Cairo University
Faculty of Engineering
Electronics and Electrical Communications Engineering Department

ELC 3030
Fall 2025

Advanced Processor Architecture - Project

Objectives:

- The goal of this project is for students to design and implement a simple 8-bit pipelined processor, von Neumann or Harvard, using VHDL or Verilog (student's choice).
- This project emphasizes resource sharing (especially the single memory resource) and the design of a Finite State Machine (FSM) Control Unit.
- 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. Instructions are 1-byte or 2-bytes depending on the type of instructions. There are four 1-byte general purpose registers; R_0, R_1, R_2, and R_3. R_3 also works as a stack pointer (SP); hence; points to the top of the stack. The initial value of SP is 255. The memory address space is 256 bytes and is byte addressable. When an interrupt occurs, the address of the instruction next to the interrupted instruction 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

```
PC<7:0>              ; 8-bit program counter
R[0:3]<7:0>          ; Four 8-bit general purpose registers
SP<7:0>:=R[3]<7:0>   ; 8-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>         ; overflow, change after arithmetic or shift operations.
```

B) Input-Output

```
IN.PORT<7:0>         ; 8-bit data input port
OUT.PORT<7:0>        ; 8-bit data output port
INTR.IN<0>           ; a single, non-maskable interrupt
RESET.IN<0>          ; reset signal
```

C) Instruction Format

```
i<7:0>:=M[PC]<7:0>              ; 8-bit instruction word
opcode<3:0>:=i<7:4>             ; 4-bit opcode
ra<1:0>:=i<3:2>                 ; 2-bit operand register and result register field
rb<1:0>:=i<1:0>                 ; 2-bit operand register field
brx<1:0>:=i<3:2>                ; 2-bit branch index field
ea<7:0>:=M[PC+1]<7:0>           ; effective address
imm<7:0>:=M[PC+1]<7:0>          ; immediate operand
```

Arithmetic operations are performed in two's complement. Shift and logical operations are bit-wise. There are 3 different instruction formats:

1- A-Format

Figure 1 depicts A-format instructions. These instructions are 1-byte. Op-code is the high order nibble and the low order nibble determines two registers.

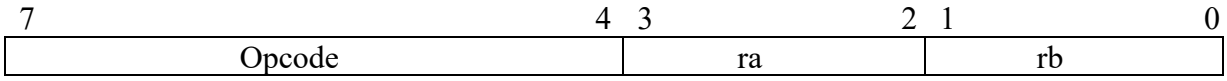

Figure 1: A-format Instructions

Table I shows op-code values for A-format instructions and explains their functionality. R[ra] and R[rb] indicate values of registers ra and rb, respectively. For example: *ADD r2, r1* instruction has op-code = 2, ra = 2, rb = 1, and bit stream of 00101001. Hence, the hexadecimal format of the instruction is: 0x29. The PUSH instruction writes the contents of register rb into the memory location addressed by SP. Then SP is decremented. In the POP instruction, SP is first incremented, and then, the contents of the memory location pointed to by SP are written into rb.

Table I: A-format Instructions (X is the stack)

