Computer Architecture 2025 Fall Lab 2

1. Problem Description

- In this homework, you are going to **extend Lab 1 into a pipelined RISC-V CPU supporting forwarding and hazard detection units.**
- CPU features:
  - 32 registers
  - 1KB instruction memory
  - Input: 32-bit binary instructions
  - Save arithmetic results into registers
- New in Lab2: support additional instructions: **lw, sw, beq, bne, nop**
- **Evaluation: dump register and data memory values after each cycle.**

1.1 Data Path

- Figure 1 describes the CPU data path.
  - **Provided by TAs: Registers, PC, Instruction Memory, Data Memory.**
- **You may implement** the modules listed in Section 1.6, and integrate them in `CPU.v`. It will be easier if you follow Figure 1
- To support pipeline execution, the first step is adding pipeline registers to the CPU. Pipeline registers store control signals and data from the **previous stage** and isolate each **stage**.
  - You have to use the non-blocking assignment in your pipeline register to get the correct result.
  - You have to properly initialize reg values in your pipeline registers.

- **Remember** the Stall and Flush **signals** in 1.3.

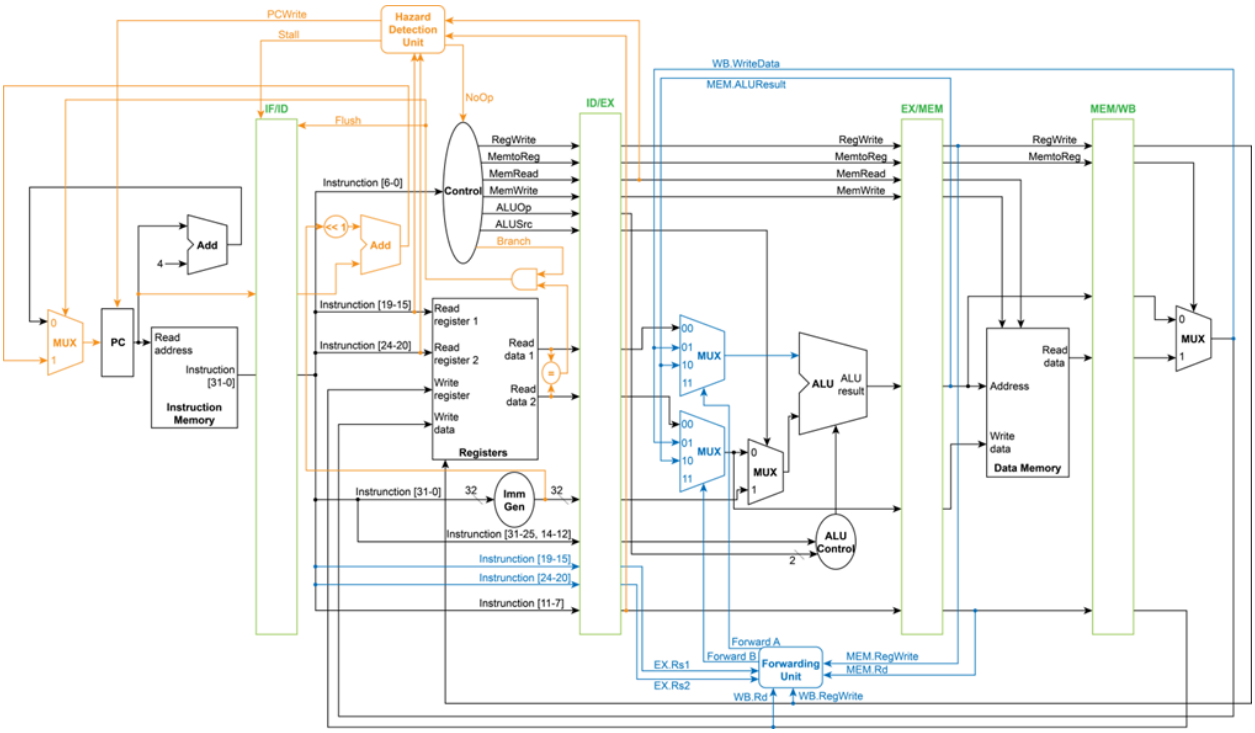


Figure 1 Final Datapath after Adding Forwarding and Hazard Detection Unit

1.2 Data Hazard and Forwarding Unit

- Example:

```
add t3, t1, t2
add t4, t1, t3
```

  - In the pipeline, `t3` is not written back until the **5th cycle**.
  - But the second instruction enters **EX stage at the 4th cycle**, leading to a wrong result compared to single-cycle execution.

