資工三　何翊豪　B09902046　　Computer Architecture HW2 report

Part 1. Implementation of ALU:

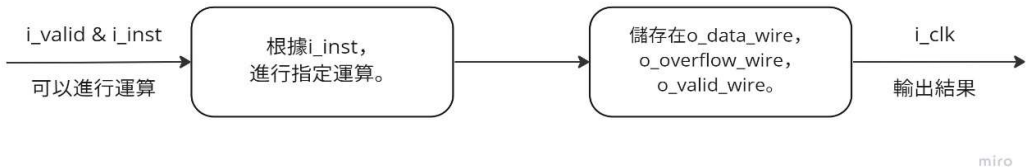
在 combinational part 中，若 i_valid 的 bit 為 1，則會根據 i_inst 所給的值，來去做相對應的運算，並將運算結果存至 o_data_wire，o_overflow_wire 與 o_valid_wire。Sequential part 則會在 i_clk 時，將上述結果送到 output。而各個運算的方式如下：

Signed 運算會使用 wire signed 的 signed_a 與 signed_b，分別取代 i_data_a 與 i_data_b 來進行運算。

1. signed add：o_data_wire = signed_a+signed_b。若正數+正數為負數，或負數+負數為正數，則 overflow，將 o_overflow_wire 設成 1。
2. signed sub：o_data_wire = signed_a-signed_b。若正數-負數為負數，或負數-正數為正數，則 overflow，將 o_overflow_wire 設成 1。
3. signed mul：o_data_wire = signed_a*signed_b。若正數*負數為正數，負數*負數為負數，或正數*正數為負數，則 overflow，將 o_overflow_wire 設成 1。
4. signed max：利用 o_data_wire = (signed_a >= signed_b)? signed_a: signed_b，來取出較大的值。
5. signed min：利用 o_data_wire = (signed_a < signed_b)? signed_a: signed_b，來取出較小的值。
6. signed LT：利用 o_data_wire = (signed_a < signed_b)? 1: 0，來判斷 signed_a 是否小於 signed_b。
7. signed GE：利用 o_data_wire = (signed_a >= signed_b)? 1: 0，來判斷 signed_a 是否大於等於 signed_b。

以下的運算則用原本的 input，分別為 i_data_a 與 i_data_b，來進行運算。

1. unsigned add, sub, mul, max, min：與 signed add, sub, mul, max, min 相似，但會利用 i_data_a 與 i_data_b 來進行運算，除了 unsigned add 與 unsigned mul 為運算結果超過 32bit 為 overflow、unsigned sub 若為小數減大數會 overflow 以外，其餘過程類似。
2. and, or, xor：在 i_data_a 與 i_data_b 上，進行&(and)，|(or)，^(xor)運算，並將結果存到 o_data_wire。
3. bitflip：在 i_data_a 上，進行~(not)運算，並將結果存到 o_data_wire。
4. bitreverse：利用 for 迴圈，將 i_data_a 裡的 bit 依序反過來放到 o_data_wire。



圖一、ALU 從進行運算到輸出結果

Part 2. Implementation of FPU:

Combinational part 與 sequential part 的寫法，與 ALU 是類似的(但 FPU 沒有 overflow bit)，故不再贅述。以下將專注說明 add 與 mul 的實作方式：

