Computer Architecture Final Project Report

B12901022 廖冠豪

1 Introduction

In this project, I used verilog to implement a simple single-cycle CPU that supports basic RISC-V instructions, such as `sub`, `mul`, `jal`, `auipc`...

2 Design Details

In this section, I will explain how I design the data path for instructions supported by this CPU.

Some convention of the data paths are as follows:

- `instruction[19:15]` is always `rs1`.
- `instruction[24:20]` is always `rs2`.
- `instruction[11:7]` is always `rs2`.
- The opcode, `instruction[6:0]`, func3, `instruction[14:12]` and func7, `instructoin[31:25]` are always used for identifying the instruction and hence correctly decode it.

2.1 R-type instructions

The R-type instructions in this architecture include `add`, `sub`, `mul`, `divu`, and `remu`.
These instructions computes the arithmetic calculation result of the contents of `rs1` and `rs2` and store it to `rd`.
Therefore, in the datapaths for R-type instructions, after the instruction is decoded, the values stored in `rs1` and `rs2` are fed into the ALU as inputs, and the ALU outputs the result of the corresponding operation. At the end of these instructions, since no branching occur, PC is incremented by 4.

Of the instructions mentioned above, `mul`, `divu`, and `remu` are implemented with the ALU in HW2, and are multiple-cycle. Hence they have to handled with some special care, the related details will be in the next section.

2.2 I-type instructions

The I-type instructions in this architecture include `addi`, `slli`, `srli`, `srai`, `slti`, `jalr`, and `lw`.
These instructions computes the arithmetic calculation result of the content of `rs1` and some immediate encoded in the instruction. This 12-bit immediate is stored in the last 12 bits of the instruction, hence it can be extracted as `instruction[31:20]`.
In the data path of I-type instructions, the value stored in `rs1` and the immediate is fed into the ALU as inputs.
After obtaining the calculation result at the ALU, different instructions have different behaviour in the write back stage.
For the `lw` instruction, the result at the ALU is the address in memory to be written to, hence it is fed into the `addr_D` port, which is the address in data memory. Also, the content of `rd2` register is fed into the port `wdata_D`, which is the data written into data memory.
For the `jalr` instruction, the result at the ALU is the address of the destination of branching, hence it is assigned to the next PC value, and the current PC plus for is stored into `rd`.
For the rest of the instructions, there is no branching, so the result of the ALU is stored into `rd`, while PC is incremented by 4.

2.3 S-type instructions

The only S-type instruction in this architecture is `sw`
For the `sw` instruction, the ALU calculates the address in the memory that should be written. The base address is stored in `rs1`. Hence the ALU calculates the sum of the value stored in `rs1` and the immediate, the immediate can be extracted as `{instruction[31:25], instruction[11:7]}`.
The address is fed into the port `addr_D`. Also, the content of `rs2` is fed into the port `wdata_D`.
Lastly, there is no branching, so PC is incremented by 4.

2.4 B-type instructions

B-type instructions supported by this architecture are `beq`, `bne`, `bge`, and `blt`.
All these instruction calculates some relation between the contents of `rs1` and `rs2`, namely equal to, not equal to, not less than, and less than, and decide whether to branch to an offset address based on the result.
Hence, the ALU takes the content of `rs1` and `rs2` as inputs and performs the comparison. If the result is true, than the lowest bit of the result is set to 1, otherwise it is set to 0.
Also, an immediate representing the address offset is fetched from the instruction, this immediate can be extracted as `{intruction[31], intruction[7], instruction[30:25], instruction[11:8],`

`1'b0}`, and a adder is used to calculate the summation of this immediate (sign extended) and the current PC.
Lastly, if the lowest bit of the ALU output is 1, than PC is assigned to this value, if not, PC is simply incremented by 4 (i.e. no branching)

2.5 U-type instructions

The U-type instructions in this architecture are `auipc` and `lui`
U-type instructions are similar to I-type instructions in some senses, but it allows the instruction to contain a larger immediate.
For both the `aupic` and `lui` instructions, the 32-bit immediate can be extracted from the instruction as `{instruction[31:12], 12'b0}`.
For `lui`, this value is directed stored into the `rd` register.
For `auipc` an adder is used to calculate the summation of this value and the current PC, and the summation value is stored into `rd`.
For both of these instructions, there is no branching, hence PC is simply incremented by 4.