- **Solution (Forwarding Unit)**
  - Forward the ALU result of the first instruction directly into the EX stage of the second instruction.
  - Forwarding sources:

- Forwarding is only required to **EX stage, f**orwarding to the **ID stage** is **not required**.

MUX	value	Source	Explanation
ForwardA	00	ID/EX	The first ALU operand comes from the register files
	10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
	01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB	00	ID/EX	The second ALU operand comes from the register files
	10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
	01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Table 1 Forwarding Control

- **Special Case (branch with dependency → will not be tested)**
  - Example sequence:

    This requires forwarding logic to ID stage, but such cases will not be e valuated.

    add x5, x6, x7
    beq x5, x4, BRANCH

  - **The** TA will insert enough NOP before beq or bne if there exists dependency, so you don't need to consider data dependency for beq or bne

```
01000000101101010000011000110011 // sub x12,x10,x11
00000000000000000000000000010011 // nop
00000000000000000000000000010011 // nop
00000000000000000000000000010011 // nop
00000000000001100000010001100011 // beq x12,x0,+8
```

1.3 Hazard Detection, Stall, and Flush

- **Load-use hazard**
  - Forwarding cannot resolve hazards caused by load instructions.
  - When the next instruction depends on the result of a load, a **stall** (nop) must be inserted so the data can be written back before being used.
  - Use ID_Stall in the CPU module to indicate a stall in the current cycle.
- **Control hazard (Branch)**
  - Introducing branch instructions (e.g., `beq, bne`) causes control hazards.
  - If the branch prediction is wrong, the pipeline must **flush** the incorrectly fetched instructions.
  - Use ID_FlushIF in the CPU module to indicate a flush in the current cycle.
- **Hazard Detection Unit**
  - Purpose: detect when to apply a **stall** or **flush**.
  - Rule: if the destination register (`rd`) in the EX stage matches the source registers (`rs1` or `rs2`) in the ID stage,

    → insert a **nop** (bubble) into the pipeline to ensure correctness.

In the CPU module, do not modify the names for these two signals

```verilog
module CPU
(
    clk_i,
    rst_n
);

// Ports
input           clk_i;
input           rst_n;

// Do not change the name of these 2 signals
wire ID_Stall;
wire ID_FlushIF;
```

1.4 Supported Instructions

- `and rd, rs1, rs2` (bitwise AND)
- `xor rd, rs1, rs2` (bitwise XOR)
- `sll rd, rs1, rs2` (logical shift left, use rs2[4:0])
- `add rd, rs1, rs2` (addition)
- `sub rd, rs1, rs2` (subtraction)
- `mul rd, rs1, rs2` (multiplication)
- `addi rd, rs1, imm` (addition with sign-extended imm)
- `srai rd, rs1, imm` (arithmetic right shift, imm[4:0])
- `lw rd, offset(rs1)` (load word, imm[11:0])
- `sw rs2, offset(rs1)` (store word, imm[11:5] & imm[4:0])
- `beq rs1, rs2, imm` (branch if equal, imm[12,10:5] + imm[4:1,11])
- `bne rs1, rs2, imm` (branch if not equal, imm[12,10:5] + imm[4:1,11])
- `nop` (do nothing)

- `ebreak` (similar to `nop`, but terminates execution **after all pipeline stages finish**)
  - The TA will insert 5 ebreak instructions to ensure all the processes are finished

```
0000000_10000_00111_010_00000_0100011 // sw x16, 0(x7)
0000000_10001_00000_010_00100_0100011 // sw x17, 4(x0)
00000000_0001_000000000000_01110011  // ebreak
00000000_0001_000000000000_01110011  // ebreak
00000000_0001_000000000000_01110011  // ebreak
00000000_0001_000000000000_01110011  // ebreak
00000000_0001_000000000000_01110011  // ebreak
```

1.5 Input / Output Format

- Provided: `testbench.v`, `instruction.txt`.
- Compile all together. Example:

  iverilog -g2012 -o CPU.out ./tb/*.v ./supplied/*.v ./src/*.v
  (iverilog -g2012 -o cpu.out <tb::path>/*.v <supplied::path>/*.v <source::path>/*.v)

- `instruction.txt`: one 32-bit ASCII binary string per line.
  - Should be placed at the same directory as CPU.out
  - Underscores `_` and `// comments` could be ignored.
  - You can modify here to test different cases

```
// Load instructions into instruction memory
// Make sure you change back to "instruction.txt" before submission
$readmemb("instruction.txt", u_CPU.u_Instruction_Memory.memory);
```

- `output.txt`: dump of registers after each cycle (generated automatically).