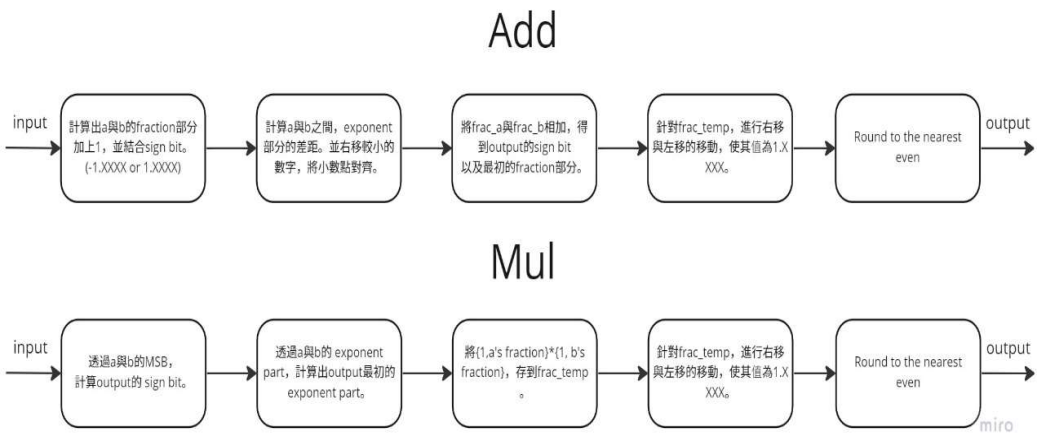
Add

1. 對於 i_data_a 與 i_data_b，先將 1 與其各自的 fraction 部分連接在一起，若其 sign bit 為 1，則再針對該值做 2's complement，以得到$\pm 1.XXXX$ …的結果(分別稱作 frac_a 與 frac_b，為 frac_x 部分)。
2. 計算 i_data_a 與 i_data_b 之間，exponent 部分的差距(稱作 diff)。並讓 exponent 部分較小的一方，對其 frac_x 部分右移 diff 個 bit(signed right shift)。並把 output 最初的 exponent 部分，設為 i_data_a 與 i_data_b 中，較大的 exponent(稱作 out_exp)。
3. 將 frac_a 與 frac_b 相加，並根據結果，提出 output 的 signbit，以及{1,output 最初的 fraction 部分(稱作 frac_out)}，合稱 frac_temp。
4. 針對 frac_temp，若 frac_temp 大於 2，則持續右移 1 個 bit，並讓 out_exp +1；若 frac_temp 小於 1，則持續左移 1 個 bit，並讓 out_exp-1。
5. 最終，做 round to the nearest even，若符合要進位的條件，就讓 frac_out +1。

Mul

1. 先透過 i_data_a 與 i_data_b 的 MSB，來算出 output 的 sign bit。
2. 再透過將 i_data_a 與 i_data_b 的 exponent part 相加再減 127，算出 output 初步的 exponent part(稱作 out_exp)。
3. 將 1 與 i_data_a 的 fraction part 相連在一起，乘上 1 與 i_data_b 的 fraction part 相連在一起的結果，存到 frac_temp (frac_temp = {1, frac_out})。
4. 針對 frac_temp，若 frac_temp 大於 2，則持續右移 1 個 bit，並讓 out_exp +1；若 frac_temp 小於 1，則持續左移 1 個 bit，並讓 out_exp-1。
5. 最終，做 round to the nearest even，若符合要進位的條件，就讓 frac_out +1。

以下為 Add 與 Mul 的概略流程圖，詳請步驟於上述：



圖二、FPU 中 ADD 與 MUL 的流程圖

Part 3. Implementation of CPU:

實作 CPU 時，instruction memory.v 與 data memory.v 已經有提供，因此只需要實作 CPU 的其餘部分。由於一次將所有元件寫進 cpu.v 太為浩大，因此，我將剩下部分分成 pc.v、multiplex.v、adder_4orsum.v、immgen.v、alu.v、regist.v 以及 cpu.v 等 7 大部分，以下將一一講述各部分的工作內容：

pc.v

pc.v 為 CPU 中的 program counter，其中會有一個 output 為 is_validtime，若 is_validtime 為 1 時，才能去 instruction memory 抓取指令來執行。而 program counter 裡，會有 ticktime 來記錄 i_clk 過了幾次，透過 ticktime 來控制 is_validtime，就能確保 CPU stall 來等待 memory 做事。

multiplex.v

本 CPU 中共有 3 個 multiplexer，第一個為選擇線路回流到 PC (PCmul)，第二個為選擇線路流到 ALU (ALUmul)，第三個為選擇線路回流到 register 的 writedata (DATAmul)。而這三個 multiplexer 都會透過其中一個 input 叫做 choosea0orb1，來決定 output 要是 data_a 或 data_b。

adder_4orsum.v

本 CPU 中有 2 個 adder，第一個為讓 PC+4 的 adder (adder4)，第二個為讓 PC + target address 的 adder(addersum)。兩個 adder 都是把 data_a 跟 data_b 餵進去當 input，並 output 出兩者相加的結果。

immgen.v

immgen.v 代表了 CPU 中的 immediate generator，其會根據目前的 instruction，來從 32bit 的 instruction 中，得到 immediate 的值，並拓展成 64bit。主要可分為 load、store 與 bne 及 beq 等三類。以上三類都可以根據 RISC-V instruction 的格式表，抓取正確的 instruction bits 變成 immediate 的值，再補 0 變成 64bits。其中，bne 與 beq 在製造 immediate 的值時，會從 index 1 開始填入 instruction bits，因此之後在餵給 adder 時，不用再 shift left 1 個 bit。

