Computer Architecture

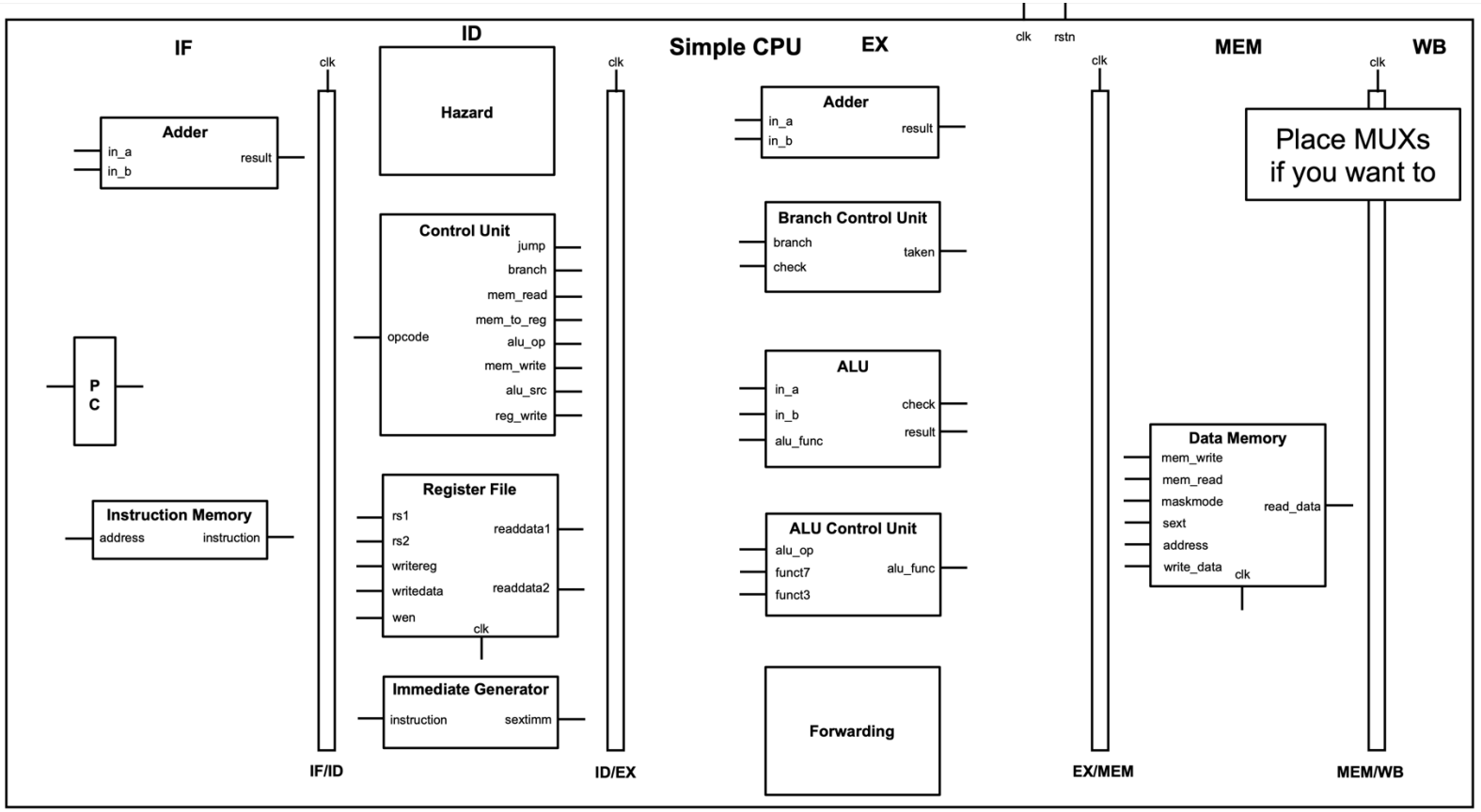
Lab 2: Pipelined CPU Implementation

Jaewoong Sim
Electrical and Computer Engineering
Seoul National University

Contributors: Junseo Lee, Joonho Hwangbo, Jungi Lee

Lab Overview (1/3)

