Computer Architecture 2017 Fall Project #1 Report

B03901078 蔡承佑

December 11, 2017

1 Implementation

Based on the previous model of single-cycle CPU, some modules are common, such as `ALU`, `ALU_Control`, `PC`, `Sign_Extend`, etc.

The only thing we need to do is to add modules to let the previous CPU support pipeline, forwarding, hazard detection, and IF flush.

1.1 Pipeline Register

In the architecture, there are 5 stages so we need 4 pipeline registers between stages and stages. These registers are intended to isolate the wire while saving the values between stages. In single-cycle CPU, all elements are handling totally the same instruction, while in pipeline CPU, we must assure the essential data are properly saved in elements of each stage respectively. All of these registers are similar, the registers inside these registers hold the values for the current cycle, and isolate from the previous stage. At the rising edge the registers update the values of registers inside using non-blocking assignment. All of them have almost the same input ports and output ports for this reason.

1.2 Forwarding

If in EX stage there are register values that haven't been write back, forwarding is needed. The forwarding unit detects whether forwarding is needed, and give proper signal to `MUX`, which selects whether the source of `ALU` is from the previous stages or from the future stages.

1.3 Hazard detection

Still some cases cannot be resolved simply by forwarding, such as a arimetic operation just after the load instruction. The hazard detection unit detects whether the instruction order triggers data hazard; if so, it stalls the pipeline by sending a `nop` to the pipeline register, and not letting PC to be updated.

1.4 IF Flush

A `beq` or `jump` instruction can only be acknowledged at ID stage, but at this moment there is a new instruction enters IF stage. Hence, we should abandom the new instruction by sending a `IF Flush` signal to `IF/ID register` to prevent the new instruction propagates further.

2 Modules

2.1 Modules inherited from single-cycle CPU

ALU Acccording to `ALU_Ctrl`, provides the proper operation for `ALU_Src1 and ALU_Src2`.

ALU_Control According th `ALU_Op` and `funct`, gives the proper `ALU_Ctrl`.

Adder Adds the two input.

Control According the instruction, outputs the corresponding control signals.

Data Memory Stores the data. If `MemWrite` is on, the data in the given address is overwritten.

Instruction Memory Stores the instructions.

MUX Selects the proper input to the output given the selecting signal.

PC Indicates the current PC. Only updated at rising edge and when `PCWrite` is on.

Registers Stores the data in the CPU.

Sign_Extend Given a 16-bit input, extends it to a 32-bit value.

2.2 Pipeline Registers

IF_ID Isolates IF stage and ID stage. `inst`(instructions) and `PC` propagates through this register. `flush_i` decides whether the fetched instruction is passed down, and `stall_i` decides whether the new instruction should be passed down to ID stage.

ID_EX Isolates ID stage and EX stage, passing down all of the control signals and registers data and addresses to the EX stage.

EX_MEM Isolates EX stage and MEM stage, passing down the result of EX and control signals that would be used in MEM or WB stage.

MEM_WB Isolates MEM stage and WB stage, passing down the result from EX and MEM stages and control signals that would be used in WB stage.

2.3 Forwarding Unit

Detects whether there is a EX hazard or MEM hazard, and gives the forwarding signal to the `MUX` in front of the `ALU`.

2.4 Hazarding Detection Unit

Detects whether **data hazard** that cannot solved simply by forwarding may happen; if so, send signals to stall the pipeline and prevent `PC` to be updated.

3 Problems and Solutions

Though superficially we only need to add two new modules and pipeline registers (except those which are basically similar to previous modules), datapath changes quite a lot from the previous work, such as control of stall, forwarding, and flush.

3.1 Register initialization

The datapath is a cycle, where the most obvious part is that the `Registers` requires signals both from ID and WB. Without proper initialization of registers, there will be unknown signals interfering the result. Handling **x** of the registers really took a lot of time.

3.2 Stall

Initially I thought **stall** is done by simply send empty control signals forward, until I found the instruction after the stall was overwritten. So I add a datapath to control `PC`.

3.3 Flush

I didn't really realize it until I saw different after `jump` between my output and the reference output.