2.6 J-type instructions

The only J-type instruction supported by this `jal`
For the `jal` instruction, a 21-bit immediate is decoded from the instruction as `{instruction[31], instruction[19:12], instruction[20], instruction[30:21], 1'b0}`. This number is the offset in address to branch to. Therefore, we use an adder to calculate the summation of the current PC and this immediate (sign extended), and assign it to the next PC.

3 Multi-Cycle Instructions

In this CPU, the `mul`, `divu`, `remu` instructions are implemented using the ALU in HW2, and are hence multi-cycle.
In this section I will explain how they are specially handled.

Since these processes are multi-cycle, we can not simply always go forward to the next stage of execution, hence I used a finite state machine to keep track of the progress and to avoid errors.

```
module FSM(clk, rst_n, next, out);
  input clk, rst_n, next;
  output [4:0] out;
  localparam ID = 5'd0;
  localparam MULDIV_WAIT = 5'd7;
  localparam IF = 5'd1;
  localparam EX = 5'd2;
  localparam MEM = 5'd3;
  localparam MEM_WAIT = 5'd4;
  localparam WB = 5'd5;
  localparam PC = 5'd6;
```

(As can be seen from the design of the module, there are some redundant states, as opposed to the typical 5-stage design, this is due to my incompetence in logic design, and the somewhat mysterious blackboxes regfile and memory)
The `next` signal of the FSM tells it to go to the next stage of the process, or to stay at the same stage. For stages other than EX (the stage where ALU performs computation), `next` is always 1, and each stage only takes one clock cycle. However, for `mul`, `divu`, `remu` and instructions, during the EX stage, the `next` signal is controlled by the ready signal of the Multiplier/Divider, only when the Multiplier/Divider complete all the operations (32 cycles) will the CPU take the output as the result for the ALU, and proceed with the process.
By doing so, we can avoid errors and other unpredictable behaviour, and ensure that our CPU operates correctly.

4 Register Table with Design Compiler

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
state_reg	Flip-flop	5	Y	N	Y	N	N	N	N

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
regwrite_reg	Flip-flop	1	N	N	Y	N	N	N	N
PC_reg	Flip-flop	31	Y	N	Y	N	N	N	N
PC_reg	Flip-flop	1	N	N	N	Y	N	N	N
next_reg	Flip-flop	1	N	N	N	Y	N	N	N
wen_d_reg	Flip-flop	1	N	N	Y	N	N	N	N
instruction_reg	Flip-flop	32	Y	N	N	N	N	N	N
muldiv_valid_reg	Flip-flop	1	N	N	N	N	N	N	N
wb_data_reg	Flip-flop	32	Y	N	N	N	N	N	N
alu_result_reg	Flip-flop	32	Y	N	N	N	N	N	N

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
mem_reg	Flip-flop	995	Y	N	Y	N	N	N	N
mem_reg	Flip-flop	29	Y	N	N	Y	N	N	N

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
over_reg	Flip-flop	1	N	N	N	N	N	N	N
div_reg	Flip-flop	32	Y	N	Y	N	N	N	N
count_reg	Flip-flop	6	Y	N	Y	N	N	N	N
ready_reg	Flip-flop	1	N	N	Y	N	N	N	N
rem_reg	Flip-flop	64	Y	N	Y	N	N	N	N

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
tmp_B_reg	Flip-flop	32	Y	N	N	N	N	N	N
ready_reg	Flip-flop	1	N	N	Y	N	N	N	N
count_reg	Flip-flop	6	Y	N	Y	N	N	N	N
out_reg	Flip-flop	32	Y	N	Y	N	N	N	N
out_reg	Flip-flop	32	Y	N	N	N	N	N	N

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
ready_reg	Flip-flop	1	N	N	Y	N	N	N	N
mul_on_reg	Flip-flop	1	N	N	N	N	N	N	N
div_on_reg	Flip-flop	1	N	N	N	N	N	N	N
out_data_reg	Flip-flop	64	Y	N	Y	N	N	N	N

5 Observations

Here are some observations I've had throughout the course of this final project

- Although there seem to be a wide range of instructions, the operations within the CPU are very similar, and hence many features can be implemented by utilizing the given hardware.
- The design pattern behind different instructions have a lot in common, and it makes implementing a set of instructions a lot easier.
- Multi-cycle operations demands much more care in design and implementation than single-cycle operations.
- The design pattern of abstraction (mulDiv in this case) can save a lot of labor since it allows constructing complex hardware with more fundamental building blocks.
- I don't get why for B-type and J-type instructions, the immediates aren't given in order when encoded in the instruction