學號:B06902136
系級:資工四
姓名:賴冠毓

1. ALU
在 sequential part 處理 output 和 reset。
(clock on 時輸出；reset 在 i_rst_n 結束時把所有 output 歸 0)

```
// sequential circuit
always @(posedge i_clk or negedge i_rst_n) begin
    if (~i_rst_n) begin
        o_data_r <= 0;
        o_overflow_r <= 0;
        o_valid_r <= 0;
    end
    else begin
        o_data_r <= o_data_w;
        o_overflow_r <= o_overflow_w;
        o_valid_r <= o_valid_w;
    end
end
```

combinational part 則處理各個運算。
4’d0: Signed Add
先將 i_data_a 跟 i_data_b 做一般 unsigned 相加，再來判斷 sign bit(正負):
(1)i_data_a 跟 i_data_b 為負，但 o_data_w 為正 => overflow
(2)i_data_a 跟 i_data_b 為正，但 o_data_w 為負 => overflow
(3)其餘情況 => 沒有 overflow

```
// signed add
4'd0 : begin
    o_data_w = i_data_a + i_data_b;
    if (i_data_a[DATA_WIDTH-1] && i_data_b[DATA_WIDTH-1] && ~o_data_w[DATA_WIDTH-1]) begin
        o_overflow_w = 1;
    end
    else if (~i_data_a[DATA_WIDTH-1] && ~i_data_b[DATA_WIDTH-1] && o_data_w[DATA_WIDTH-1]) begin
        o_overflow_w = 1;
    end
    else begin
        o_overflow_w = 0;
    end
    o_valid_w = 1;
end
```

4’d1: Signed Sub
先將 i_data_a 跟 i_data_b 做一般 unsigned 減法，再來判斷 sign bit(正負):
(1)i_data_a 為正，i_data_b 為負，但 o_data_w 為負 => overflow
(2)i_data_a 為負，i_data_b 為正，但 o_data_w 為正 => overflow
(3)其餘情況 => 沒有 overflow

```
// signed sub
4'd1 : begin
    o_data_w = i_data_a - i_data_b;
    if (~i_data_a[DATA_WIDTH-1] && i_data_b[DATA_WIDTH-1] && o_data_w[DATA_WIDTH-1]) begin
        o_overflow_w = 1;
    end
    else if (i_data_a[DATA_WIDTH-1] && ~i_data_b[DATA_WIDTH-1] && ~o_data_w[DATA_WIDTH-1]) begin
        o_overflow_w = 1;
    end
    else begin
        o_overflow_w = 0;
    end
    o_valid_w = 1;
end
```

4’ d2: Signed Mul

先將 i_data_a 跟 i_data_b 做一般 unsigned 相乘，再來判斷 sign bit(正負):

(1)i_data_a 跟 i_data_b 同號，但 o_data_w 為負 => overflow

(2)i_data_a 跟 i_data_b 異號，但 o_data_w 為正 => overflow

(3)其餘情況 => 沒有 overflow

```
// signed mul
4'd2 : begin
    o_data_w = i_data_a * i_data_b;
    if ((i_data_a[DATA_WIDTH-1] == i_data_b[DATA_WIDTH-1]) && o_data_w[DATA_WIDTH-1]) begin
        o_overflow_w = 1;
    end
    else if ((i_data_a[DATA_WIDTH-1] != i_data_b[DATA_WIDTH-1]) && ~o_data_w[DATA_WIDTH-1]) begin
        o_overflow_w = 1;
    end
    else begin
        o_overflow_w = 0;
    end
    o_valid_w = 1;
end
```

4’ d3: Signed Max

先判斷 sign bit(正負)不同的大小:

(1)i_data_a 為正，i_data_b 為負 => 輸出 i_data_a

(2)i_data_a 為負，i_data_b 為正 => 輸出 i_data_b

再比較同樣 sign bit(正負)的大小:

