2019 Computer Architecture Project 1

Member

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

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.

TODO: state additional instruction support and the git branch

`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 foward-ing 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. 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. 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 file is incorrect: **TODO**

`testbench.v`

TODO

Difficulties encountered and solutions of this projects

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.)