CA Homework #3

ID: B07902070 Name: 陳昱妤

ALU

- 使用 xxx_r 及 xxx_w（宣告為 reg），xxx_w 負責取得目前計算出的結果或狀態，xxx_r 可以在每次 clock trigger 時儲存 xxx_w 的資訊，並接上 xxx 輸出。
- 使用 always @(*) ，只要在內的 RHS 值改變，就看 i_valid 決定是否計算。
- 使用 i_inst 決定為哪種計算。
- 有號數加減乘法使用兩個輸入及一個輸出的 sign bit 判斷是否溢位。
- 有號數計算會先將輸入值導入 wire signed_data_x 以利接下來的計算。
- BitFlip 將輸入和 32'hffffffff 做 XOR
- BitReverse 使用 for 迴圈將輸入和輸出的線路反接。
  需在 always 外設計數器 integer i 。
- 其他類型的計算直接使用相符的 operator 實作。

FPU

先將 exponent 和 fraction 另外用 wire 分開，且 fraction 保留前三位為 001 ，代表 1.xxxx 。
分成加法和乘法實作：

- 加法：分成 alignment 、addition 、normalization 、rounding 四個部分，每個部分以 register 記錄資料。
  - alignment: 仍要保留原本的精確度直到做 rounding 。
  - addition: 浮點數計算的 fraction 是絕對值，不能以 signed 的概念做加法：如果原數字為一正一負，比較經過 alignment 的 fraction 後決定誰減誰，否則相加。
  - normalization: 分成三種狀況處理：
    * 變成 10.xxxx
      fraction 右移一位，exponent 加一。
    * 正確
    * 變成 0.xxxx
      計算需左移多少位，並對 exponent 扣除對應數字。
  - rounding: 使用先前記錄的 rounding bit 及 sticky bit 決定是否將 fraction 加一。
    其實 sticky bit 是要將所有 rounding bit 後面的位元 OR ，但只取 rounding 的下一位，目前測資仍皆正確。
- 乘法：分成 exponents addition 、significands multiplication 、normalization 、rounding 、sign determination 五個部分，每個部分以 register 記錄資料。
  - exponents addition: 由於代表的浮點數指數部分是有減掉 bias 的，因此相加兩者的 exponent 後需再減掉一個 bias。
  - significands multiplication: 以無號數相乘，需保留所有數字。
  - normalization: 同加法部分。
  - rounding: 同加法部分。
  - sign determination: 若兩者符號相同為正，否則為負。

