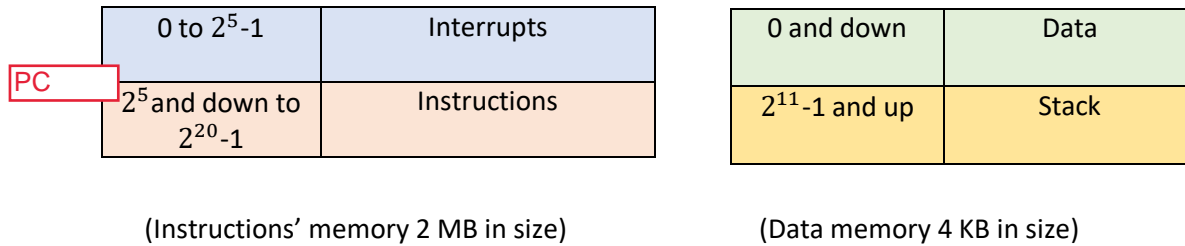


Figure.1 Memory Units

ISA specifications

A) Registers

$R[0:7]<15:0>$: Eight 16-bit general purpose registers

$PC<31:0>$: 32-bit program counter

$SP<31:0>$: 32-bit stack pointer

$CCR<3:0>$: condition code register that can be divided to

$Z<0>:=CCR<0>$: 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) Other registers to hold the operands and opcodes of the instructions




$Rsrc$: 1st operand register

$Rdst$: 2nd operand register and result register field

Imm : Immediate Value

D) Instructions (some instructions will occupy **more than one memory location**)

Table 1: ISA

Mnemonic	Function
One Operand	
 NOP	$PC \leftarrow PC + 1$
SETC	$C \leftarrow 1$
CLRC	$C \leftarrow 0$
 NOT Rdst	NOT value stored in register Rdst $R[Rdst] \leftarrow 1's \text{ Complement}(R[Rdst]);$ If $(1's \text{ Complement}(R[Rdst]) = 0)$: $Z \leftarrow 1$; else: $Z \leftarrow 0$; If $(1's \text{ Complement}(R[Rdst]) < 0)$: $N \leftarrow 1$; else: $N \leftarrow 0$
INC Rdst	Increment value stored in Rdst $R[Rdst] \leftarrow R[Rdst] + 1$; If $((R[Rdst] + 1) = 0)$: $Z \leftarrow 1$; else: $Z \leftarrow 0$; If $((R[Rdst] + 1) < 0)$: $N \leftarrow 1$; else: $N \leftarrow 0$
DEC Rdst	Decrement value stored in Rdst $R[Rdst] \leftarrow R[Rdst] - 1$; If $((R[Rdst] - 1) = 0)$: $Z \leftarrow 1$; else: $Z \leftarrow 0$; If $((R[Rdst] - 1) < 0)$: $N \leftarrow 1$; else: $N \leftarrow 0$
OUT Rdst	$OUT.PORT \leftarrow R[Rdst]$
IN Rdst	$R[Rdst] \leftarrow IN.PORT$
Two Operands	
MOV Rsrc, Rdst	Move value from register Rsrc to register Rdst
 ADD Rsrc, Rdst	Add the values stored in registers Rsrc, Rdst and store the result in Rdst If the result $= 0$ then $Z \leftarrow 1$; else: $Z \leftarrow 0$; If the result < 0 then $N \leftarrow 1$; else: $N \leftarrow 0$
SUB Rsrc, Rdst	Subtract the values stored in registers Rsrc, Rdst and store the result in Rdst If the result $= 0$ then $Z \leftarrow 1$; else: $Z \leftarrow 0$; If the result < 0 then $N \leftarrow 1$; else: $N \leftarrow 0$
AND Rsrc, Rdst	AND the values stored in registers Rsrc, Rdst and store the result in Rdst If the result $= 0$ then $Z \leftarrow 1$; else: $Z \leftarrow 0$; If the result < 0 then $N \leftarrow 1$; else: $N \leftarrow 0$
OR Rsrc, Rdst	OR the values stored in registers Rsrc, Rdst and store the result in Rdst If the result $= 0$ then $Z \leftarrow 1$; else: $Z \leftarrow 0$; If the result < 0 then $N \leftarrow 1$; else: $N \leftarrow 0$
SHL Rsrc, Imm	Shift left Rsrc by #Imm bits and store result in same register Don't forget to update carry
SHR Rsrc, Imm	Shift right Rsrc by #Imm bits and store result in same register Don't forget to update carry

Memory Operations	
PUSH Rdst	$X[SP--] \leftarrow R[Rdst];$
POP Rdst	$R[Rdst] \leftarrow X[++SP];$
LDM Rdst, Imm	Load immediate value (15 bit) to register Rdst $R[Rdst] \leftarrow Imm<15:0>$
LDD Rsrc, Rdst	Load value from memory address Rdst to register Rdst $R[Rdst] \leftarrow M[Rsrc];$
STD Rsrc, Rdst	Store value in register Rsrc to memory location Rdst $M[Rdst] \leftarrow R[Rsrc];$
Branch and Change of Control Operations	
JZ Rdst	Jump if zero If $(Z=1)$: $PC \leftarrow R[Rdst]$; $(Z=0)$
JN Rdst	Jump if negative If $(N=1)$: $PC \leftarrow R[Rdst]$; $(N=0)$
JC Rdst	Jump if negative If $(C=1)$: $PC \leftarrow R[Rdst]$; $(C=0)$
JMP Rdst	Jump $PC \leftarrow R[Rdst]$
CALL Rdst	$(X[SP] \leftarrow PC + 1; sp-2; PC \leftarrow R[Rdst])$
RET	$sp+2, PC \leftarrow X[SP]$
RTI	$sp+2; PC \leftarrow X[SP];$ Flags restored
Input Signals	
Reset	$PC \leftarrow 2^5h$ //memory location of the first instruction
Interrupt	$X[Sp] \leftarrow PC; sp-2; PC \leftarrow 0;$ Flags preserved

