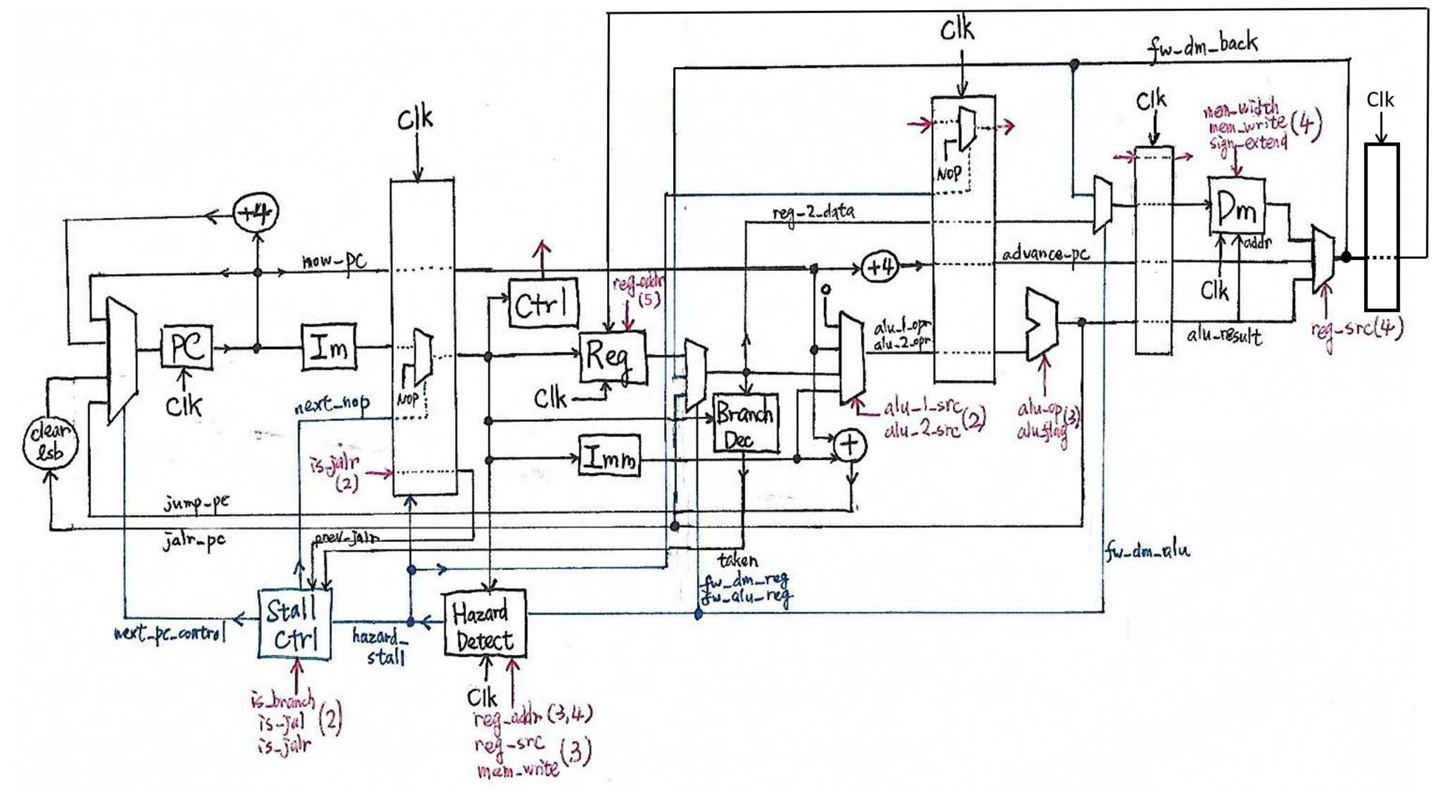
2019 Computer Architecture Project 1

Member

- 資工三 b06902021 吳聖福
    - Design the datapath. (100%)
    - Design and implement all the instructions. (100%)
    - Design and implement the hazard detection. (100%)
    - Design and implement fowarding control. (100%)
    - Write the report (40%)
- 資工三 b06902026 吳秉柔
    - Split the datapath into five stages (IF, ID, EX, MEM, WB). (100%)
    - Design and implement the IF-ID, ID-EX, and EX-MEM pipline register. (100%)
    - Design the MEM-WB pipline register. (100%)
    - Write the report. (20%)
- 資工三 b06902093 王彥仁
    - Implement the MEM-WB stage pipline register. (100%)
    - Calculate the number of stall and flush. (100%)
    - Test and check the correctness of all the components and the output. (100%)
    - Write the report. (40%)

Design & implementation

Datapath



Wires of control signals are omitted for simplicity. They're marked by red arrows instead. The number in the parenthesis indicates the source stage of the control signal (2=ID, 3=EX, ...).