Mnemonic	Opcode	Length (Bytes)	Function
NOP	0	1	PC ← PC + 1
MOV	1	1	R[ra] ← R[rb]; PC ← PC + 1
ADD	2	1	R[ra] ←R[ra] + R[rb]; PC ← PC + 1; Change C, V flags ((R[ra] + R[rb]) = 0): Z ←1; else: Z ←0; ((R[ra] + R[rb]) < 0): N ←1; else: N ←0
SUB	3	1	R[ra] ←R[ra] – R[rb]; PC ← PC + 1; Change C, V flags ((R[ra] – R[rb]) = 0): Z ←1; else: Z ←0; ((R[ra] – R[rb]) < 0): N ←1; else: N ←0
AND	4	1	R[ra] ←R[ra] AND R[rb]; PC ← PC + 1; ((R[ra] AND R[rb]) = 0): Z ←1; else: Z ←0; ((R[ra] AND R[rb]) < 0): N ←1; else: N ←0
OR	5	1	R[ra] ←R[ra] OR R[rb]; PC ← PC + 1; ((R[ra] OR R[rb]) = 0): Z ←1; else: Z ←0; ((R[ra] OR R[rb]) < 0): N ←1; else: N ←0
RLC	6	1	(ra = 0): C ←R[rb]<7>; R[rb] ← R[rb]<6:0>&C; PC ← PC + 1
RRC	6	1	(ra = 1): C ←R[rb]<0>; R[rb] ←C&R[rb]<7:1>; PC ← PC + 1
SETC	6	1	(ra = 2): C ←1; PC ← PC + 1
CLRC	6	1	(ra = 3): C ←0; PC ← PC + 1
PUSH	7	1	(ra = 0): X[SP--] ← R[rb]; PC ← PC + 1
POP	7	1	(ra = 1): R[rb] ← X[++SP]; PC ← PC + 1
OUT	7	1	(ra = 2): OUT.PORT ← R[rb]; PC ← PC + 1
IN	7	1	(ra = 3): R[rb] ←IN.PORT; PC ← PC + 1
NOT	8	1	(ra = 0): R[rb] ← 1's Complement(R[rb]); PC ← PC + 1; (1's Complement(R[rb]) = 0): Z ←1; else: Z ←0; (1's Complement(R[rb]) < 0): N ←1; else: N ←0
NEG	8	1	(ra = 1): R[rb] ← 2's Complement(R[rb]); PC ← PC + 1; (2's Complement(R[rb]) = 0): Z ←1; else: Z ←0; (2's Complement(R[rb]) < 0): N ←1; else: N ←0
INC	8	1	(ra = 2): R[rb] ←R[rb] + 1; PC ← PC + 1; Change C, V flags ((R[rb] + 1) = 0): Z ←1; else: Z ←0; ((R[rb] + 1) < 0): N ←1; else: N ←0
DEC	8	1	(ra = 3): R[rb] ←R[rb] – 1; PC ← PC + 1; Change C, V flags ((R[rb] – 1) = 0): Z ←1; else: Z ←0; ((R[rb] – 1) < 0): N ←1; else: N ←0

2- B-Format

B-format instructions are 1-byte and include instructions that break the sequential execution of programs. As figure 2 shows, b-format instructions have op-code, ra "also named brx", and rb. The brx field determines the type of the branch instruction.

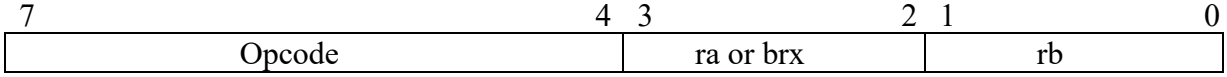

Figure 2: B-format Instructions

Table II shows the details of B-format instructions. JMP instruction jumps to the destination address determined by the rb field. JZ is a conditional branch. If Z flag is one, it jumps to the destination address determined by rb. Similarly, JN, JC, and JV jump to the destination address determined by rb if N, C, and V flags are one, respectively. CALL is used for subroutine call. The processor has a dedicated pin for external interrupt. On the rising edge of the interrupt pin, the address of the next instruction is written into the stack and the SP is decremented (X[SP--] ← PC). PC is loaded from address 1 (PC ← M[1]), and processor jumps into interrupt service routine. Flags are also saved. At the end of the interrupt service routine, RTI instruction executes. RTI is similar to b-format instructions. It increments SP and load PC from the top of the stack. Flags are restored.

Table II: B-format Instructions (X is the stack)

Mnemonic	Opcode	Length (Bytes)	Function
JZ	9	1	(brx=0 ∩ Z=1): PC ←R[rb]; (brx=0 ∩ Z=0): PC ← PC + 1
JN	9	1	(brx=1 ∩ N=1): PC ←R[rb]; (brx=1 ∩ N=0): PC ← PC + 1
JC	9	1	(brx=2 ∩ C=1): PC ←R[rb]; (brx=2 ∩ Z=0): PC ← PC + 1
JV	9	1	(brx=3 ∩ V=1): PC ←R[rb]; (brx=3 ∩ V=0): PC ← PC + 1
LOOP	10	1	R[ra] ←R[ra] – 1; ((R[ra] – 1) != 0): PC ←R[rb]; ((R[ra] – 1) = 0): PC ← PC + 1
JMP	11	1	(brx=0): PC ←R[rb]
CALL	11	1	(brx=1): (X[SP--] ← PC + 1; PC ← R[rb])
RET	11	1	(brx=2): PC ← X[++SP]
RTI	11	1	(brx=3): PC ← X[++SP]; Flags restored

3- L-Format

L-format instructions are either one or two bytes and are used for load/store instructions. For two bytes instructions, the second byte holds the address of memory or an immediate value. Figure 3 shows L-format instructions.

