Cairo University | CMP N 301
Faculty of Engineering | Spring 2017
Computer Engineering Department

# Architecture Project

## Objective

To design and implement a simple 5-stage pipelined processor, von Neumann or 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; $R_0$, $R_1$, $R_2$, $R_3$, $R_4$, $R_5$, $R_6$ and $R_7$. $R_7$ works as program counter (PC). $R_6$ works as a stack pointer (SP); and hence; points to the top of the stack. The initial value of SP is 1023. The memory (both in case of Harvard) address space is 1 KB of 16-bit width and is word addressable.

When an interrupt occurs, the processor finishes the currently executing instruction, then the address of the next instruction (in PC) is saved on top of the stack, and PC is loaded from address 1 of the memory. To return from an interrupt, an RTI instruction loads PC from the top of stack, and the flow of the program resumes from the instruction after the interrupted instruction.

## ISA Specifications

**A) Registers**

R[0:7]<15:0> ; Eight 16-bit general purpose registers
PC<15:0>:=R[7]<15:0> ; 16-bit program counter
SP<15:0>:=R[6]<15:0> ; 16-bit stack pointer
CCR<3:0> ; 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.
V<0>:=CCR<3> ; over flow, 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

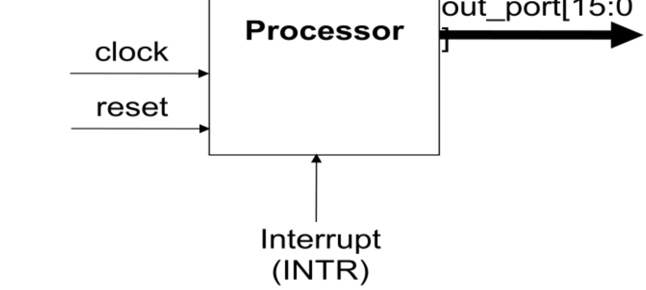

Table 1 shows a summary of the Instruction Set Architecture (ISA) that must be implemented by your processor.

Table 1: Processor ISA (X is the stack and M is the Memory)

Rsrc1 ; 1st operand register<br>
Rsrc2 ; 2nd operand register<br>
Rdst ; Result register field<br>
EA ; Effective address<br>
Imm ; Immediate Value (16 bit)

| **Mnemonic** | **Function** |
|---|---|
| NOP | PC ← PC + 1 |
| MOV Rsrc1, Rdst | Move value from register Rsrc1 to register Rdst |
| ADD Rsrc1, Rsrc2, Rdst | Add the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst<br>If the result =0 then Z ←1; else: Z ←0;<br>If the result <0 then N ←1; else: N ←0 |
| SUB Rsrc1, Rsrc2, Rdst | Subtract the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst<br>If the result =0 then Z ←1; else: Z ←0;<br>If the result <0 then N ←1; else: N ←0 |
| AND Rsrc1, Rsrc2, Rdst | AND the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst<br>If the result =0 then Z ←1; else: Z ←0;<br>If the result <0 then N ←1; else: N ←0 |
| OR Rsrc1, Rsrc2, Rdst | OR the values stored in registers Rsrc1, Rsrc2 and store the result in Rdst<br>If the result =0 then Z ←1; else: Z ←0;<br>If the result <0 then N ←1; else: N ←0 |
| RLC Rdst | Rotate left Rdst ; C ←R[ Rdst ]<15>;<br>R[ Rdst ] ← R[ Rdst ]<14:0>&C |
| RRC Rdst | Rotate right Rdst ; C ←R[ Rdst ]<0>;<br>R[ Rdst ] ←C&R[ Rdst ]<15:1>; |
| SHL Rdst, Imm | Shift left Rdst by #Imm bits<br>R[ Rdst ] ← R[ Rdst ]<(15-Imm):0>&0 |
| SHR Rdst, Imm | Shift right Rdst by #Imm bits<br>R[ Rdst ] ← 0&R[ Rdst ]<15:Imm> |
| SETC | C ←1 |
| CLRC | C ←0 |
| PUSH Rdst | X[SP--] ← R[ Rdst ]; |
| POP Rdst | R[ Rdst ] ← X[++SP]; |
| OUT Rdst | OUT.PORT ← R[ Rdst ] |
| IN Rdst | R[ Rdst ] ←IN.PORT |
| NOT Rdst | NOT value stored in register Rdst<br>R[ Rdst ] ← 1's Complement(R[ Rdst ]);<br>If (1's Complement(R[ Rdst ]) = 0): Z ←1; else: Z ←0;<br>If (1's Complement(R[ Rdst ]) < 0): N ←1; else: N ←0 |
| NEG Rdst | Negate the value stored in register Rdst<br>R[ Rdst ] ← 2's Complement(R[ Rdst ]);<br>If (2's Complement(R[ Rdst ]) = 0): Z ←1; else: Z ←0;<br>If (2's Complement(R[ Rdst ]) < 0): N ←1; else: N ←0 |
| INC Rdst | Increment value stored in Rdst<br>R[ Rdst ] ←R[ Rdst ] + 1;<br>If ((R[ Rdst ] + 1) = 0): Z ←1; else: Z ←0;<br>If ((R[ Rdst ] + 1) < 0): N ←1; else: N ←0 |

