Computer Architecture Project 1

B04902037 顏子斌
B04902039 羅際禎
B04902041 陳禹樵

Members & Team Work

B04902037　顏子斌 - 33% Reusing old code without verification
B04902039　羅際禎 - 33% Forgetting ;s in every lines
B04902041　陳禹樵 - 33% Getting blamed by bad verilog grammar
TAs　1% Thank you for your hard work（只有1%TA不會森77嗎？）

Overall Implementation of Pipelined CPU

The pipeline CPU is implemented according to the data path given in the homework spec. Larger modules in the data path are constructed individually in its own .v file and referenced in the CPU module, while the smaller ones like “and”, “or” gates are implemented directly with verilog operators in the CPU module. Specific implementation of each modules is elaborated in the following “Implementation of Modules” section.

Implementation of Modules

1. CPU:

The CPU module loads all other modules and connects them together. Most modules’ outputs are left empty, while their inputs are directly connected to the outputs of other modules via direct reference to the targeted module outputs.

2. Pipeline Registers:

The pipeline registers are constructed in four separate modules. In each of these modules, the registers save all of their inputs but still maintain the outputs to be the last stored inputs via the “<=” verilog operator. The outputs are maintained until the next positive clock edge when new inputs arrive at the pipeline registers. Two additional one bit inputs are added to the IFID pipeline register to support the flush and hold operations used in resolving of control hazards and lw data hazards. The pipeline registers are all cleared (in the testbench) before the execution of any instruction.

3. Multiplexers:

There consists three kinds of multiplexers. 2 to 1 5-bit multiplexer, 2 to 1 32-bit multiplexer, and 3 to 1 32-bit multiplexers. Note that Mux3 in the data flow looks too much like Mux8, generating misunderstandings.

4. Control units(main control and ALU control):

In main control unit, we need to decide the value controlling the read/write units, selections for MUXs and branch/jump. The input is opcode segmented from instruction.

In short, there are six scenario for control unit: R-Type, addi, lw, sw, beq and jump. Implement them one by one leading us to finish the implementation of main control.

As for ALU control, we need to tell what ALU should do for each instruction: adding for add, addi, lw, sw; multiplying for mul; abstracting for sub, beq; and and/or for and/or.

5. Hazard Detection Unit:

The hazard detection unit implements the algorithm described in the textbook. An if statement containing all comparisons is located in an always(*) block which sets the control outputs to appropriate values to stall the CPU in case of an lw data hazard.

6. Forwarding Unit:

The forwarding unit implements the algorithm described in the textbook. An always block containing four if statements detects the occurence of data hazards and set the control outputs to multiplexers to forward the appropriate data to the targeted ALU input. The “forwardA” signal connects to “Mux6”, while the “forwardB” signal connects to “Mux7”.

7. Data Memory:

At first we implement a 32byte memory with 32bit per memory slot, but we found that this is not testbench expected. So in second version and after, we implement data memory with 32 one-byte size memory slot in Big-Endian tradition.

The biggest problem we faced when building data memory unit was that the race-condition on read/write control signals. If we put procedures of reading and writing in an always-begin block and used an if-else condition to choose whether to read or write. We would face some unexpected outcomes. Some of them were really weird and hard to understand. They took us lot of time to figure out what was going on and debugged them.

In the end we separate the procedures for reading and writing. We implement the reading part in a straight-forward assignment of output to memory blocks with a ternary operator. And the writing is implemented in an always-begin block triggered by clock.

8. ALU:

A big portion copied from the last homework. Added branch which has nothing to do with the ALU itself.

9. Registers

The register file is implemented with assign statements and an always block. The assign statements propagate the desired register data to the outputs at all times with the exception that if the conditions that a write to register is requested, Rd address is

the same as the Rs or Rt address, and the Rd address is not 5'b0 are all met, the Rd data input is assigned to the output instead of the content in the register file to achieve the effect of data forwarding to ID stage. The always block writes data into the targeted register at positive clock edges whenever a write is requested.

10. PC

The PC module stays mostly the same with the exception that a new input is added to support the lw data hazard stall. The if statement is also changed a little to support the stall.

11. Other operating units

Modules are made for adder, equal, sign_extend and shift_bits respectively with a single line of assign statement doing the job in each of the module; other gate operations including and or are simply implemented in the CPU module with verilog operators.

Problems Encountered and Solution

In this section we describe two major problems that were encountered while working on this project and give the solutions that we took. The problems are stated as follow:

1. Racing Condition in Register and Data Memory

When implementing register and data memory, at first we put the reading of registers and writing of register in the same always block. Nevertheless, it happened that with this implementation, in some cycles, the reading of data just somehow gave incorrect values though the register contents in the registers of the module were correct. After some inspection, we believe that the cause behind this outcome is that when the positive clock edge is detected in the always block, the address of the desired read data had not been presented to the module, thus resulting in the either reading at the incorrect address or reading at address "xxxxxxx". The solution toward this problem is to separate the read data and write data functionality with the read data part implemented using the assign statements to ensure the correct data output at all times, and the write data part implemented using the old fashioned always block that only writes when a rise in clock edge is detected. Moreover, some internal data forwarding is done in the both modules to ensure the correct data output when the read and write addresses just happen to be the same.

2.

Some components are much alike the ones used in the last homework, such like PC and ALU. For these modules, we simply moved the whole module from the folder of

the last homework to the project's. The problem is that we forgot to modify the modules to fit with the others a few times, and it was hard to debug the result. For instance Stall_hold was not connected to PC until the last commit. Luckily, we managed to track out all mistakes.

Q2: Data flows misconception
A2: Draw one yourself

Q3: Strange Memory problem: Writes data to memory at address of memory data
A3: Ta Da~~

Q4: Wrong/Forgotten wire connecting
A4: Backtrace every single pixel of the dataflow.