HW4: Single-Cycle CPU in Verilog

Report B06102020 外文五 楊晴雯

Development Environment

OS: Mac OS

IDE: Visual Studio (with Verilog extension and WaveTrace)

Compiler: iverilog

Modules

1. Control

Control reads in opcode and outputs ALUop signal, ALUsrc signal and RegWrite signal. Opcode (7-bit) is the last 7 bits of an instruction and it helps identify the current instruction's format. Control unit processes the input opcode and produces (1) ALUop (2-bit) to specify the operation that ALU needs to carry out; (2) ALUSrc (1-bit) to specify the second source operand of ALU: 0 lets data2 pass and 1 lets immediate pass; (3) RegWrite (1-bit) to specify whether there'sa write-in of register file: 1 indicates True.

Implementation details:

For ALUop, RISC-V's convention is to use 00 for load & store; 01 for branch; 10 for R-type; for others I use 11.

```
module Control(Op_i, ALUOp_o, ALUSrc_o, RegWrite_o);
input [6:0] Op_i; // 7-bit opcode in instruction
output reg [1:0] ALUOp_o; // 2-bit // load or store:
output reg ALUSrc_o; // 1-bit, select = 1-> output im
output reg RegWrite_o; //1-bit, 有rd(dest reg) = 1
```

2. ALU Control

```
module ALU_Control(funct_i, ALUOp_i, ALUCtrl_o);
```

ALU control reads in function code and ALUop and outputs ALU control signal, which is directly sent to ALU for final operation decision. Altogether with Control unit, this CPU implements multi-level (2-level) control on ALU.

Function code (10-bit) is the concatenation of funct7 and funct3 fields in the instruction; ALUop (2-bit) is one of the output signals from Control. ALU Control (3-bit) first decides instruction type, then with function code decides ALUCtrl signal by assigning it to the corresponding number (the following parameters are self-defined).

```
// parameter list for ALUctrl signal assignment
parameter AND = 3'b000;
parameter XOR = 3'b001;
parameter SLL = 3'b010;
parameter ADD = 3'b011;
parameter SUB = 3'b100;
parameter MUL = 3'b101;
parameter ADDI = 3'b110;
parameter SRAI = 3'b111;
```

3. Sign Extend

```
module Sign_Extend(data_i, data_o);
input [11:0] data_i;
output [31:0] data_o;
assign data_o = {{20{data_i[11]}}, data_i}; // extend the MSB
endmodule
```

Sign Extend reads in the 12-bit data in the immediate field (if it were an I-type instruction), and extends it to 32-bit immediate with original value (keeps sign). The module simply copies the MSB and repeats it for 20 times for the upper part of the output value.

4. ALU

```
module ALU(data1_i, data2_i, ALUCtrl_i, data_o, Zero_o);
input signed [31:0] data1_i, data2_i;// 32-bit // remember the sign
input [2:0] ALUCtrl_i; // 3-bit
// put the output into a register
output reg signed [31:0] data_o; //32-bit
output reg Zero_o;
```

ALU reads in data1_i and data2_i as two source operands; ALUCtrl signal to decide source operator. It outputs data_o as the ALU result and Zero_o to indicate whether the result is zero or not: set to 1 if the result is 0 (is used along with opcode to determine branch target and PC value; not in HW4). ALU carries out the calculation required (`AND, XOR, SLL, ADD, SUB, MUL, ADDI, SRAI`) of this CPU with Verilog operators. Note that signed values needed to be stored in signed wires/registers for correct calculation.

5. PC, instruction memory, registers, testbench (TA implemented, skipped)

6. MUX32

```
module MUX32 (data1_i, data2_i, select_i, data_o);
```

MUX reads in register 2's data and the sign-extended immediate value, then if select_i is 0, it outputs the former, otherwise the latter. It is used to decide the second source operand of ALU.

7. Adder

```
module Adder(data1_in, data2_in, data_o);
```

This Adder is a PC adder that adds the original PC value with constant 4 to get a sequential, non-branched next PC value. `data2_in` is set to constant 4.

8. CPU (top module)

CPU module aggregates the above modules and implements the datapath.

CPU reads in `clk` (clock signal), `rst`(reset signal, to set PC to 0) and `start` (start signal, to indicate whether the CPU is running or not). PC starts at value 0, and each time clk is at posedge or rst is at negedge, start is examined and if the CPU has been started, PC value updates (`addr` updates).

`Instr` takes the instruction read from `addr`, and Registers reads in instruction to output register data; Control sends signals to ALUControl; Sign Extend extends the immediate field; MUX decides to let which side pass based on Control's `ALUSrcSig`; ALUControl decides operation based on Control's `ALUOp` and function codes, the result is saved at `wire [31:0] result` and updated in Registers once the `clk` is at posedge. Finally, PC value is updated (`addr <- update_addr`).

Note that although I try writing modules according to its data-dependent order, these steps are NOT executed in sequential order.

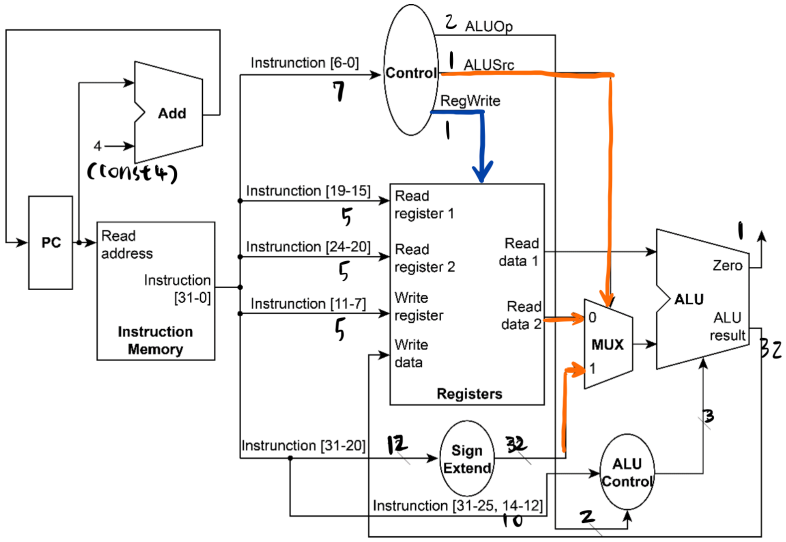

Figure 1 Data path of the CPU in this homework