Phase 1 Report
Design of a 5-stage Pipelined RISC-like Processor

Team Members:

Ammar Yasser Fadlallah ID: 9230593
Mohamed Yasser El-batesh ID: 9230825
Mohamed Hany Saqr ID: 9230818
Abdallah Ahmed Abd-Alfatah ID: 9230553

Course: Architecture Project Fall 2025

November 27, 2025

Contents

1 Project summary and design choices

This report documents Phase 1 of the pipelined processor project. It reflects the design decisions you provided:

- Instruction word / memory word width: **32 bits**.
- Instructions are categorized into **4 types**:

  R-type: `Type = 00`
  I-type: `Type = 01`
  J-type: `Type = 10`
  System / Push-Pop: `Type = 11`
- The **opcode field is 7 bits**. Within these 7 bits: **1 bit = HasImm**, **2 bits = Type**, **4 bits = sub-opcode (function)**. So opcode format (MSB→LSB): `HasImm : Type[1:0] : Func[3:0]`.
- If **HasImm = 1**, the instruction uses an immediate value located in the **successor memory word** (the next 32-bit memory location).
- System-type instructions (Type=11) include: **PUSH, POP, INT, RET, RTI, HLT**.
- The processor is pipelined with **5 stages**: IF (Fetch), ID (Decode/Register read), EX (Execute/ALU), MEM (Memory access), WB (Writeback).
- Data forwarding and static branch prediction rules are those you provided (detailed below).
- The user provided a full schematic — included as an image placeholder (page end).

2 Instruction-format conventions

2.1 Opcode bitfield convention

The 7-bit opcode is laid out as:

`[6] HasImm | [5:4] Type | [3:0] Func`

Where:

- `HasImm` = 1 means a 32-bit immediate (or target) is stored in the next memory word.
- `Type` selects the instruction format: 00=R, 01=I, 10=J, 11=System.
- `Func` is a 4-bit subtype/opcode within the selected type.

Rationale: embedding HasImm and Type inside the 7-bit opcode keeps decoding compact and explicit and matches your requirement that HasImm and Type are part of the opcode.

2.2 R-type instruction (single 32-bit word)

Notes: Many R-type instructions (ADD, SUB, AND, NOT, INC, MOV, SWAP, etc.) are encoded here. Examples for syntax:

- `NOT Rdst` (meaning: `not rdst, rdst`)
- `SWAP Rsrc, Rdst` encoded as `op, Rdst, Rsrc, Rdst` if needed by your assembler convention.

2.3 I-type instruction (two words: inst + immediate)

Word 2: 32-bit Immediate/Constant (signed or unsigned per instruction)

Example: `IADD Rdst, Rsrc, Imm` uses HasImm=1 and places Imm in the next memory location.

2.4 J-type instruction (two words: inst + target)

Opcode (7) Rest unused in this word

Word 2: 32-bit Jump target address (absolute)

Notes: JMP, JZ, JN, JC, CALL use the J-type format; HasImm bit = 1 (target in successor word).

2.5 System / Push-Pop type (Type = 11)

These instructions are single-word or may use a following word for additional data depending on the instruction. Included instructions: **PUSH, POP, INT, RET, RTI, HLT**. They have `Type=11` in opcode. PUSH/POP operate on the stack pointer SP as described in the ISA.

3 Example opcode table (designed for clarity)

Below is a consistent 7-bit opcode mapping following `HasImm | Type[1:0] | Func[3:0]`. The binary shown is `b6 b5 b4 b3 b2 b1 b0`:

2whitegray!10

Instruction	Type	Notes	Opcode (7-bit)
ADD	R (00)	Rdst = Rsrc1 + Rsrc2	`0 00 0001`
SUB	R (00)		`0 00 0010`
AND	R (00)		`0 00 0011`
MOV	R (00)	move reg-¿reg	`0 00 0100`
SWAP	R (00)	swap two regs (see note)	`0 00 0101`
NOT	R (00)	unary: NOT Rdst	`0 00 0110`
INC	R (00)	Rdst += 1	`0 00 0111`
IADD	I (01)	Rdst = Rsrc + Imm (Imm in next word)	`1 01 0001`
LDM	I (01)	Load immediate to register (Imm next word)	`1 01 0010`
LDD	I (01)	Load from mem: Rdst ← M[Rsrc + offset(next word)]	`1 01 0011`
STD	I (01)	Store: M[Rsrc2 + offset(next word)] ← Rsrc1	`1 01 0100`
JMP	J (10)	Jump target in next word	`1 10 0001`
JZ	J (10)	Jump if Z (target next word)	`1 10 0010`
JN	J (10)	Jump if N	`1 10 0011`
JC	J (10)	Jump if C	`1 10 0100`
CALL	J (10)	Call: push return PC and branch (target next word)	`1 10 0101`
RET	Sys (11)	Return (stack pop into PC)	`0 11 0001`
PUSH	Sys (11)	Push Rdst onto stack	`0 11 0010`
POP	Sys (11)	Pop into Rdst	`0 11 0011`
INT	Sys (11)	Software interrupt (index in func or next word)	`0 11 0100`
RTI	Sys (11)	Return from interrupt	`0 11 0101`
HLT	Sys (11)	Halt until reset	`0 11 0110`
OUT	Sys/R?	Output register to OUT.PORT	`0 00 1000`
IN	Sys/R?	Input port to register	`0 00 1001`
NOP	Sys/R?	No operation	`0 00 0000`