Note that there are some differences between our datapath and that in the textbook. The most notable difference is that **fowarding is done at the end of the stage** rather than at the beginning.
The reason is that branch decision is done at the second stage, so in case of a hazard, the data must be forwarded from the third or fourth stage to avoid an additional stalling cycle. As a result, the types of forwarding are marked differently in this report (and in the code). For example, the original *MEM/WB to EX* fowarding becomes *MEM to ID* forwarding.

We've implemented the whole RV32IM instruction set (except system-related instructions). In this report, descriptions related with those additional instructions will be marked in green.

In the spec, `Data_Memory.v` is on the *MUST DELETE* list. However, the given data memory module is incorrect (moreover, it is necessary to make modifications to support the additional instructions; refer to Data Memory section for details). Thus, we make our submitted directory a git repository. It is initially in branch `hw4-must-delete-compat` where the files are compatible with the given *MUST DELETE* files, and the corrected (and extended) files are in branch `master`.

`CPU.v`
Contains the main CPU module, which connects all modules as in our datapath.

`MUX32.v`
Contains `MUX32_2` and `MUX32_4`, which are 2-way and 4-way multiplexers, respectively.
`Sign_Extend.v`
Contains `Immediate_Gen`, which outputs the immediate in instruction.
`Opcode.v`
Defines some opcode constants for better code readability.

`Pipline_Reg.v`
Contains four modules — `IF_ID`, `ID_EX`, `EX_MEM`, and `MEM_WB`. Each module represents a pipeline register between two stages.
`Control.v`
Contains a module `Control`, which outputs all the control signals.
Note that we've merged `ALU_Control` into this module (as opposed to HW3). We also removed `reg_write` control signal and use `rd = x0` to indicate `reg_write = false` to simplify hazard detection.
`ALU.v`
Contains the ALU module, which completes all possible arithmetic operations in the RV32IM instruction set.

`Adder.v`
Contains a simple adder module, which is used to calculate the next instruction address.

`Branch_Decision.v`
Contains a module `Branch_Decision` to decide whether the branch is taken.
`Hazard_Detection.v`
Contains two modules: `Hazard_Detection` and `Stall_Control`.
`Hazard_Detection`
This module implements hazard detection and fowarding control.

There are several situations to consider:

1. Load-use hazard

   A load instruction followed by another instruction using its result (in case of store instruction, it need to use the result as **the base address**). Below are two examples:

   ```
   lw  x1, offset(x2)
   add x2, x1, x4
   lw x1, offset(x2)
   sw x3, offset(x1)
   ```
   For this situation, we need to **stall one cycle** (which is done by output a signal to `Stall_Control` module), followed by a *MEM to ID* forwarding. If there are one

instruction between them, no stalling is needed, but *MEM to ID* forwarding are still necessary.

2. Register write-store hazard

   A register-writing instruction followed by a store instruction that uses its result as **the data to be written**. For example:

   ```
   lw  x1, offset(x2)
   sw  x1, offset(x3)
   ```

   For this situation, we need a *MEM to EX* forwarding. If there are one instruction between them, a *MEM to ID* forwarding is done instead.

3. Other data hazard

   A ordinary (i.e. not load) register-writing instruction followed by another instruction that uses its result. For example:

   ```
   add  x1, x2, x3
   addi x2, x1, 1
   ```

   For this situation, we need an *EX to ID* forwarding (note that *EX to ID* forwarding must override *MEM to ID* forwarding if they coexist). If there are one instruction between them, a *MEM to ID* forwarding is done instead.

`Stall_Control`
This module implements stalling mechanism.

There are several situations to consider:

1. Load-use hazard

   Stated in the previous section. The `Hazard_Detection` module gives the required signal. Note that one need to clear the branch taken signal in this case.

2. Taken branch or direct jump (JAL)

   Since the target address of jump is available at the ID stage, it is necessary to stall one cycle to wait for the address. This also triggers a flush (which replaces the current instruction in the ID stage by a NOP).

3. Indirect branch (JALR)

   The jump address of an indirect branch is available at the EX stage, so one needs to **stall two cycles** in this case. Implemented by adding a `prev_jalr` bit to the IF/ID register.

`Data_Memory.v`

The given data memory module is incorrect: it didn't shift the address, nor does it support unaligned access. Thus, we re-implemented the module in branch `master`. We also added two additional control signals (`width` and `sign_extend`) in order to support all load/store instructions (LB, LH, LBU, LHU, SB, SH).

