ECE4700J Lab 3 - RISCV Five-Stage Pipeline Optimization

Mingjian Li

UM-SJTU JI

lmjhshxhc@sjtu.edu.cn

June 3, 2024

Overview

Lab Logistics

Lab 3 Administrivia

- Due time: 23:59 on June 18th (Beijing time)
- Contents
  - Build a fully functional five stage pipeline from one with no ability to resolve the hazards
- Starter Code
  - A 'broken' 5 stage pipeline which executes one instruction every 5 stages to avoid all the hazards(Add, Noop, Noop, Noop, Noop, Add ...)
- Task
  - Add codes so that the pipeline can resolve hazards (data forwarding, always not taken strategy)

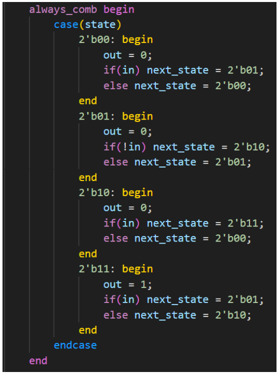
Feedback from lab1

lab1 feedback

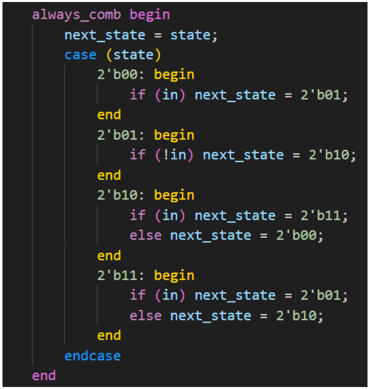
- If we are trying to build a synchronous system (it always is throughout the 470 course), asynchronous reset is not necessary
- You can just write something like
  assign a = (b == 1);
  rather than
  assign a = (b == 1)? 1:0;

lab1 feedback

- Be always aware that non-blocking assignment should always be in an ‘always_ff‘ block.
- For a variable in ‘always_comb‘, all conditions should be considered.



```
always_comb begin
    case(state)
        2'b00: begin
            out = 0;
            if(in) next_state = 2'b01;
            else next_state = 2'b00;
        end
        2'b01: begin
            out = 0;
            if(!in) next_state = 2'b10;
            else next_state = 2'b01;
        end
        2'b10: begin
            out = 0;
            if(in) next_state = 2'b11;
            else next_state = 2'b00;
        end
        2'b11: begin
            out = 1;
            if(in) next_state = 2'b01;
            else next_state = 2'b10;
        end
    endcase
end
```



```
always_comb begin
    next_state = state;
    case (state)
        2'b00: begin
            if (in) next_state = 2'b01;
        end
        2'b01: begin
            if (!in) next_state = 2'b10;
        end
        2'b10: begin
            if (in) next_state = 2'b11;
            else next_state = 2'b00;
        end
        2'b11: begin
            if (in) next_state = 2'b01;
            else next_state = 2'b10;
        end
    endcase
end
```

Advanced SystemVerilog Syntax

struct

- Used to define a packet of signals.
- Definition syntax:

```
typedef  struct  packed {
      logic  valid ;
      INST  inst ;
      logic  [`XLEN-1:0] NPC;
      logic  [`XLEN-1:0] PC;
} IF_ID_PACKET;
```

- Usage syntax:

```
IF_ID_PACKET if_id_packet;
assign  if_id_packet . valid  =  ...;
```

union

- union allows one packet of data to be represented in different ways.
- Definition syntax: (complete examples are in sys_defs.svh file of lab starter file)

```
typedef union packed {
        logic  [31:0]  inst;// The data stored in this union
    // " fields view" to the struct
        struct packed {
                logic  [11:0]  imm;
                logic  [4:0]   rs1; //base
                logic  [2:0]   funct3;
                logic  [4:0]   rd;  //dest
                logic  [6:0]   opcode;
        } i; //immediate or load instructions
        struct packed {
                logic  [19:0]  imm;
                logic  [4:0]   rd;
                logic  [6:0]   opcode;
        } u; //upper immediate instructions
    ...
} INST; //instruction typedef, this should cover all types of instructions
```

union

- Usage syntax

```
INST inst;
 logic  [11:0]  imm_12;
 logic  [19:0]  imm_20;
assign imm_12 = inst.i.imm;
assign imm_20 = inst.u.imm;
```

Often used in instruction definition as there are multiple types of instructions (i type, j type, b type ...)