(3)i_data_a 大於 i_data_b => 輸出 i_data_a

(4)其餘情況 => 輸出 i_data_b

```
// signed max
4'd3 : begin
    if (~i_data_a[DATA_WIDTH-1] && i_data_b[DATA_WIDTH-1]) begin
        o_data_w = i_data_a;
    end
    else if (i_data_a[DATA_WIDTH-1] && ~i_data_b[DATA_WIDTH-1]) begin
        o_data_w = i_data_b;
    end
    else if (i_data_a > i_data_b) begin
        o_data_w = i_data_a;
    end
    else begin
        o_data_w = i_data_b;
    end
    o_overflow_w = 0;
    o_valid_w = 1;
end
```

4’d4: Signed Min

先判斷 sign bit(正負)不同的大小:

(1)i_data_a 為正，i_data_b 為負 => 輸出 i_data_b

(2)i_data_a 為負，i_data_b 為正 => 輸出 i_data_a

再比較同樣 sign bit(正負)的大小:

(3)i_data_a 小於 i_data_b => 輸出 i_data_a

(4)其餘情況 => 輸出 i_data_b

```
// signed min
4'd4 : begin
    if (~i_data_a[DATA_WIDTH-1] && i_data_b[DATA_WIDTH-1]) begin
        o_data_w = i_data_b;
    end
    else if (i_data_a[DATA_WIDTH-1] && ~i_data_b[DATA_WIDTH-1]) begin
        o_data_w = i_data_a;
    end
    else if (i_data_a < i_data_b) begin
        o_data_w = i_data_a;
    end
    else begin
        o_data_w = i_data_b;
    end
    o_overflow_w = 0;
    o_valid_w = 1;
end
```

4’d5: Unsigned Add

直接做加法，在數字前多一位數判斷有無多進一位，有則 overflow。

```
// unsugned add
4'd5 : begin
    {o_overflow_w, o_data_w} = i_data_a + i_data_b;
    o_valid_w = 1;
end
```

4’d6: Unsigned Sub

直接做減法，因為是 unsigned 減法，所以一定要大的減小的讓結果為正:

(1)i_data_a 小於 i_data_b，則 o_data_w 為負 => overflow

(2)i_data_a 大於或等於 i_data_b，則 o_data_w 為正 => 沒有 overflow

```
// unsigned sub
4'd6 : begin
    o_data_w = i_data_a - i_data_b;
    if (i_data_a < i_data_b) begin
        o_overflow_w = 1;
    end
    else begin
        o_overflow_w = 0;
    end
    o_valid_w = 1;
end
```

4’d7: Unsigned Mul

因為乘法的商位數最多會到 2 倍 DATA_WIDTH，所以先用一個兩倍位數的 o_mul_temp 存計算結果，然後判斷多出來的位數:

(1)有任何一個位數為 1 => overflow

(2)都為 0 => 沒有 overflow

再取正確位數的 o_mul_temp 給 o_data_w。

```
reg [2*DATA_WIDTH-1:0] o_mul_temp;
```

```
// unsigned mul
4'd7 : begin
    o_mul_temp = i_data_a * i_data_b;
    o_overflow_w = 0;
    for (i = DATA_WIDTH; i < 2 * DATA_WIDTH; i = i + 1) begin
        if (o_mul_temp[i]) begin
            o_overflow_w = 1;
        end
    end
    o_data_w = o_mul_temp[DATA_WIDTH-1:0];
    o_valid_w = 1;
end
```

4’d8: Unsigned Max

直接比大小，輸出比較大的數:

(1)i_data_a 大於 i_data_b => 輸出 i_data_a

(2)其餘情況 => 輸出 i_data_b

```
// unsigned max
4'd8 : begin
    if (i_data_a > i_data_b) begin
        o_data_w = i_data_a;
    end
    else begin
        o_data_w = i_data_b;
    end
    o_overflow_w = 0;
    o_valid_w = 1;
end
```

4’d9: Unsigned Min

直接比大小，輸出比較小的數:

(1)i_data_a 小於 i_data_b => 輸出 i_data_a

(2)其餘情況 => 輸出 i_data_b

```
// unsigned min
4'd9 : begin
    if (i_data_a < i_data_b) begin
        o_data_w = i_data_a;
    end
    else begin
        o_data_w = i_data_b;
    end
    o_overflow_w = 0;
    o_valid_w = 1;
end
```

4’d10: And

直接做 and bit operation。

```
// and
4'd10 : begin
    o_data_w = i_data_a & i_data_b;
    o_overflow_w = 0;
    o_valid_w = 1;
end
```

4’d11: Or

直接做 or bit operation。

```
// or
4'd11 : begin
    o_data_w = i_data_a | i_data_b;
    o_overflow_w = 0;
    o_valid_w = 1;
end
```

4’d12: Xor

$\text{xor}(a,b) = a\bar{b} + \bar{a}b$

```
// xor
4'd12 : begin
    o_data_w = (i_data_a & ~i_data_b) | (~i_data_a & i_data_b);
    o_overflow_w = 0;
    o_valid_w = 1;
end
```

4’ d13: BitFlip

直接做 not bit operation。

```
// bitflip
4'd13 : begin
    o_data_w = ~i_data_a;
    o_overflow_w = 0;
    o_valid_w = 1;
end
```

4’ d14: BitReverse

使用 for 迴圈，與 array 元素 reverse 原理相同。

```
// bitreverse
4'd14 : begin
    for (i = 0; i < DATA_WIDTH; i = i + 1) begin
        o_data_w[i] = i_data_a[DATA_WIDTH-1-i];
    end
    o_overflow_w = 0;
    o_valid_w = 1;
end
```

2. FPU

在 sequential part 處理 output 和 reset。

(clock on 時輸出；reset 在 i_rst_n 結束時把所有 output 歸 0)

```
// sequential circuit
always @(posedge i_clk or negedge i_rst_n) begin
    if (~i_rst_n) begin
        o_data_r <= 0;
        o_valid_r <= 0;
    end
    else begin
        o_data_r <= o_data_w;
        o_valid_r <= o_valid_w;
    end
end
```

combinational part 則處理各個運算。

而在進行指令運算之前，先將 exponent 跟 fraction 分開存方便操作。

為了方便運算，fraction 增加一位數，完整表示個位數字 1。

```
assign int = 1;
// combinational circuit
always @(*) begin
    if(i_valid) begin
        exponent_a_w = i_data_a[30:23];
        exponent_b_w = i_data_b[30:23];
        fraction_a_w = {int, i_data_a[22:0]};
        fraction_b_w = {int, i_data_b[22:0]};
```

1’d0: Add

(1)先將比較小的 exponent 對齊比較大的 exponent，把它的 fraction 向右做 shift。

shift 的同時記錄 R(round bit)跟 S(sticky bit):

R 為最後一個被 shift 掉的 bit(緊跟著剩下來的最後一個 bit)

其餘被 shift 掉的 bit 中如果有 1，則 S 為 1，否則為 0。

```
// add
1'd0 : begin
    // shift number with smaller exponent right
    if (exponent_a_w > exponent_b_w) begin
        S = 0;
        for (i = 0; i < exponent_a_w - exponent_b_w - 1; i = i + 1) begin
            if (fraction_b_w[i] == 1) begin
                S = 1;
            end
        end
        R = fraction_b_w[i];
        fraction_b_w = fraction_b_w >> (exponent_a_w - exponent_b_w);
        exponent_b_w = exponent_a_w;
        o_exponent_w = exponent_a_w;
    end
    else if (exponent_a_w < exponent_b_w) begin
        S = 0;
        for (i = 0; i < exponent_b_w - exponent_a_w - 1; i = i + 1) begin
            if (fraction_a_w[i] == 1) begin
                S = 1;
            end
        end
        R = fraction_a_w[i];
        fraction_a_w = fraction_a_w >> (exponent_b_w - exponent_a_w);
        exponent_a_w = exponent_b_w;
        o_exponent_w = exponent_b_w;
    end
    else begin
        R = 0;
        S = 0;
        o_exponent_w = exponent_b_w;
    end
```

