1. Control & ALU_Control

   相比 Homework 4，這次的 Control 多了從 Hazard Detection Unit 來的 `NoOp_i`，還有接到 Data Memory 相關的幾個 control signal。這次的 Control Unit 在收到 opcode 全部都是 0 (出現在 pipeline flush) 或是`NoOp_i`的時候把所有輸出的訊號都設為 0，否則就根據 opcode 的內容設定相應的 control signal。

   ALU Control 則和上次大同小異，同樣是根據`ALUOp_i`和 `funct_i`來決定`ALUCtrl_o`，不過這次多了判斷 load 和 store instruction 的`funct3`。

2. Pipeline Registers

   我把四組 Pipeline Registers 放在`pipeline_registers` 資料夾下，並從`CPU.v` include 這四個 module。這 4 組 registers 會在程式開始時的 reset 訊號產生時將各自的 register 重設成 0。每次到 clock 的 positive edge 就把 input 用 unblocking 的方式存到對應的 register。

   其中的 IF_ID_REGISTER 比較特別，它還會從 Hazard Detection Unit 讀要不要 Stall 的訊號，以及從判斷要不要 branch 的 AND Gate 讀 Flush 訊號。如果要 Stall 就不會把 register 的內容換成新的，如果要 Flush 就會把 register 歸零。

3. Forwarding Unit

   它會讀 EX_MEM_REGISTER 的 rd 和 ID_EX_REGISTER 的 rs1 或 rs2 是否相同，如果相同就把 EX_MEM_REGISTER 的 `ALUResult` forward 給相對應的 ALU 輸入位置，這樣就可以避免 EX hazard。

   如果沒有 EX hazard，那就用類似的方法判斷有沒有 MEM hazard，並在正確的時機把 WB 階段的`WriteData` forward 給 ALU。

4. Hazard Detection Unit

   在 Hazard_Detection.v。它會讀 EX 階段的 instruction 是否要 MemRead 和 rd，以及 ID 階段的 rs1 和 rs2，如果 EX 需要讀記憶體，而且 rd 和 ID 階段的 rs1 或 rs2 相同時，就輸出 stall 訊號給 IF_ID _REGISTER，來避免 load-use hazard。

5. testbench

   我沒有修改 testbench，因為我把 pipeline register initialization 放在它們各自的 module 裡面。

6. Others

   (a) Imm_Gen

   負責從 instruction 把 immediate 值讀出來進行 sign extension 並傳給 ALU 還有負責 branch instruction 的 Adder。

   (b) defs

   這個檔案裡定義了各種 opcode，以及 ALU control 的 signal。

(c) Misc.

i. MUX32_2

和 Homework 4 的 MUX 一樣，會根據輸入的訊號決定輸出 `data1`或`data2`的其中一個

ii. MUX32_4

這個 MUX 會根據兩個 bit 的訊號，從 4 個輸入中決定要輸出的一個。使用 3 個 MUX32_2 連接而成。

iii. Add32_2

這個 circuit 會讀入兩個數字，並輸出它們的和，不考慮溢位。

iv. Equal32_2

這個 circuit 會讀入兩個數字，並輸出兩數是否相等。

v. SL1_32

這個 circuit 會讀入一個數字，並輸出其 bit shift left 1 的值。

7. CPU

在這裡把所有東西根據 spec 的 datapath 接在一起，並且把 reset 接到各個 pipeline register 上。判斷是否要 flush 的 AND Gate 也在這裡，因為 verilog 的 AND gate 直接用 assign 語法就可以做出來了，所以我沒有放在額外的檔案裡。在 testbench 的 Flush 也是接到這個 AND Gate 上。