[25%] Phase1: Making a simple sequence (pipeline-like) processor

[Sunday of week 9]

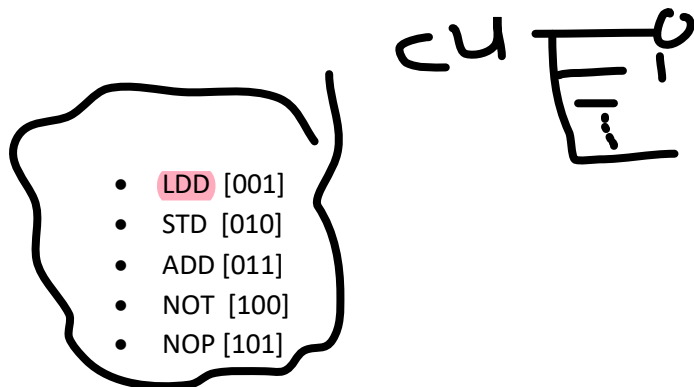
This phase aims to introduce you to the basic components that build the 5-stages pipeline, namely (Fetch -Decode-Execute-Memory-Write Back). So, as a starting point, we will simulate a simple pipeline that issues new instruction after 5-time steps. In every time step, the instruction passes through one of the stages. Each instruction should walk through the complete 5 stages even if it doesn't need to do anything in some of the stages. A new instruction enters only after the previous instruction is completed.

This phase implements no hazards, no branches and no concurrent execution of instructions.

This phase has to implement only the following five instructions using their description in Table.1 above:

1. LDD Rsrc, Rdst
2. STD Rsrc, Rdst
3. ADD Rsrc, Rdst
4. NOT Rdst
5. NOP

Since this is a temporary phase, we will use temporary opcodes of these instructions that can be changed later in phases 2 and 3. The opcodes are as follows:



You are asked to write your own testbench that tests the instruction memory that includes the below instructions **in order**, and get (and show that) your registers, memory, and condition code register are affected accordingly. (use any reasonable values in the memory).

1. LDD R1,0h
2. LDD R2,2h
3. NOP
4. ADD R2,R1
5. NOT R1
6. STD R2,R1

[25%] Phase2: Designing a complete 5-stages concurrent pipeline processor [Sunday of week 9]

In this phase, you will deliver a **design report** for your concurrent pipelined 5-stages processor.

The processor issues one instruction each cycle, and should be able to handle possible **hazards**, data forwarding, stalling, interrupts and branches with **static branch prediction** for prediction branches.

You should design and deliver a paper report (not software) containing the following:

- Instructions' format of your design
 - Opcode of each instruction.
 - Instruction bits details.
- **Schematic diagram** of the processor with data flow details
 - ALU / Registers / Memory Blocks.
 - Dataflow Interconnections between Blocks & its sizes.
 - Control Unit detailed design.
- Pipeline stages design
 - Pipeline registers details (Size, Input, Connection, ...).
 - Types of hazards in your design and your solution to them.
 - Data Forwarding.
 - Static Branch Prediction.

[50%] Phase3: Final delivery of working 5-stages concurrent pipeline processor [Sunday of week 13]

In this phase, you should:

- Implement and integrate your processor containing:
 - Verilog Implementation of each component of the processor.
 - A Verilog file that integrates the different components in a single module.

- A simulation test code that reads a program file and execute it on the processor that.
 - Setups the simulation wave
 - Loads Memory Files & Run the test program
- An assembler code that converts assembly program (Text File) into machine code according to your design (Memory File).
- A report that contains any design changes after phase2
- A report that contains pipeline hazards considered and how your design solves it.

Common design mistakes in phase 3:

Here are some of the design and implementation mistakes that you should avoid in your projects:

- Intermediate buffers and registers should not operate on the same clock edge, otherwise data will be late one cycle. Same for memory and intermediate buffers.
- Stalling and flushing are not the same thing, make sure to understand the difference.

Project testing

- You will be given different test programs. You are required to compile and load it onto the memories and **reset** your processor to start executing from memory location 2⁵h. 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).

Phase 3 evaluation criteria

In phase3, each project will be evaluated according to:

- The number of instructions that are functioning correctly.
- Pipelining hazards handled in the design.
- Efficient handling of the hazards.
- The branch prediction.
- 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.
- 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 full mark, make sure you balance your work distribution.

Team members

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

General advices

1. Compile your design on regular bases (after each modification) so that you can figure out new errors early. Accumulated errors are harder to track.
2. Use the engineering sense to back trace 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 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 memory with the registers, then integrate with them the ALU ...).
9. Use coding convention to know each signal functionality 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 phase1, 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 Verilog code is a HW system (logic gates, Flipflops and wires).
15. Use Do files instead of re-forcing all inputs each time

Stay tuned and develop amazing processors :D