(2)再來做 fraction 數字部分的運算:

因為 fraction 為絕對值，所以先根據 a、b 的 sign bit(正負)，同號做加法，異號做減法。減法的情況再判斷 fraction 大小，決定 output 的 sign bit。

同時用 o_overflow_w 判斷是否需要進位。

```
// add or sub
if (~i_data_a[DATA_WIDTH-1] && ~i_data_b[DATA_WIDTH-1]) begin
    {o_overflow_w, o_fraction_w} = fraction_a_w + fraction_b_w;
    o_sign_w = 0;
end
else if (i_data_a[DATA_WIDTH-1] && i_data_b[DATA_WIDTH-1]) begin
    {o_overflow_w, o_fraction_w} = fraction_a_w + fraction_b_w;
    o_sign_w = 1;
end
else if (~i_data_a[DATA_WIDTH-1] && i_data_b[DATA_WIDTH-1]) begin
    if (fraction_a_w > fraction_b_w) begin
        {o_overflow_w, o_fraction_w} = fraction_a_w - fraction_b_w;
        o_sign_w = 0;
    end
    else begin
        {o_overflow_w, o_fraction_w} = fraction_b_w - fraction_a_w;
        o_sign_w = 1;
    end
end
else begin
    if (fraction_a_w < fraction_b_w) begin
        {o_overflow_w, o_fraction_w} = fraction_b_w - fraction_a_w;
        o_sign_w = 0;
    end
    else begin
        {o_overflow_w, o_fraction_w} = fraction_a_w - fraction_b_w;
        o_sign_w = 1;
    end
end
```

(3)如果 o_overflow_w 為 1，表示需要進位，exponent 加 1。

然後將 fraction 往右 shift 一位，同時更新 R、S。

```
// normalize
if (o_overflow_w) begin
    o_exponent_w = o_exponent_w + 8'b1;
    if (R || S) begin
        S = 1;
    end
    R = o_fraction_w[0];
    o_fraction_w = o_fraction_w >> 1;
end
```

(4)再來做 rounding，有 4 種情況:

① RS = 00，表示結果正確，不需要 rounding。

② RS = 01，捨棄 RS。

③ RS = 11，要進位加 1。

要注意 fraction 為絕對值，所以只有兩個為正 fraction 才加 1，否則 fraction 減 1。

④ RS = 10，要看 guard bit。

如果 guard bit 為 1 進位，否則捨棄。

然後用 o_overflow_w 判斷 rounding 後是否需要再進位。

```
// rounding to the nearest even
if (R == 0 && S == 0) begin
    {o_overflow_w, o_fraction_w} = o_fraction_w;
end
else if (R == 0 && S == 1) begin
    {o_overflow_w, o_fraction_w} = o_fraction_w;
end
else if (R == 1 && S == 1) begin
    if (~i_data_a[DATA_WIDTH-1] && ~i_data_b[DATA_WIDTH-1]) begin
        {o_overflow_w, o_fraction_w} = o_fraction_w + 24'b1;
    end
    else begin
        {o_overflow_w, o_fraction_w} = o_fraction_w - 24'b1;
    end
end
else if (R == 1 && S == 0) begin
    if (o_fraction_w[0]) begin
        if (~i_data_a[DATA_WIDTH-1] && ~i_data_b[DATA_WIDTH-1]) begin
            {o_overflow_w, o_fraction_w} = o_fraction_w + 24'b1;
        end
        else begin
            {o_overflow_w, o_fraction_w} = o_fraction_w - 24'b1;
        end
    end
    else begin
        {o_overflow_w, o_fraction_w} = o_fraction_w;
    end
end
```