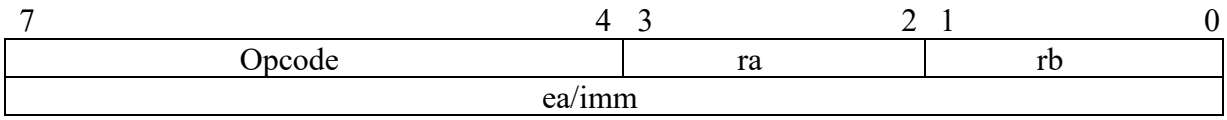

Figure 3: L-format Instructions

Table III shows the details of L-format instructions. LDM writes a constant value (imm) into register rb. LDD and STD instructions use direct addressing. They read/write the contents of register rb from/into address ea. M[ea] means the content of a memory location with address ea. LDI and STI use indirect addressing. They read/write the contents of register rb

from/into the memory location pointed to by ra. M[R[ra]] means the content of a memory location with address R[ra].

Table III: L-format Instructions (M is the Memory)

Mnemonic	Opcode	Length (Bytes)	Function
LDM	12	2	(ra = 0): R[rb] ← imm; PC ← PC + 2
LDD	12	2	(ra = 1): R[rb] ← M[ea]; PC ← PC + 2
STD	12	2	(ra = 2): M[ea] ←R[rb]; PC ← PC + 2
LDI	13	1	R[rb] ← M[R[ra]]; PC ← PC + 1
STI	14	1	M[R[ra]] ←R[rb]; PC ← PC + 1

Table V shows a summary of the Instruction Set Architecture (ISA).

Implementation Requirement:

You can use whatever tool for simulation you want. If you don't have any tool available, you can use **EDA Playground** for simulation and verification. This is an online tool you can use, it doesn't require you to download any software on your laptop.

Website Link: https://www.edaplayground.com/

Key Task: Ensure you enable **VCD dump** in the run options and analyze the waveforms. Your submission should include the final HDL file and a screenshot of the waveform.

General Advice:

- Compile your design on a regular basis (after each modification) so that you can figure out new errors early. Accumulated errors are harder to track.
- Use the engineering sense to back trace the error source.
- As much as you can, don't ignore warnings.
- After each major step, and if you have a working processor, save the design before you modify it.
- Always save the ram files to easily export and import them.
- Start early and give yourself enough time for testing

Submission Details:

- This is a team project, and each team shall consist of a maximum **of six members**.
- The deadline for submission is **20-12-2025 11:59 PM**.
- Each team should submit a single .zip folder on Google classroom containing the following:
    - A PDF file including your system design (Architecture, FSM, .. etc), as well as the HDL codes of the processor and the used test benches. Screenshots of the waveforms of the outputs used for verification shall be included too.
    - A video including all team members presenting a demo of their working processor using several test cases.
    - A folder containing the HDL files of the processor for the TA's evaluation.
- Don't forget to mention the team members on the PDF's cover page!
- Following the rules, submitting before the deadline and neat reports are part of the final grade.

Evaluation Criteria:

Each team will be evaluated according to the number of instructions that are implemented. Table IV shows the evaluation criteria in detail. 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 using Von Neumann architecture will be rewarded with 10 bonus marks.

Table IV: Evaluation Criteria

Marks Distribution	Reset	5 marks
	Each instruction (out of 32 instructions)	2.5 marks (80 marks total)
	Data Forwarding	10 marks
	Interrupt	5 marks
Bonus Marks	• Synthesizing the processor (Getting frequency and FPGA utilization) • Using Von Neumann memory architecture	10 marks bonus (5 marks each)