alu.v

alu.v 內含了本 CPU 中的 ALU，其運作原理與先前在 PART 1 所製作的 ALU 是差不多的，故不贅述。本 ALU 也會根據目前的 instruction，而執行不同的運算，運算的方式也與先前製作的 ALU 相同。但本 ALU 會多輸出一個 isALUzero 的 bit，用來幫助在 bne 與 beq 指令時，協助本 CPU 來決定要不要 branch。

regist.v

regist.v 為本 CPU 中的 register，有一塊 Myregister 的 register 負責儲存 register 的資料。當需要存 register 讀資料時，會有 read_reg1 與 read_reg2 兩個 index 作為 input，再將 Myregister[index] 存取的資料作為 output 輸出出去。且當 writedatasignal 為 1 時，代表要將 writedata 寫入，此時就會把 writedata 寫進 Myregister[write_reg]。

cpu.v

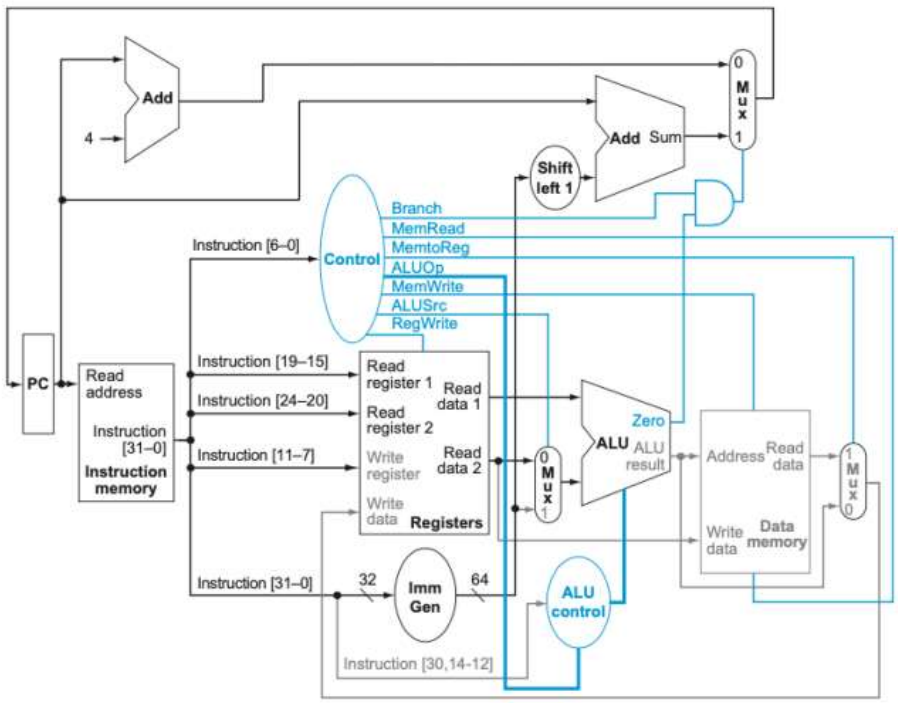
cpu.v 主要負責串接本 CPU 中所有元件以及控制元件運作所需要的信號等兩大功能。串接元件主要是透過引入其他 module 再透過 wire 將這些元件結合起來，做成一個完整的 CPU。

控制信號的部分，包含了 Memread、Memwrite、Regwrite、3 個控制 multiplexer 的訊號、ALUzero 等，由於我直接把 instruction 餵給 ALU 當 input，因此不需要 ALU control。

Memread 與 Memwrite 會依據目前的 instruction，來判斷若是 load 指令則讓 Memread = 1，若為 store 指令則讓 Memwrite = 1。若目前指令為 R-type 以及 load 的時候，Regwrite = 1。ALUzero 則是在 ALU 算出來若 input_a = input_b 後，會透過 CPU 的接線，送到 multiplexer 來幫助 CPU 判斷要不要 branch。

將 output 送至 PC 的 multiplexer，會根據目前 instruction 是否為 bne 或 beq，以及 ALUzero，來決定信號；將 output 送至 ALU 的 multiplexer，會根據目前的 instruction 若為 immediate 或 load and store，就將 output 設定為從 immgen 送入的 input；將 output 送至 register 的 multiplexer，只有當 instruction 為 load 指令時，會將 multiplexer 的 output 設為從 data memory 進來的 input。

我的 CPU 設計圖大致依照下圖：



圖三、CPU 設計圖 (圖源：HW2 的 SPEC)

而 Control Unit 的部分，我已經將其包含在 cpu.v 裡面來控制(依上述)，故沒有額外單獨的 control.v 檔案。