CA Lab1 Report

b10705005 資工三 陳思如

Module explanation

1. Adder.v

   It takes two 32-bit inputs and outputs the 32-bit sum of these two inputs.

   The adder is used in counting the next program counter. It takes the current program counter 32-bit value and the 32-bit constant of 4. Then it outputs the 32-bit sum of them.

2. Sign extend.v

   It takes in one 12-bit input and sign extends into 32-bit format.

   The front 20 bits is decided by the first bit of the input. If the first bit of the input is `1`, then the front 20 bits will be set as all `1`. Or else, the front 20 bits will be set as all `0`.

3. Control.v

   It takes 7-bit opcode input and outputs three variables. The three outputs are 2-bit ALUop, 1-bit RegWrite and 1-bit ALUsrc.

   - RegWrite: All the 8 instructions need to write to register. Always set to `1`.
   - ALUsrc: `0` for the source from instruction, `1` for the immediate value from instruction.
   - ALUop:`10` for using source from instruction. `11` for using immediate from instruction.

   Classify these two instructions (`addi` and `srai`) from the others with the 7-bit opcode input. `addi` and `srai` have the opcode of `0110011`. Others are with the opcode of `0110011`.

4. MUX32.v

   This takes in two 32-bit input and 1-bit select signal. It outputs the selected 32-bit value.

   If the signal value is `0`, it will choose the first parameter input. Else if the signal value is `1`, it will choose the second parameter input.

5. ALU_Control.v

   It takes 2-bit ALUop and 10-bit function code and outputs a 3-bit code tells the operation type.

   First, I use ALUop to tell which instruction involves the immediate value from the instruction.

   - ALUop: 11 -> `addi` or `srai`
   - ALUop: 10 -> `and` , `xor`, `sll`, `add`, `sub` or `mul`

   Second, I use the 10-bit function code to tell the difference between each instruction. Most of them can be classified by last three bit of the function code. Since there is only 8 instructions in this lab, I classify these instructions in the 3-bit control signal for the ALU in the spec order.

   - ALUop: 11 -> `addi`, `srai`

function [2:0]	000	101
operation	addi	srai
ALU_control	110	111

   - ALUop: 10 -> `and` , `xor`, `sll`, `add`, `sub` or `mul`

function	111	100	001	0000000000	0100000000	0000001000
operation	and	xor	sll	add	sub	mul
ALU_Control	000	001	010	011	100	101

6. ALU.v

   It takes into two 32-bit input and a 3-bit ALUcontrol. It does the operation based on the 3 bit ALU control signal. It outputs the 32-bit result afterwards.

   Here is the signal and what the ALU will do:

ALU Control	000	001	010	011	100	101	110	111
operation	and	xor	sll	add	sub	mul	addi	srai
implement	&	^	<<	+	-	*	+	>>>

7. CPU.v

   We do the data path inorder to pass each state to get the value for next state.

   1. Goes through the **PC module** to get the current PC value.
   2. Passes PC value and the constant 4 in 32-bit format into the **Adder** to get next PC value.
   3. Passes PC value into the **instruction memory module** to get the 32-bit instruction.
   4. Passes the 32-bit instruction into the **Registers module** to get the register source value.
   5. Passes **last 7-bit** instruction into **Control module** to get ALUop, ALUsrc, and regWrite value.
   6. Passes **front 12-bit** instruction into the **Sign-Extended module** to get extended immediate.
   7. Passes the **second register** source value, the **sign extended** value and the **ALUsrc** into **MUX32 module** to get the selected source to input into the ALU state.
   8. Concats instruction [31:25] and instruction [14:12] to get the **10-bit function code**.
   9. Passes the **10-bit functioncode** and the ALUop into the **ALU_Control module**. to get 3-bit ALUCtrl.
   10. Passes the **first register source** , **selected source** and **aluCtrl**into **ALU module**. Then it outputs the **32-bit ALUresult value** based on the instruction operation.

   The above ten steps will be continuously executed until all the instructions are finished. The data is stored and passed into the wire variable so that it can be used in the next state.

Environment

I run the lab in the Windows system with the iverilog IDE. I used the second option to implement my CPU.

```
$cp testcases/instruction_n.txt instruction.txt
$iverilog -o cpu code/src/*.v code/supplied/*.v
$vvp cpu
```