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

Mnemonic	Opcode	Length	Function
NOP	0	1	PC ← PC + 1
MOV	1	1	R[ra] ← R[rb]; PC ← PC + 1
ADD	2	1	R[ra] ←R[ra] + R[rb]; PC ← PC + 1; Change C, V flags ((R[ra] + R[rb]) = 0): Z ←1; else: Z ←0; ((R[ra] + R[rb]) < 0): N ←1; else: N ←0
SUB	3	1	R[ra] ←R[ra] – R[rb]; PC ← PC + 1; Change C, V flags ((R[ra] – R[rb]) = 0): Z ←1; else: Z ←0; ((R[ra] – R[rb]) < 0): N ←1; else: N ←0
AND	4	1	R[ra] ←R[ra] AND R[rb]; PC ← PC + 1; ((R[ra] AND R[rb]) = 0): Z ←1; else: Z ←0; ((R[ra] AND R[rb]) < 0): N ←1; else: N ←0
OR	5	1	R[ra] ←R[ra] OR R[rb]; PC ← PC + 1; ((R[ra] OR R[rb]) = 0): Z ←1; else: Z ←0; ((R[ra] OR R[rb]) < 0): N ←1; else: N ←0
RLC	6	1	(ra = 0): C ←R[rb]<7>; R[rb] ← R[rb]<6:0>&C; PC ← PC + 1
RRC	6	1	(ra = 1): C ←R[rb]<0>; R[rb] ←C&R[rb]<7:1>; PC ← PC + 1
SETC	6	1	(ra = 2): C ←1; PC ← PC + 1
CLRC	6	1	(ra = 3): C ←0; PC ← PC + 1
PUSH	7	1	(ra = 0): X[SP--] ← R[rb]; PC ← PC + 1
POP	7	1	(ra = 1): R[rb] ← X[++SP]; PC ← PC + 1
OUT	7	1	(ra = 2): OUT.PORT ← R[rb]; PC ← PC + 1
IN	7	1	(ra = 3): R[rb] ←IN.PORT; PC ← PC + 1
NOT	8	1	(ra = 0): R[rb] ← 1's Complement(R[rb]); PC ← PC + 1; (1's Complement(R[rb]) = 0): Z ←1; else: Z ←0; (1's Complement(R[rb]) < 0): N ←1; else: N ←0
NEG	8	1	(ra = 1): R[rb] ← 2's Complement(R[rb]); PC ← PC + 1; (2's Complement(R[rb]) = 0): Z ←1; else: Z ←0; (2's Complement(R[rb]) < 0): N ←1; else: N ←0
INC	8	1	(ra = 2): R[rb] ←R[rb] + 1; PC ← PC + 1; Change C, V flags ((R[rb] + 1) = 0): Z ←1; else: Z ←0; ((R[rb] + 1) < 0): N ←1; else: N ←0
DEC	8	1	(ra = 3): R[rb] ←R[rb] – 1; PC ← PC + 1; Change C, V flags ((R[rb] – 1) = 0): Z ←1; else: Z ←0; ((R[rb] – 1) < 0): N ←1; else: N ←0
JZ	9	1	(brx=0 ∩ Z=1): PC ←R[rb]; (brx=0 ∩ Z=0): PC ← PC + 1
JN	9	1	(brx=1 ∩ N=1): PC ←R[rb]; (brx=1 ∩ N=0): PC ← PC + 1

JC	9	1	(brx=2 ∩ C=1): PC ←R[rb]; (brx=2 ∩ Z=0): PC ← PC + 1
JV	9	1	(brx=3 ∩ V=1): PC ←R[rb]; (brx=3 ∩ V=0): PC ← PC + 1
LOOP	10	1	R[ra] ←R[ra] – 1; ((R[ra] – 1) != 0): PC ←R[rb]; ((R[ra] – 1) = 0): PC ← PC + 1
JMP	11	1	(brx=0): PC ←R[rb]
CALL	11	1	(brx=1): (X[SP--] ← PC + 1; PC ← R[rb])
RET	11	1	(brx=2): PC ← X[++SP]
RTI	11	1	(brx=3): PC ← X[++SP]; Flags restored
LDM	12	2	(ra = 0): R[rb] ← imm; PC ← PC + 2
LDD	12	2	(ra = 1): R[rb] ← M[ea]; PC ← PC + 2
STD	12	2	(ra = 2): M[ea] ←R[rb]; PC ← PC + 2
LDI	13	1	R[rb] ← M[R[ra]]; PC ← PC + 1
STI	14	1	M[R[ra]] ←R[rb]; PC ← PC + 1

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