RISCV Processor Recap

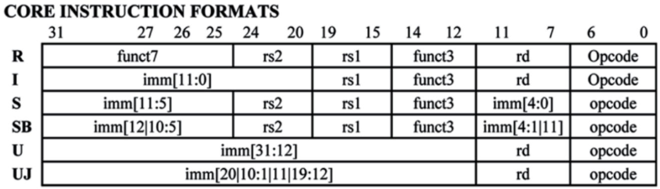
Overview

In this lab, we will try to recap you some fundamental knowledge surrounding RISC-V processors

- RISC-V ISA
- hardware structure: (single-cycle version without forwarding path, hazard detection, branch detection...)
- different kinds of hazards

ISA Overview

- Instruction format:



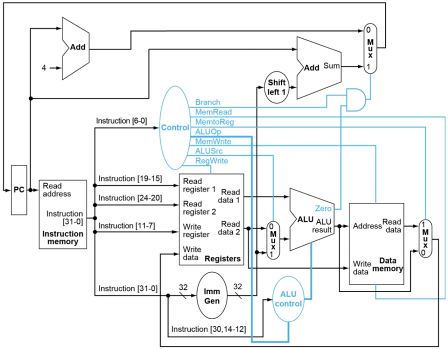
CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	Opcode
I	imm[11:0]		rs1	funct3	rd	Opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

- For more info, pleae refer to CANVAS→Files→RISC-V.

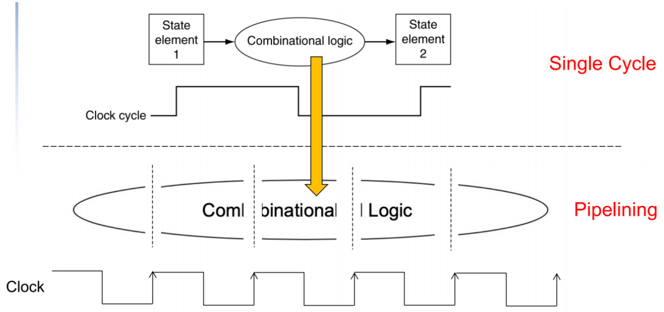
Single-Cycle Processor

- Generally, execute one instruction in one clock cycle
- Only one instruction is within the below structure at a time



Five-Stage Pipeline Processor

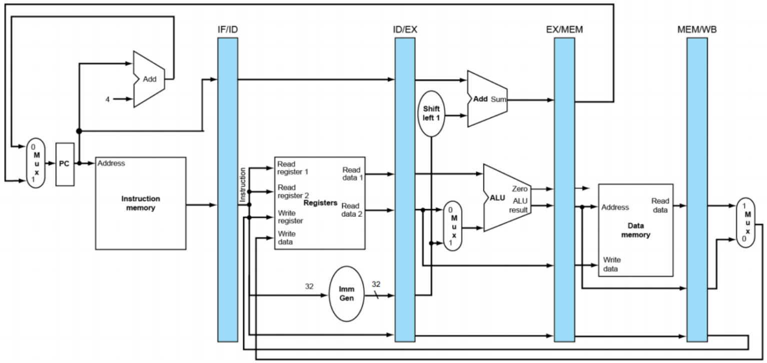
- Break the above structure into 5 stages and insert stage registers



- So that we can have 5 instructions concurrently existing in the structure at a time
- Clock period can thus be reduced (shorter critical path)

Five-Stage Pipeline Processor (Cont.)

- Hardware structure:



Five-Stage Pipeline Processor (Cont.)