- **Goal**: Implement pipelined CPU design with Verilog
  - Addition of pipeline registers for intermediate values
  - New units (forwarding unit, hazard detection unit)



Lab Overview (2/3)

- There are total 6 tasks + task-fibo + task-sum that you need to pass
  - **Task1**: 1 & 2
  - **Task2**: 1 & 2
  - **Task3**: 1
  - **Task4**: 1 & 2
  - **Task5**: 1
  - **Task6**: 1 & 2
  - **Task-fibo**
  - **Task-sum**

Lab Overview (3/3)

- Lab2 will be split into five parts, which you can implement step by step

- DO NOT modify any I/O ports in the modules
  - Except for forwarding / hazard detection unit + pipeline register modules

- 5 Steps
  - **Part 0**: Complete all the modules that you implemented in Lab1
  - **Part 1**: Add pipeline registers to single-cycle CPU
  - **Part 2**: Implement forwarding
  - **Part 3**: Implement hazard detection: 1. flush
  - **Part 4**: Implement hazard detection: 2. stall

Part 0: Complete Lab1 Modules

- You first need to complete the modules below
  - `module/control/alu_control.v`
  - `module/control/branch_control.v`
  - `module/control/control.v`
  - `module/memory/data_memory.v`
  - `module/operation/alu.v`
  - `module/operation/immediate_generator.v`
  - Fortunately, you can use the modules that you have implemented in Lab1 ☺
    - However, be careful to match the module name (e.g. `Alu.v` → `alu.v`)!
  - In addition, you need to add some muxes in `simple_cpu.v` as you did in Lab1
  - You do not have to care about branch instructions at this part (just go to next instruction)
    - This will be covered by hazard detection unit (Part 3)!

Part 1: Add Pipeline Registers to 1-Cycle CPU

- The modules for the pipeline registers are already in the `module/memory/` directory and are added to `simple_cpu.v`

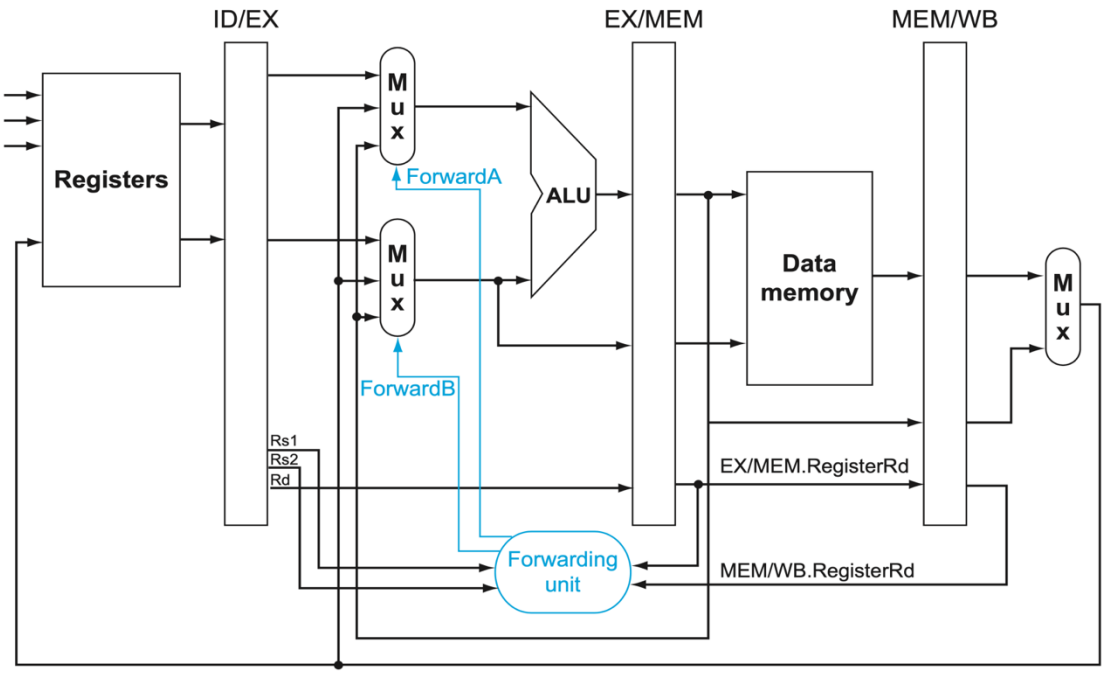
- You just need to …
  - Declare & connect the wires in `simple_cpu.v`
  - Implement the pipeline register modules
    - `module/memory/idex_reg.v`
    - `module/memory/exmem_reg.v`
    - `module/memory/ifid_reg.v`
    - `module/memory/memwb_reg.v`
  - You do not need to care about flushes and stalls for now. Just ignore them.