1.6 Required Modules (You may integrate it)

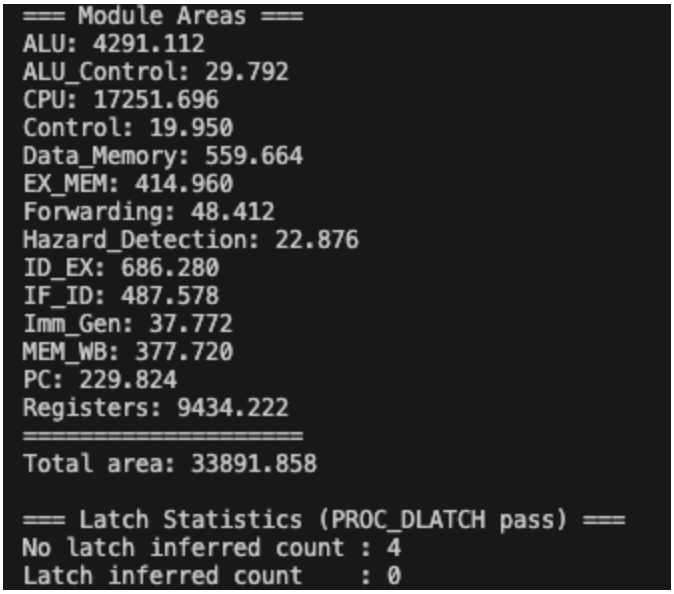
- **Control / ALU_Control**
  - Extend control signals to support load/store instructions (not included in Lab1).
- **Pipeline Registers**
  - Implement 4 pipeline registers to isolate the 5 pipeline stages.
  - Pass essential information to the next stage (use non-blocking assignment).
- **Forwarding Unit**
  - Detect data hazards and control forwarding.
  - Use two multiplexers to forward data from MEM stage or WB stage to EX stage.
- **Hazard Detection Unit**
  - Handle load-use hazard by inserting stall (nop).
  - Handle control hazard by flushing pipeline when necessary.
- **Others**
  - You may add extra helper modules if needed.
  - The Figure 1 datapath is a recommendation; details can be adjusted as long as CPU works correctly.
- **CPU (Top Module)**
  - Use structural modeling to connect inputs and outputs of all modules.
  - Remember the ID_Stall and ID_FlushIF signals !!!!!!

1.7 Synthesizability and Area

- All RTL code must be synthesizable (no unintended latches).
- The TA will check your design using **yosys** (`synth.log`). You do not need to modify any yosys scripts. The results can be found in Section 10.6 of `./log/synth.log` , including the estimated area of your module.

- run parse.py to get area and latch results, paste the results, and fill the numbers in the report

---



1.8 Testcases

- TAs will test your modules with **public** and **hidden** testcases.
- You may implement a Python simulator for debugging. The TA will use their own simulators to generate test cases from C programs and compile them into the corresponding assembly code.
- The TA will compare the output of your testbench (TB) with the expected output, including register value, data memory value, and the number of both flush and stall. They must match **exactly (100%)** in order to receive credit, so do not modify the provided TB.

```
cycle =           0, Stall = 0, Flush = 0
PC =          0
Registers
x0 =          0, x8  =          0, x16 =          0, x24 =        -24
x1 =          0, x9  =          0, x17 =          0, x25 =        -25
x2 =          0, x10 =          0, x18 =          0, x26 =        -26
x3 =          0, x11 =          0, x19 =          0, x27 =        -27
x4 =          0, x12 =          0, x20 =          0, x28 =         56
x5 =          0, x13 =          0, x21 =          0, x29 =         58
x6 =          0, x14 =          0, x22 =          0, x30 =         60
x7 =          0, x15 =          0, x23 =          0, x31 =         62
Data Memory: 0x00 =          5
Data Memory: 0x04 =          6
Data Memory: 0x08 =         10
Data Memory: 0x0C =         18
Data Memory: 0x10 =         29
Data Memory: 0x14 =          0
Data Memory: 0x18 =          0
Data Memory: 0x1C =          0
```

2.1 Report

- **Explain modules** in plain language (not Verilog).
- Example (acceptable): describe PC behavior with reset/clk/start.