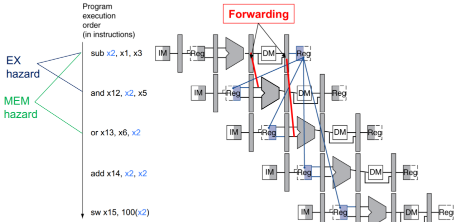
- Stages
  - IF: Instruction fetch. Components includes PC, instruction memory, etc.
  - ID: Intruction decode. Read data from register file, generate immediate, etc.
  - EX: Execution. Calculate using ALU, etc.
  - MEM: Access data memory. Read or write data memory.
  - WB: Write back. Write data back to register file.

Hazards

- Categories of hazards:
  - Data hazards: Need to wait for previous instruction to complete its data read/computation.
  - Control hazards: Decision on control action depends on previous instruction.
  - Structure hazards: A required resource is busy

Hazards (Cont.)

- Resolve data hazards:
  - Forwarding: Use a forwarding path to access necessary results in advance



  - Stall: In some cases, forwarding is not enough (e.g. data has not been read from Dmem). So we need to wait for a while, i.e. insert a `nop` instruction

Branches

Resolve control hazards:

- Branch execution: In this lab, branch taken or not is figured out in EX stage and we assume always not-taken
- If taken: Flush all the instructions in ID, EX, MEM in the next cycle and jump to the target PC in IF stage to fetch a correct instruction

Memory

Resolve structural hazards:

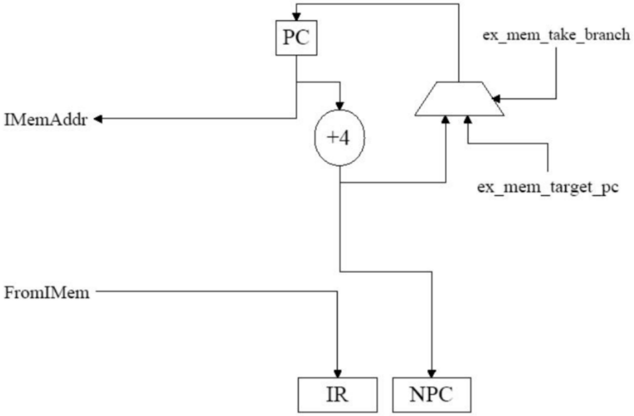
- In this lab, we are using the memory module which only support one access in one cycle (Data read, Data write and Instruction read should not happen together)
- Add stall on instruction read if it conflicts with data access

Tips for Lab 3

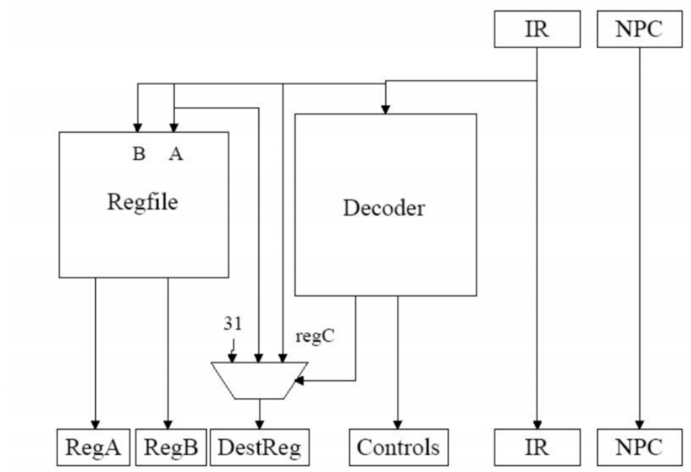
Tips for Lab 3

- Please read the manual carefully to ensure that your implementation meets the requirements.
- Please read the provided baseline design code for full understanding.
- Resolutions on different hazards may conflict with each other, try to consider this and think about their priority