- After completing Part 1, task1 ~ task3 should pass

Part 2: Implement Forwarding

- Now implement `module/control/forwarding.v` to generate forwarding control signals
  - Method: Forward MEM / WB stages' results to Execute Stage (v2)
    - Refer to the lecture slides and textbook for how to create the signals

- After completing Part 2, task1 ~ task4/1 should pass



Part 3: Implement Hazard Detection Unit (Flush)

Pipeline Flush on Branch Misprediction

- In the case of a branch, the next instruction (PC + 4) will be executed through the pipeline before the resolution of the branch.

- Thus, we need to …
  1. Flush the pipeline (instructions from the wrong path) **if the branch is resolved as 'taken'**
  2. Update PC with the branch target

- In `module/control/hazard.v`
  - Declare flush signals & assert them accordingly
    - Branch resolution should be done at the MEM stage
  - Add a new control signal to update PC with the branch target (taken or jump)
    - Update your mux for PC accordingly!

- Add flush signals to the input of the pipeline register modules
  - Update the pipeline registers properly when the flush signal is asserted

- After completing Part 3, task1 ~ task5/1 should pass

Part 4: Implement Hazard Detection Unit (Stall)

Data Hazard on an immediately preceding LW instruction

- Even with forwarding, RAW dependence on an immediately preceding LW instruction leads to a hazard

- Thus, we need to stall the pipeline in those cases!

- In `module/control/hazard.v`
  - Declare the stall signal & assert it when needed
  - Hint: You need to compare signals between ID and EX stages
    - Refer to the docs/Pipelined-CPU-diagram.pdf

- Add the stall signal to the input of the pipeline register module
  - Update the pipeline registers properly when the stall signal is asserted

- After completing Part 4, task1 ~ task6/2 & task-fibo & task-sum should pass

Tips for Lab2

- Before you start coding, draw the diagram (Pipelined-CPU-diagram) first!
  - You should instantiate additional modules and wires

- Complete implementing the modules that are used in Lab1
  - All the modules except for forwarding / hazard / pipeline register are the same as the ones in Lab1

- Think carefully how to implement forwarding & hazard detection units

- **Use GTKWave for debugging!!**
  - It is a very powerful debugging tool for Verilog coding
  - If you run `./simple_cpu`, it will output sim.vcd (value change dump file)
  - Run `gtkwave sim.vcd` ➔ add wires to the waveform at the right side

  ➔ (-) & (+) buttons at the top left are for zoom-out & zoom-in
  (You should be able to see the clock after a few zoom-outs)

Another Way to Debug (1/2)

- `simple_cpu` works by reading `inst.mem`, `register.mem`, and `data_memory.mem` in the `/data` directory