| | |
|---|---|
| DEC Rdst | Decrement value stored in Rdst<br>R[ Rdst ] ←R[ Rdst ] – 1;<br>If ((R[ Rdst ] – 1) = 0): Z ←1; else: Z ←0;<br>If ((R[ Rdst ] – 1) < 0): N ←1; else: N ←0 |
| JZ Rdst | Jump if zero<br>If (Z=1): PC ←R[ Rdst ]; (Z=0) |
| JN Rdst | Jump if negative<br>If (N=1): PC ←R[ Rdst ]; (N=0) |
| JC Rdst | Jump if carry<br>If (C=1): PC ←R[ Rdst ]; (C=0): |
| JMP Rdst | Jump<br>PC ←R[ Rdst ] |
| CALL Rdst | (X[SP--] ← PC + 1; PC ← R[ Rdst ]) |
| RET | PC ← X[++SP] |
| RTI | PC ← X[++SP]; Flags restored |
| LDM Rdst, Imm | Load immediate value to register Rdst<br>R[ Rdst ] ← Imm<15:0> |
| LDD Rdst, EA | Load value from memory address EA to register Rdst<br>R[ Rdst ] ← M[EA]; |
| STD Rsrc, EA | Store value in register Rsrc to memory location EA<br>M[EA] ←R[Rsrc]; |

| **Input Signals** | |
|---|---|
| Reset | PC ← M[0] |
| Interrupt | X[SP--]←PC; PC ← M[1]; Flags preserved |

## Phase1 Requirement

- Instruction 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, …)
  - Pipeline hazards and your solution including
    1. Data Forwarding
    2. Static Branch Prediction

## Phase2 Requirement

- Implement and integrate your architecture
  - VHDL Implementation of each component of the processor
  - VHDL file that integrates the different components in a single module
- Simulation Test code that reads a program file and execute it on the processor.
  - Setup the simulation wave
  - Load Memory File & Run the test program
- Assembler code that converts assembly program (Text File) into machine code according to your design (Memory File)
- Report that contains any design changes after phase1
- Report that contains pipeline hazards considered and how your design solves it.

## 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 memory location 0000h. 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 with the main signals showing that your processor is working correctly (R0, R1, R2, R3, PC, …etc).

## Evaluation Criteria

- Each project will be evaluated according to the number of instructions that are implemented, and Pipelining hazards handled in the design. Table 2 shows the evaluation criteria in details.
- Failing to implement a working processor will nullify your project grade. No credits will be given to individual modules or a non-working processor.
- Synthesizing your processor and correctly carrying out timing simulation will be rewarded by bonus 1 mark that accumulates towards the 40-marks semester work.

Table 2: Evaluation Criteria

| | | |
|---|---|---|
| **Marks Distribution** | Each instruction (out of 30 instructions) | 0.5 mark (15 marks total) |
| | Data Forwarding | 2 marks |
| | Static Branch Prediction | 2 marks |
| | Interrupt | 1 mark |
| **Bonus Marks** | • Synthesizing the processor (Getting frequency and FPGA utilization) | 1 mark bonus to semester work grade |

## Team Members

- Each team shall consist of a maximum of four members

## Phase1 Due Date

- Week 9 (Thursday April 13rd, 2017). Section time.

## Project Due Date

- Week 13 (Thursday May 18th, 2017). The demo will be during the regular lab session.

## General Advice

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 RAM 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 VHDL code is a HW system (logic gates, Flipflops and wires).
15. Use Do files instead of re-forcing all inputs each time.