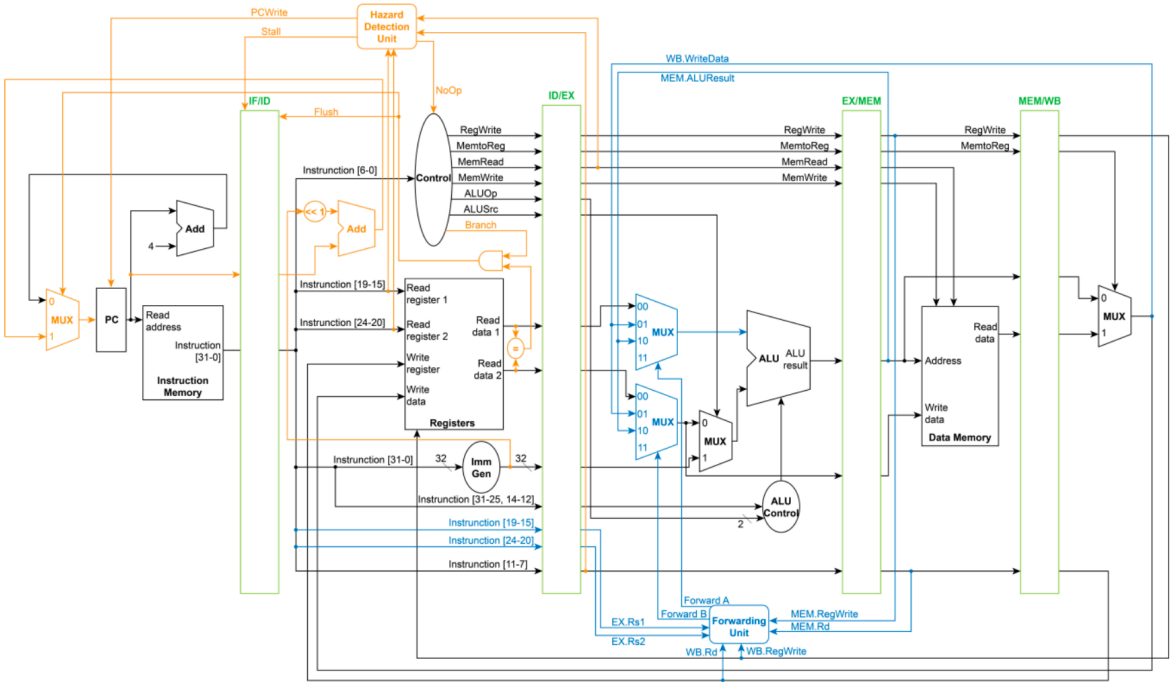
2020 Computer Architecture Project 1

	B07902053 許浩鳴	B07902133 彭道耘	B07902141 林庭風
workload	Report, Forwarding, Module Modification	Report, Connecting Modules	Report, Hazard Detection, Pipeline Registers

Development Environment

- OS: Ubuntu 18.04.
- Compiler: iverilog.
- IDE: Vim.
- Method to Debug: `gtkwave` and `$display`.

Final Datapath



Module Implementations

CPU

- input: clock signal(*clk_i*), reset signal(*rst_i*), and start signal(*start_i*).

This module is just a place to put all wires connecting all modules described in the following.

Forwarding

- input: *EX_RSaddr1_i*, *EX_RSaddr2_i*, *Mem_RegWrite_i*, *Mem_RDaddr_i*, *WB_RegWrite_i*, *WB_RDaddr_i*.
- output: two selected signals for each MUX3.

We implement this module to detect EX hazard or Mem hazard. The rules are shown in the following, which is followed by the spec.

```
if (MEM_RegWrite_i &&
    Mem_RDaddr_i != 0 &&
    Mem_RDaddr_i == EX_RSaddr1_i) Forward EX result to RS1
else if (WB_RegWrite_i &&
         WB_RDaddr_i != 0 &&
         WB_RDaddr_i == EX_RSaddr1_i) Forward Mem result to RS1

if (MEM_RegWrite_i &&
    Mem_RDaddr_i != 0 &&
    Mem_RDaddr_i == EX_RSaddr2_i) Forward EX result to RS2
else if (WB_RegWrite_i &&
         WB_RDaddr_i != 0 &&
         WB_RDaddr_i == EX_RSaddr2_i) Forward Mem result to RS2
```

Hazard Detection

- input: *IDEX_MemRead_i*, *IDEX_RDaddr_i*, *IFID_RSaddr1_i*, *IFID_RSaddr2_i*.
    - note that *IFID_RSaddr1_i* represents instruction[19-15] and represents *IFID_RSaddr2_i* instruction[24-20]
- output: *PCwrite_o*, *Stall_o*, *NoOp_o*.
    - possible outcomes:
        - (1, 0, 0) for non-stall situation
        - (0, 1, 1) when we need to stall