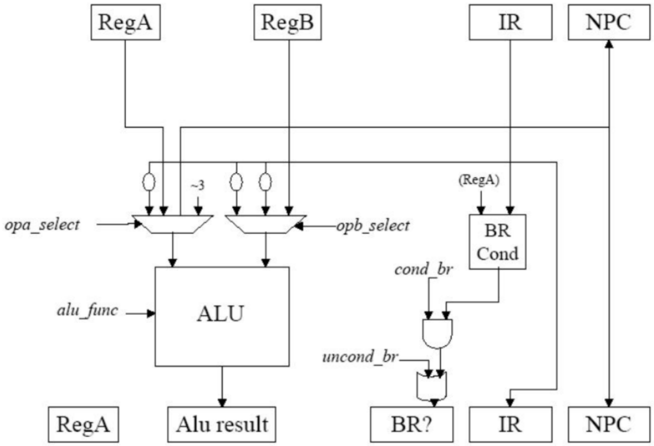
Hints - IF stage



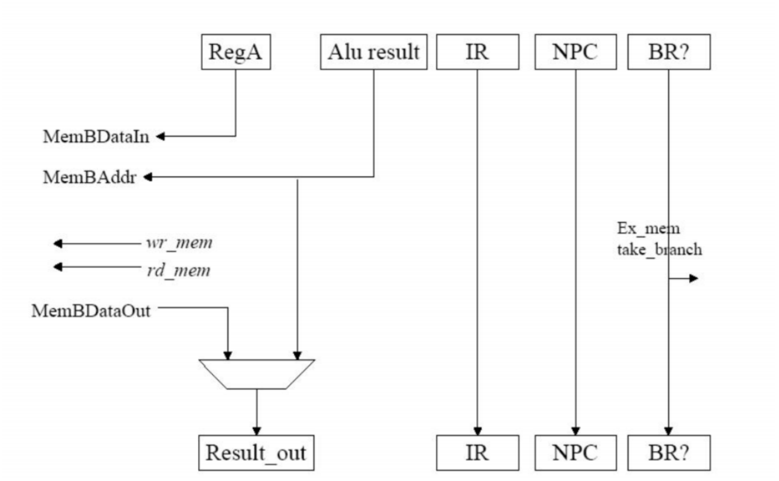
Hints - ID stage



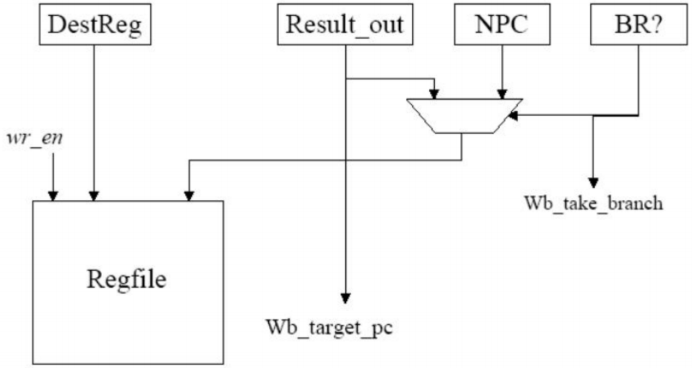
Hints - EX stage



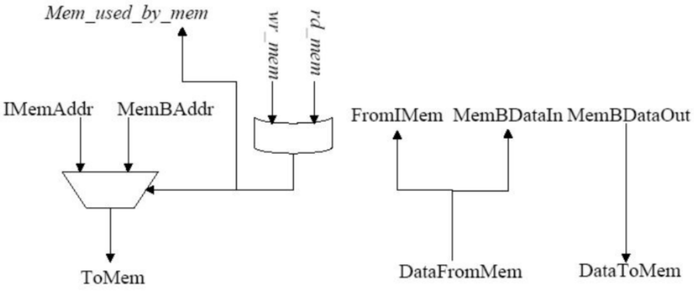
Hints - MEM stage



Hints - WB stage



Hints - Memory arbitration



Reference

[1] Haoyang Zhang, ECE4700J SU 2023 Lab2 & Lab3
[2] Runxi Wang, VE370 SU 2022 RC2 & RC3

Thank you!