> PC module reads clock signals, reset bit, start bit, and next cycle PC as input, and outputs the PC of current cycle. This module changes its internal register “pc_o” at positive edge of clock signal. When reset signal is set, PC is reset to 0. And PC will only be updated by next PC when start bit is on.

- Example (unacceptable): just pasting Verilog code. (0 point)

The inputs of PC are clk_i, rst_i, pc_i, and ouput pc_o.
It works as follows:

```
always@(posedge clk_i or negedge rst_i) begin
    if(~rst_i) begin
        pc_o <= 32'b0;
    end
    else begin
            pc_o <= pc_i;
    end
end
```

- Report also includes:
  - Synthesizability / latch check screenshot.
  - Development environment (OS, compiler, IDE).

3.1 Development Environment

Requirement

- **Docker** is required.

Directory Structure (after unzip)

```
Lab2/
├── Lab2_spec.pdf
├── Lab2_slide.pdf
├── Lab2/
│   ├── Makefile
│   ├── code/
│   │   ├── src/            # <Implement your CPU here>
│   │   ├── supplied/       # <supplied codes, don't modify them>
│   │   └── tb/             # <you may add debug print here>
│   ├── docker-compose.yml
│   ├── dockerfile
│   ├── judge.yaml
```

```
│   ├── parse.py            # calculate area from ./log/synth.log
│   ├── testcases/          # <sample testcases>
│   └── log/                # <logs after execution>
```

💡 **You should modify files in** `code/src/`

Do not change anything in `code/supplied/` `code/tb/`

3.2 Run with Docker

After implementation, execute:

```
sudo make run
```

- Results (logs, text, waveforms) will be generated under `log/`.

3.3 Run without Docker (alternative)

If you don't have Docker, run on **Ubuntu 22.04** with **iverilog 11.0-1.1 & Yosys & nangate45**

```
cp testcases/instruction_n.txt instruction.txt   # tb 會去讀 instruction.txt
iverilog -g2012 -o CPU.out ./tb/*.v ./supplied/*.v ./src/*.v
(iverilog -g2012 -o cpu.out <tb::path>/*.v <supplied::path>/*.v <source::path
>/*.v)

diff -u output.txt testcases/output_n.txt       # 會去比較 tb 產生的 output.txt

# 合成
yosys -p "
  read_verilog and2.sv;
  hierarchy -top and2;    #要改成 Top module 名字: CPU
  proc; opt; fsm; opt; memory; opt;
  techmap; opt;
  #make sure nangate libraray path is correct
    dfflibmap -liberty ~/nangate45/NanGate45/lib/NangateOpenCellLibrary_typ
ical.lib
```

```
    abc       -liberty ~/nangate45/NanGate45/lib/NangateOpenCellLibrary_typic
al.lib
    stat      -liberty ~/nangate45/NanGate45/lib/NangateOpenCellLibrary_typica
l.lib
  write_verilog -noattr mapped.v
" > ./log/synth.log
```

area results
python parse.py

⚠️ **Note:** You will get **0 points** if your code cannot run correctly on the official environmen

---

4.1 Submission Rules

- Directory structure after unzip:

```
studentID_lab2/
  ├── src/   (all Verilog codes you wrote)
  ├── studentID_lab2_report.pdf
```

- Do **not** include: Instruction_Memory.v, PC.v, Registers.v, testbench.v, instruction.txt, output.txt.
- File name: `studentID_lab2.zip`.
- Submit via NTU COOL.
- Deadline: 2025/11/25 (Tue.) 14:20.

5.1 Evaluation Criteria

- **Programming (80%)**
  - Public testcases: 60% (6 × 10%)

- Hidden testcases: 20% (5 × 4%)
- Not synthesizable: −10%
- Compilation error (naming, or minor errors in 3 lines): −5%

- **Report (20%)**
  - Latch & Area results (10%)
  - Description (10%)
- **Other penalties**:
  - Plagiarism: get 0 in this homework, and TA will inform teacher
  - Submitting unnecessary files (tb or supplied): -5%
  - Late submission: -10% per day, at most a week
  - Major mistakes causing compilation error → programming part 0. (Judeged by TA)
  - Wrong directory format: −5%.

Contact

Questions: email **eclab.ca.ta@gmail.com**