We implement this module to determine if **stall** operation should be operated. Specifically, if *IDEX_Memread_i* is 1 and *IDEX_RDaddr_i* equals either *IFID_RSaddr1_i* or *IFID_RSaddr2_i*, we need to do **stall** operation. The stall signal will be sent to IF/ID, and the data in IF stage will not be passed to ID stage until the next cycle. The rule are shown in the following.

```
if (IDEX_MemRead_i &&
   ((IDEX_RDaddr_i == IFID_RSaddr1_i) ||
    (IDEX_RDaddr_i == IFID_RSaddr2_i)) Stall
```

Pipeline Registers (IF_ID, ID_EX, EX_MEM, MEM_WB)

To seperate data into five stages, we implement four modules, to pass data to the next stage. Besides, they are all given the clock signal(*clk_i*) and start signal(*start_i*). When *start_i* is 0, we initial all registers as 0. After that, once *clk_i* is set to 1, we pass all data needed to the next stage.

Additionally, we implement the action **stall** and **flush** in IF_ID module.

And

- input: branch signal and a boolean value if two data are the same in register.
- output: do the **and** operation for two inputs to determine if **flush** operation is needed. If **flush** operation is needed, then we pass flush signal to IF_ID and MUX_PC, which resets PC at ID stage. Thus, the pipeline registers of the instruction after `beq` will be set zero, and the instruction in the next cycle will jump to the one spcified in beq.

Adder

- common operation
  - input: reads from two wires *data1_i* and *data2_i*.
  - output: put *data1_i* + *data2_i* to *data_o*.
- Adder_PC
  - input: 4 and current PC address
  - output: next address (PC+4) to MUX_PC
- Adder_Branch
  - input:
    - imm gen that is left-shifted by 1
    - current PC address
  - output: branch address to MUX_PC

ALU

- input: ALU control signal, *data1_i* and *data2_i*.
- output: 32-bit calculated result and a zero signal.
- apply arithmetic operation according to a 3-bit ALU control signal.

ALU_Control

- input: *funct3* , *funct7* and ALU opcode.
- output: 3-bit select signal to ALU.
- signals for addi, srai, and, xor, sll, add, sub, mul, sw, lw

Control

- input: *Op_i*, *NoOp_i*
  - *Op_i* represents instruction opcode (instruction[6:0])
  - *NoOp_i* comes from Harzard Detection Unit
- output: *ALUOp_o*, *ALUSrc_o*, *Branch_o*, *MemRead_o*, *MemWrite_o*, *RegWrite_o*, *MemtoReg_o*
- determine output signal values according to *Op_i*, and if *NoOp_i* is 1, then we do **stall** operation and set signals zero.

Sign_Extend (Imm Gen)

- input: 32-bit instruction.
- output: duplicate the MSB of 12-bit immediate by 20 times and extend to 32-bit.

We had some trouble when implementing this module since we ignored when parsing *sw* and *beq* instructions, the position where immediates stored are not the last 12 bits. While finding this problem, we just fixed it by simple case analysis.

MUX3

- input: *data1_i*, *data2_i*, *data3_i* and *select*.
    - *data1_i*: directly from register file
    - *data2_i*: WB_WriteData (from MEM_WB)
    - *data3_i*: MEM_ALUResult (from EX_MEM)
    - *select*: signal of the fowarding type
- output: choose one from the three data according to the select signal and put the selected data to *data_o*.
- two MUX3; one for ID_EX_RSaddr1 and another for ID_EX_RSaddr2

MUX32 (MUX_PC, MUX_ALU, MUX_WB)

- MUX_PC, MUX_ALU and MUX_WB are MUX for IF, EX, and WB stages
- MUX_PC
    - input: *IF_pc_add_out*, *ID_add_out* and *Flush*
        - *IF_pc_add_out*: from Adder_PC
        - *ID_add_out*: from Adder_Branch. It is selected when flush is true
    - output: put the select data to *data_o* (*IF_pc_in*).
- MUX_ALU
    - input: *data1_i*, *data2_i* and *select*.
        - *data1_i*: output from the MUX3 for ID_EX_RSaddr2
        - *data2_i*: output from Sign_Extend
        - *select*: ALUSrc signal
    - output: put the select data to *data_o* (*EX_ALU_input2*).
- MUX_WB
    - input: *WB_mux1*, *WB_mux2* and *WB_memto_reg*
    - output: put the select data to *data_o* (*WB_write_data*).

Other Difficulties Encountered

- The datapath pdf file prints by linux(ubuntu) can not be printed at 7-11.
    - Final solution: Use the android print to pdf instead.