`testbench.v`

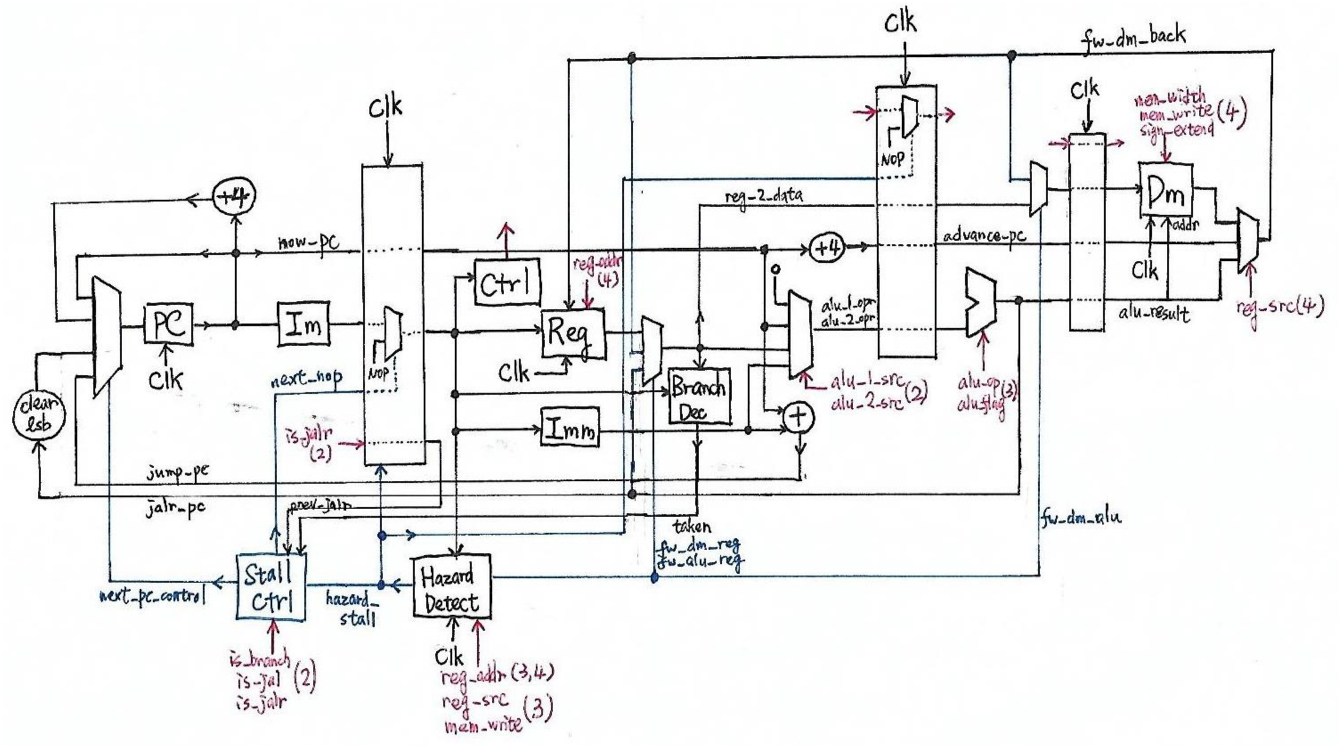
The number of stalls and flushes is counted in this module.
The number of stalls will increase when the hazard detection output that the current cycle should hazard stall(`hazard_stall` is 1).
The number of flushes will increase when the stall control output that the current cycle should be a nop(`next_nop` is 1).
We added a command line option `+file=[insfile]` to specify input instruction file. If not specified, it defaults to `instruction.txt`.
The memory printing code in given testbench is also inconsistent with the given data memory module (it seems to treat data memory as 8-bit cells; it outputs correct results when there's no unaligned write, though). In branch `master`, since we'd re-implemented data memory, the memory printing code is modified accordingly.

`mytest.v`, `testcase/`

`mytest.v` contains a testing module intended to provide an execution enviroment same as Jupiter simulator for debugging. It outputs the content of registers and PC **after executing each instruction** (as if it is executed by a single-cycle machine).
Jupiter simulator places the first instruction of the main function at PC=65544, and there are also unwritable address ranges, so adjustments must be made for load/store/JAL/JALR/AUIPC to work. Thus, in branch `hw4-must-delete-compat`, the initial `sp` are changed, and datapaths related to JAL/JALR/AUIPC are modified in `CPU.v`; in branch `master`, instruction(data) memory are only accessed by the lowest 10(12) bits, so that `mytest.v` can be implemented without modification of the CPU module.
`testcase/` contains a testdata generator `generate_rand_insr.cpp` and an automatic testing script `test.sh`. The script will compile the generator, generate random instructions, run the instructions by Jupiter and the implemented CPU (`../code/mytest.vvp`), and finally compare the results. If the results don't match, it will output a file `error.log` containing the instruction and the results of both programs.

Difficulties encountered and solutions of this projects

1. Our first design is as the following picture. We miss one pipeline register (MEM/WB), so the writing back to the register is earlier one cycle than the standard answer.

   It’s obvious to add a MEM-WB pipeline register to resolve the problem. (However, it seems that the pipeline register is somewhat redundant: in case of a load instruction (which is the only instruction that uses both MEM and WB stages), there is little difference between writing to pipeline register and writing to the register file.)

2. When implementing hazard stalling, we forgot to clear the branch taken signal, which caused CPU jumping to incorrect PC when a branch follows a load and the branch should be not taken but is decided taken before data forwarding. (Since this situation rarely occurs, we didn’t find this bug until we implemented `mytest.v`.)