CA Homework#3 report

b07902076
資工三 許世儒

ALU

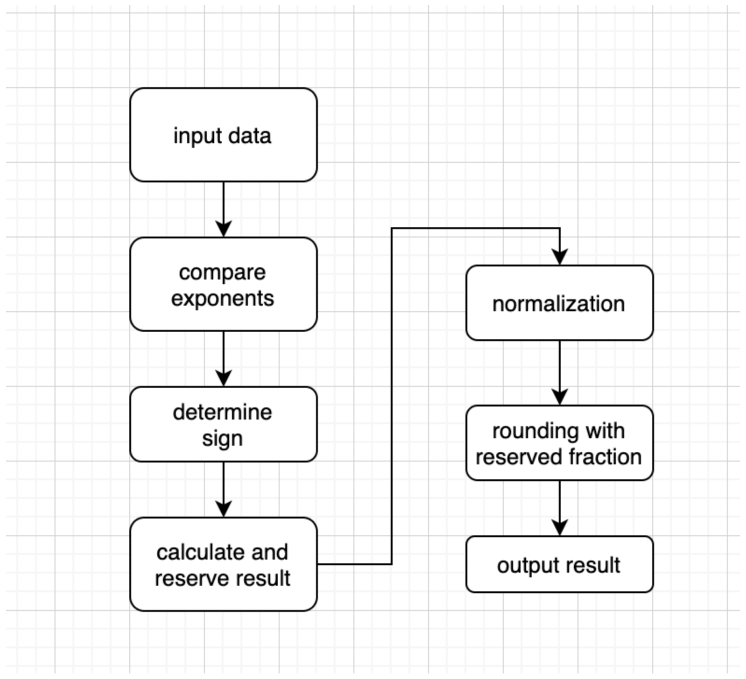
The structure is basicly in four parts. First part is **wires and registers**.
I simply define some necessary regs and wires as TA demonstrated in the video.
Note that I used another signed reg data_ext to deal with multiplication tasks. The second part is **continuous assignment**. I assigned some reg with r type (register) data to output and wires to input data a and b. The third part is **combinational part**. In this part, I first checked whether $i\_valid = 1$ then determined the instruction.
As for signed addition and substraction instruction are as follows.

Signed Addition				Signed Substraction			
a	b	result	overflow	a	b	result	overflow
+	+	+	no	+	+	*	no
+	+	-	yes	+	-	+	no
+	-	*	no	+	-	-	yes
-	+	*	no	-	+	+	yes
-	-	+	yes	-	+	-	no
-	-	-	no	-	-	*	no

And signed multiplication and unsigned multiplication, I checked whether the extension part are all 0(signed and unsigned) or 1(only signed). If it did not, then the result of overflow would be set.
The rest of instructions were straightforward, I simply used relative operators.
The last part is **sequential part**, the registers would be updated to the value of wires during the positive edge of clock and reset during the negative edge of reset.

FPU

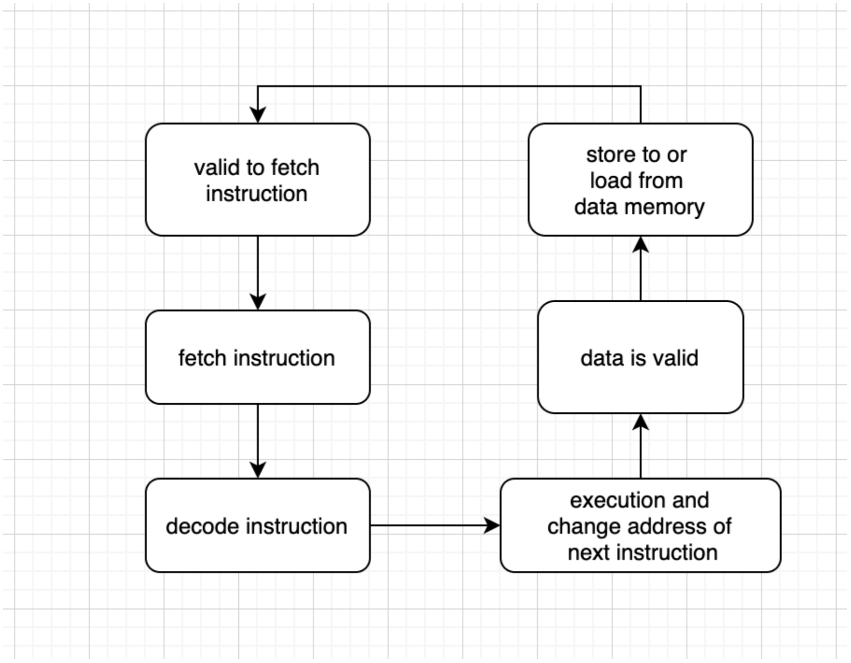
The structure is similar to ALU. The main difference is in the **combinational part**. There are two instructions to operate. The diagram roughly illustrates how the adder works.



There are some more details about implementation. I use while-loop to normalize the result. I create another register to record the omitted bits after shifting so that I could perform rounding correctly. Shift operation is used to reserve the omitted bits. For multiplicator, it's needless to compare exponents and the sign of result is $sign(a)$ exclusive or $sign(b)$. And the rest operations are similar to adder.

CPU

In this problem, I added another part in my program that was **initialization part**. In the first part, I defined 32 64-bit registers, current state and next state which were slightly different from the previous problems. In the initialization part, I initialized the registers in CPU and some wires to 0. Also, there was a part of program changed the state in each cycle. When current state is 0 and the instruction is valid, then I would fetch the instruction. After that, the next cycle and the state would be set to 1, and the instruction had been loaded. Therefore, it began to decoded the instruction and changed the signal of wires. Then, it would store or load data from data memory if necessary or the branch would change the address of next instruction before it fetched the next instruction. Finally, the current state would finally became 0 and fetched the next instruction. The following block diagram briefly explains the process.


valid to fetch instruction
fetch instruction
decode instruction
execution and change address of next instruction
data is valid
store to or load from data memory

In addition, in sequential part, I updated the current state to be the next state.