- Copy `inst_disassembled.mem` (rename it to `inst.mem`), `register.mem` from the task you want to debug to the `/data` directory

```
  input  clk,
  input  mem_write,
  input  mem_read,
  input  [1:0] maskmode,
  input  sext,
  input  [DATA_WIDTH-1:0] address,
  input  [DATA_WIDTH-1:0] write_data,

  output reg [DATA_WIDTH-1:0] read_data
);

  // memory
  reg [DATA_WIDTH-1:0] mem_array [0:2**MEM_ADDR_SIZE-1]; // change memory size
  initial $readmemh("data/data_memory.mem", mem_array);
  // wire reg for writedata
  wire [MEM_ADDR_SIZE-1:0] address_internal; // 256 = 8-bit address

  assign address_internal = address[MEM_ADDR_SIZE+1:2]; // 256 = 8-bit address

  // update at negative edge
  always @(negedge clk) begin
    if (mem_write == 1'b1) begin
-- VISUAL --
```

```
// instruction_memory.v

module instruction_memory
#(parameter DATA_WIDTH = 32)(
  input [DATA_WIDTH-1:0] address,

  output reg [DATA_WIDTH-1:0] instruction
);

localparam NUM_INSTS = 64;

reg[DATA_WIDTH-1:0] inst_memory[0:NUM_INSTS-1];
initial $readmemb("data/inst.mem", inst_memory);

always @(*) begin
  instruction = inst_memory[address[7:2]];
end


endmodule

-- VISUAL --
```

Another Way to Debug (2/2)

- Use the "`$monitor`" or "`$display`" functions to inspect the variables you want
- Do "`make`" in the root directory
- Run `./simple_cpu` in the root directory (don't run `python test.py`)

```
 4
 5 reg clk, rstn;
 6 wire [31:0] inst;
 7
 8 integer i;
 9
10 initial begin
11   clk  = 1'b0;
12   rstn = 1'b0;
13   $display($time, " ** Start Simulation **");
14   $display($time, " Instruction Memory ");
15   $monitor($time, " [PC] pc : %d", my_cpu.PC);
16   #60 rstn = 1'b1;
17   #4000;
18   rstn = 1'b0;
19   $display($time, " ** End Simulation **");
20
21   ////////////////////////////////////////////////////
22   // [WARNING] : DO NOT ERASE when using "test.py"
23   ////////////////////////////////////////////////////
24   $display($time, " REGISTER FILE");
   @
-- VISUAL LINE --
```

```
                   0 ** Start Simulation **
                   0 Instruction Memory
VCD info: dumpfile sim.vcd opened for output.
                   0 [PC] pc :          x
                   5 [PC] pc :          0
                  65 [PC] pc :          z
                4060 ** End Simulation **
                4060 REGISTER FILE
                4060 Reg[           0]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
                4060 Reg[           1]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
                4060 Reg[           2]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
                4060 Reg[           3]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
                4060 Reg[           4]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
                4060 Reg[           5]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
                4060 Reg[           6]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
                4060 Reg[           7]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
                4060 Reg[           8]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
                4060 Reg[           9]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
                4060 Reg[          10]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
                4060 Reg[          11]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
                4060 Reg[          12]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
                4060 Reg[          13]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
                4060 Reg[          14]:          x (xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)
```

How to Compile, Run, and Test Your Code

How to Compile, Run, and Test

- Do just same as what you did in Lab1
- Run: `make` & `./simple_cpu`
- Test: `python test.py`
- Please do "`make clean`" before zipping your code
  - It cleans up the directory (see `Makefile`)

Submission

- **Due**: 5/8 **(THU) 11:59 PM**

- **Late Policy**
  - 10% discount per day (5/9 12:00 AM is a late submission!)
  - After 5/12 (MON) 12:00 AM, you will get a **zero** score for the assignment

- **What to Submit?**
  - Your code that we can simulate and run
  - Submit all the files/directories in Lab2 (Do `make clean` first!)
  - 1-page document that describes your work

- **How to Submit?**
  - Upload your compressed file (zip) to eTL
  - **Format**: Your student ID_YOURLASTNAME_lab#
    - e.g.,) 2025-12345_KIM_lab2.zip
  - Please make sure you follow the format
    - **10% penalty for the wrong format**