Notes:

- The table is intentionally extensible. The 4-bit func field allows 16 sub-opcodes per type.
- Opcodes marked *Sys/R?* appear as R-type function codes for compactness (examples: IN/OUT could also be represented in System type depending on your final choice).
- The assembler should interpret the `HasImm` bit and read an extra memory word when it is 1.
- The assembly syntax follows your notes: `NOT rdst` means `not rdst, rdst`. `SWAP` serialized as `op, rdst, rsrc, rdst` if necessary.

4 High-level datapath components

Blocks:

- **Instruction / Data Memory (Unified)**: 1 MB address space, 32-bit width per word. Memory location 0 holds `M[0]` used for PC reset vector; `M[1]` used for interrupt vector per project spec.
- **Register File:** 8 general-purpose registers R0..R7, each 32 bit; PC, SP, CCR (flags: Z,N,C).
- **ALU:** 32-bit arithmetic and logic operations (add, sub, and, not, shifts if needed).
- **Control Unit:** Main finite logic that decodes Opcode, produces control signals (RegWrite, MemRead, MemWrite, ALUSrc, ALUOp, Branch signals, etc.). An ALU control submodule produces ALU operation codes.
- **Forwarding Unit:** Compares destination register fields in pipeline registers and selects forwarded operands.
- **Hazard Unit:** Detects load-use hazards and stalling, flushes on control hazards where needed.

5 Pipeline stages and pipeline registers

5.1 Five stages

IF (Instruction Fetch): Read instruction word from memory at PC. If HasImm=1 for some instructions, the next word is also read by MEM stage or handled in ID/EX as required by your implementation.

IF/ID register: Holds fetched instruction (32-bit), PC+1 (or PC+2 if immediate word is pre-fetched), and minimal control signals required for decode.

ID (Decode/Register read): Decode opcode, read register file (Rsrc1/Rsrc2), sign-extend immediate if needed (or mark immediate fetch requirement), compute branch conditions (Z/N/C read). Evaluate conditional branches in ID (per your static-not-taken scheme).

ID/EX register: Holds register operands, control signals for EX stage, immediate word pointer / indicator if immediate exists.

EX (Execute): ALU operation, branch target calculation; apply forwarded operands if required.

EX/MEM register: Holds ALU result, destination register specifier, control signals for MEM stage.

MEM (Memory access): Perform loads/stores by address computed in EX.

MEM/WB register: Holds memory read data or ALU result to write back to register file.

WB (Writeback): Write results to register file (R0-R7), update CCR flags if appropriate.

5.2 Sizes of pipeline registers (representative)

- IF/ID: Instruction (32b) + PC (32b) + control bits ($\approx$ 8-12b) $\rightarrow$ **80 bits**.
- ID/EX: Read data 1 (32b), read data 2 (32b), immediate pointer/word (32b or flag), register specifiers (3+3+3 bits), control bits ($\approx$ 12b) $\rightarrow$ **120 bits**.

- EX/MEM: ALU result (32b), write data (32b), dest reg (3b), control bits ($\approx$ 8b) $\rightarrow$ **80 bits**.
- MEM/WB: Memory read data (32b), ALU result (32b), dest reg (3b), control bits ($\approx$ 8b) $\rightarrow$ **80 bits**.

6 Control unit design (detailed)

6.1 Signals generated by main control (examples)

- `RegWrite` — write register in WB stage.
- `MemRead`, `MemWrite` — memory operations in MEM stage.
- `ALUSrc` — choose between register operand or immediate.
- `ALUOp[3:0]` — ALU function select.
- `Branch` — branch detected; in our design, we perform branch evaluation in ID and flush IF/ID on taken branch (incurs 1-cycle penalty).
- `IsLoad` — used by Hazard Unit to detect load-use cases requiring stalls.
- `IsSys` — indicates system operation (PUSH/POP/INT/RET/RTI/HLT).

6.2 Control sequencing

The Main Control decodes the 7-bit opcode and produces the control bus for use across pipeline registers. An ALU Control submodule accepts ALUOp and function specifiers to produce the ALU operation.