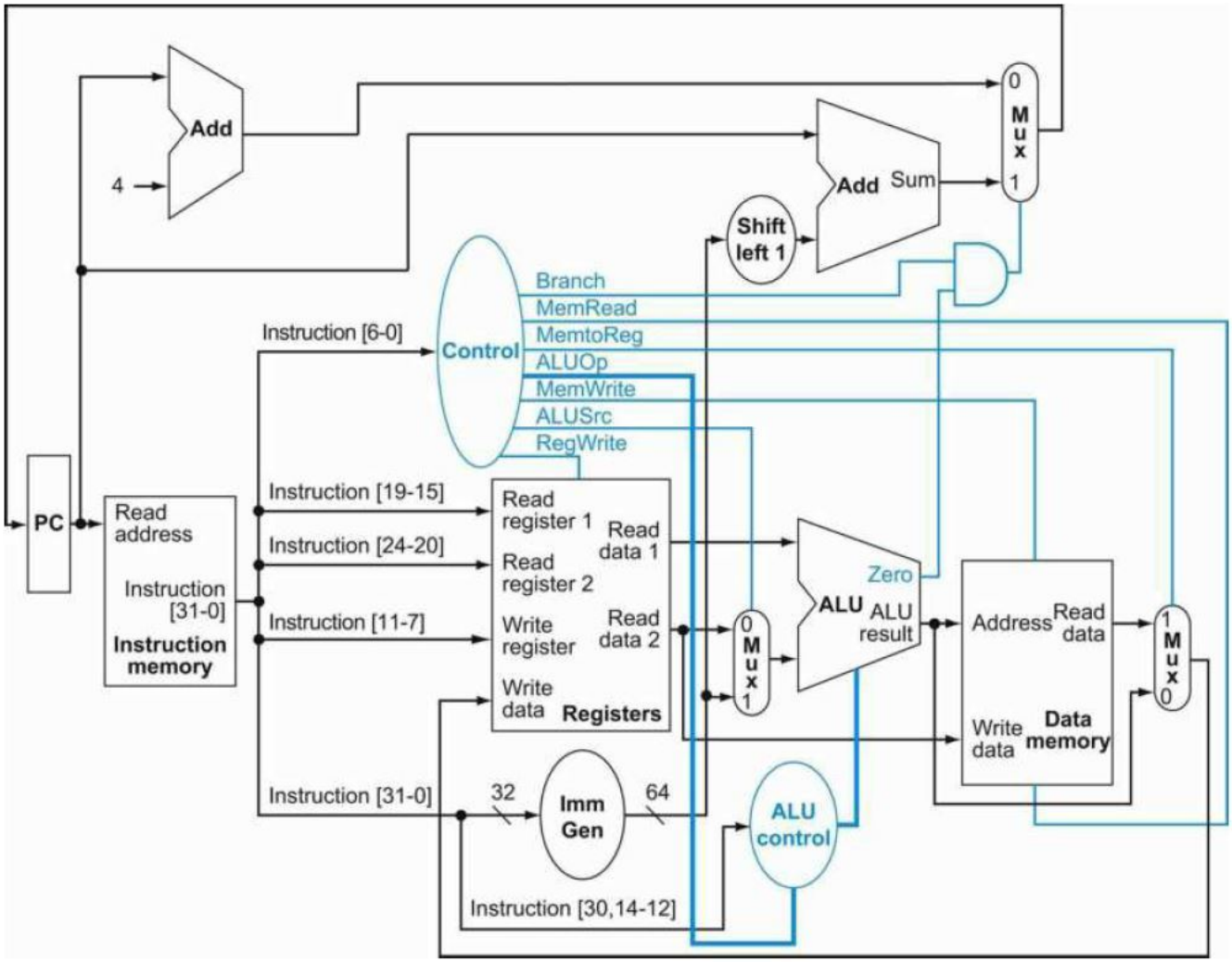
CPU

- 將所有 unit 以呼叫 module 的方式呈現，以 wire 連接它們。
- 部分 unit 間有以 valid wire 確認所需資料有從前一個 unit 傳出。
- 對於所有控制訊號（圖中藍線處），不以 valid wire 做資料傳出的確認。
- module:
  - `program_counter`
    - 設定每 14 cycle 跑一個 instruction （cs range: 0 13）。
    - 初始將 cs 設為 13 ，pc 為 0 。
    - 當 cs 為 12 時，將 pc (reg) 設為新的 instruction address 。
    - 只有當 cs 為 13 時，才讓 `instruction_memory` 可以接收到 pc 的值（`o_i_valid_addr` 為 1）。
  - `controller`
    - 如果接收到 `Stop` 指令，由這裡回傳 `o_finish` 為 1 。
    - 可以只從 `inst[6:4]` 判斷需要送出的控制訊號，
      分別為 `Ld` 、`Sd` 、`Branch` 、`Imm` （需要 immediate 的純運算）、
      `Calc` （不需要 immediate 的純運算）、`Stop`。
    - 另外，用以上分類也可讓 `alu_controller` 運作，
      故值皆以 `inst[6:4]` 傳成 `ctrl_ALUOP` 控制訊號。
    - 由於之後需要辨別是哪一種 branch (`BEQ` or `BNE`) ，
      因此需要 `i_branch` 得知 `inst[7]` 來做判斷。
  - `registers`
    - 如果 `i_i_valid_inst` 為 1 ，就拿到兩個 read register 位置，
      並立刻傳出其內的值，且設 `o_valid_result` 為 1 。
    - 因為 write register 位置可能不會馬上用到，
      須以 sequential circuit 方式留存記錄。
  - `imm_gen`
    - 依照不同指令產生不同組合的 immediate 。
    - 如果 `i_i_valid_inst` 為 1 ，立刻產生對應的 immediate ，
      且設 `o_valid` 為 1 。

- alu_controller
  - 根據 controller 傳來的 ctrl_ALUOP 及 inst[30] 、inst[14:12] 判斷。
  - 因為需要的運算種類只有 7 種，故不採用一般的 ALU control 訊號 (4 bits) ，改為和 inst[9:7] 類似的值（3 bits ；做減法為 3'b010）。
- alu
  - 除了主要的 ALU 外，unit_add_branch 和 unit_add_pc 會一直讓 i_alu_ctrl 為 3'b000 （做加法）。
  - 只使用 valid_data_2 做為運算數字皆已準備好的依據 （通常第二筆資料較晚好）。
  - o_zero 為 1 代表運算結果為 0 。
  - 做完運算後會傳出 o_valid 為 1 。
- mux
  - unit_mux_write_back 的 valid_data_1 ， 如果沒有需要讀 data memory ，會直接讓它為 1 。
  - 做完運算後會傳出 o_valid 為 1 。
- pseudo_and
  - 因為有 BNE ，為運算結果不為 0 時才 branch ， 故需考慮 BNE 、BEQ 和沒有 branch 的狀況。
  - i_branch 值分別為 2'10, 2'01, 2'00 。

- Other:
  o_d_MemRead 及 o_d_MemWrite 皆需考慮主要 ALU 是否計算完，
  故將 controller 傳出的控制訊號和 valid_alu_result 做 AND 運算後才傳出。

- 參考圖示（來源：課本）


Add
4
Add
Sum
Shift
left 1
0
M
u
x
1
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [6-0]
PC
Read
address
Instruction
[31-0]
Instruction
memory
Instruction [19-15]
Instruction [24-20]
Instruction [11-7]
Read
register 1
Read
register 2
Write
register
Write
data
Read
data 1
Read
data 2
Registers
Zero
ALU
ALU
result
Address
Read
data
Write
data
Data
memory
Instruction [31-0]
32
Imm
Gen
64
ALU
control
Instruction [30,14-12]