7 Hazards and solutions

7.1 Data hazards (RAW)

We implement the following forwarding paths and rules (as you provided):

1. If `EX/MEM.RegWrite = 1` and `EX/MEM.DestReg` $\neq$ `0` and `EX/MEM.DestReg == ID/EX.rs or ID/EX.rt` then forward from `EX/MEM` into EX stage inputs.
2. Else if `MEM/WB.RegWrite = 1` and `MEM/WB.DestReg` $\neq$ `0` and `MEM/WB.DestReg == ID/EX.rs or ID/EX.rt` then forward from `MEM/WB`.

Load-use hazard: If the EX/MEM stage is a `Load` that produces the operand needed by the instruction in ID/EX (i.e., load destination equals a source register), a single-cycle stall is inserted. This stall is implemented by the Hazard Unit: it prevents IF/ID and PC update for one cycle and inserts a bubble (NOP) into ID/EX.

7.2 Control hazards (branches)

- **Static branch prediction: Not-Taken**. Branches are evaluated in **ID**. If a taken branch is detected, the IF/ID register is flushed and the PC is updated to branch target. This flush introduces a **1-cycle penalty**.
- Implementation detail: The control signals for the flushed IF/ID are zeroed so that a NOP is executed in the pipeline stage after the flush.

7.3 Structural hazards

The design uses a single unified memory for instructions and data (Von-Neumann). To avoid structural hazards the memory is designed to allow instruction fetch and data access in separate phases of the cycle or uses separate memory ports in simulation; if using single port memory in hardware, the pipeline may need to schedule accesses (this is a design trade-off and must be reflected in Phase 2 implementation).

8 ALU and flags behavior

- ALU is 32-bit. Addition and subtraction update Carry flag `C`, Zero flag `Z`, and Negative flag `N`. Logical operations update `Z` and `N`.
- SETC sets `C` = 1.
- NOT and INC update `Z` and `N` as specified in the ISA.

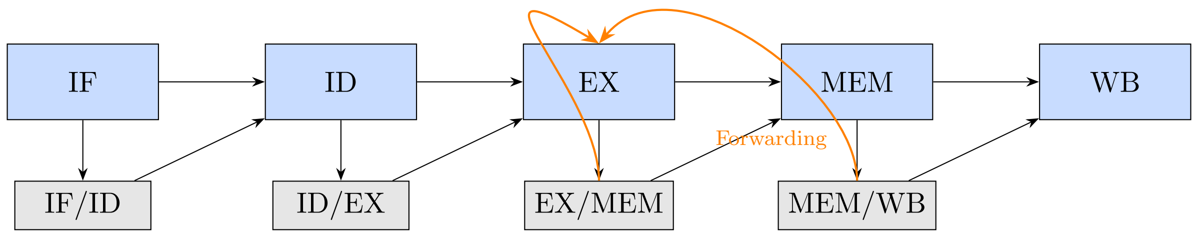
9 Stack, interrupts and reset

- **SP initial value:** $2^{18} - 1$ (per your ISA).
- **Interrupt handling:** When a non-maskable interrupt occurs, the processor finishes currently fetched instructions already entered into the pipeline, then the address of the next instruction (in PC) is saved on top of the stack and PC is loaded from memory address 1 (i.e., `PC := M[1]`). Flags are preserved (or saved as specified). **RTI** pops PC and restores flags.
- **Reset:** `PC := M[0]` (reset vector).

10 Assembler / encoding notes (for Phase 2)

- The assembler must set the HasImm bit and generate the immediate/target successor word whenever needed.
- For I-type and J-type instructions the assembler should emit two consecutive 32-bit words: the instruction word and the immediate/target word.
- Provide consistent syntax: e.g., `IADD R1, R2, 123` will set HasImm=1, Type=01, func=IADD, and place #123 as next memory word.

11 Pipeline diagrams (TikZ)



12 Testing and verification plan (Phase 1 expectations)

- Prepare assembly test programs that exercise:
  - All arithmetic, logical, and shift instructions.
  - Loads and stores with different offsets (testing load-use hazards).
  - Branches (taken and not-taken) to verify branch penalty handling and flush behavior.
  - Interrupt sequence and RTI/INT handling.
  - PUSH/POP correctness and stack boundary corner cases.
- Prepare a waveform viewer DO file (ModelSim or equivalent) showing: `R0-R7, PC, SP, Flags, CLK, Reset, Interrupt, IN.port, OUT.port`.
- Verify forwarding unit correctness by constructing tests that would otherwise produce RAW hazards.

13 Provided schematic (reference)

Below is a placeholder for the large schematic you provided. In Overleaf, upload your schematic image file `8b931ffb-a90b-4976-bea4-08406a0a1ce7.png` into the project root and the figure will be included here.

14 Appendix: Quick reference / encoding rules

- Always set the 7-bit opcode with the HasImm bit and Type bits. If HasImm=1, the assembler must append the immediate word after the instruction word.
- Word addressing: PC counts 32-bit words. After fetching an instruction at PC, normally PC := PC + 1. For instructions with a successor immediate word, fetch semantics must ensure the immediate word is available in ID/EX or MEM stage (implementation detail left to Phase 2).
- Stack grows downward: `PUSH` stores Rdst into memory at address SP, then SP := SP - 1. `POP` increments SP then loads from X[SP].
- Interrupt handling: on INT, push PC+1 then set PC := M[1]. RTI pops PC and resumes; flags are restored as designed.

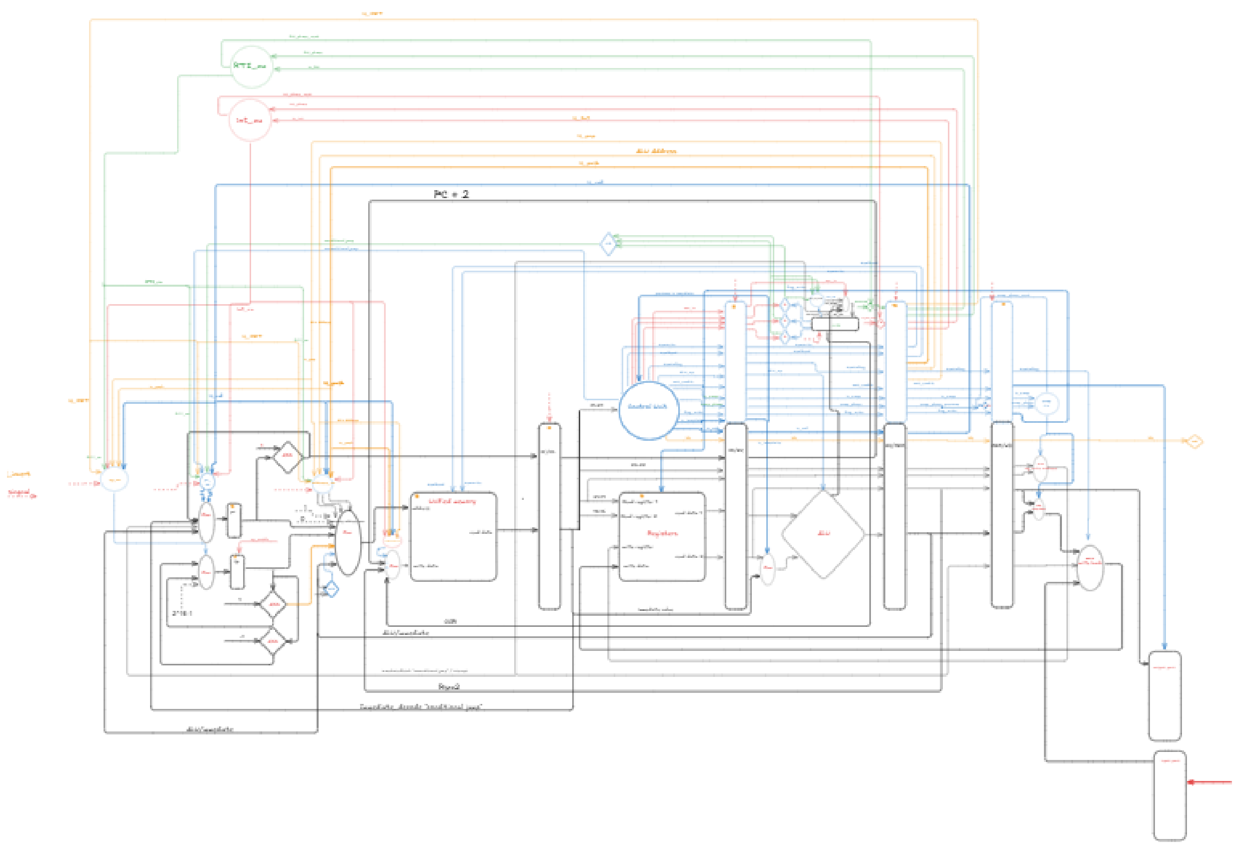


Figure 1: Full 5-Stage Pipelined Processor Schematic (Provided by the team).

15 Concluding remarks

This phase 1 document captures your architecture choices and provides a clear specification for Phase 2 (VHDL implementation, assembler, simulation). For Phase 2 we will:

1. Implement each block in VHDL (register file, ALU, memory wrapper, control, forwarding, hazard detection).
2. Integrate into a top-level pipelined module.
3. Produce testbenches that load the assembled memory file and run simulation waves for submission.

Prepared automatically to match the design choices you provided. Update opcodes or function codes if you prefer different assignments; the file is purposely extensible.