(5)如果 o_overflow_w 為 1，表示需要進位，exponent 加 1。

```
// re-normalize
if (o_overflow_w) begin
    o_exponent_w = o_exponent_w + 8'b1;
    o_fraction_w = o_fraction_w >> 1;
end
```

(6)最後將 sign、exponent、fraction 三部分合併。

(fraction 要去掉先前為了方便計算所加上去的個位數，只記錄小數位)

```
// combine
o_data_w = {o_sign_w, o_exponent_w, o_fraction_w[22:0]};
o_valid_w = 1;
```

1’d1: Mul

(1)先將兩個 exponent 相加。

```
// mul
1'd1 : begin
    // add exponents
    o_exponent_w = exponent_a_w + exponent_b_w - 8'd127;
```

(2)因為乘法的商位數最多會到 2 倍 DATA_WIDTH，所以先用一個兩倍位數-1 的 o_mul_temp 存計算結果，同時用 o_overflow_w 判斷是否需要進位

同時記錄 R(round bit)跟 S(sticky bit):

R 為最後一個被 shift 掉的 bit(緊跟著剩下來的最後一個 bit)

其餘被捨棄掉的 bit 中如果有 1，則 S 為 1，否則為 0。

再取正確位數的 o_mul_temp 給 o_data_w。

```
reg [46:0] o_mul_temp;
```

```
// mul
{o_overflow_w, o_mul_temp} = fraction_a_w * fraction_b_w;
S = 0;
for (i = 0; i < 22; i = i + 1) begin
    if (o_mul_temp[i] == 1) begin
        S = 1;
    end
end
R = o_mul_temp[22];
o_fraction_w = o_mul_temp[46:23];
```

(3)判斷 sign bit(正負)，同號為正，異號為負。

```
// sign
if (i_data_a[DATA_WIDTH-1] == i_data_b[DATA_WIDTH-1]) begin
    o_sign_w = 0;
end
else begin
    o_sign_w = 1;
end
```

(4)如果 o_overflow_w 為 1，表示需要進位，exponent 加 1。

然後將 fraction 往右 shift 一位，同時更新 R、S。

```
// normalize
if (o_overflow_w) begin
    o_exponent_w = o_exponent_w + 8'b1;
    if (R || S) begin
        S = 1;
    end
    R = o_fraction_w[0];
    o_fraction_w = o_fraction_w >> 1;
end
```

(5)再來做 rounding，有 4 種情況:

① RS=00，表示結果正確，不需要 rounding。

② RS=01，捨棄 RS。

③ RS=11，要進位加 1。

要注意 fraction 為絕對值，所以只有兩個為正 fraction 才加 1，否則 fraction 減 1。

④ RS=10，要看 guard bit。

如果 guard bit 為 1 進位，否則捨棄。

然後用 o_overflow_w 判斷 rounding 後是否需要再進位。

```
// rounding to the nearest even
if (R == 0 && S == 0) begin
    {o_overflow_w, o_fraction_w} = o_fraction_w;
end
else if (R == 0 && S == 1) begin
    {o_overflow_w, o_fraction_w} = o_fraction_w;
end
else if (R == 1 && S == 1) begin
    if (~i_data_a[DATA_WIDTH-1] && ~i_data_b[DATA_WIDTH-1]) begin
        {o_overflow_w, o_fraction_w} = o_fraction_w + 24'b1;
    end
    else begin
        {o_overflow_w, o_fraction_w} = o_fraction_w - 24'b1;
    end
end
else if (R == 1 && S == 0) begin
    if (o_fraction_w[0]) begin
        if (~i_data_a[DATA_WIDTH-1] && ~i_data_b[DATA_WIDTH-1]) begin
            {o_overflow_w, o_fraction_w} = o_fraction_w + 24'b1;
        end
        else begin
            {o_overflow_w, o_fraction_w} = o_fraction_w - 24'b1;
        end
    end
    else begin
        {o_overflow_w, o_fraction_w} = o_fraction_w;
    end
end
```

(6)如果 o_overflow_w 為 1，表示需要進位，exponent 加 1。

```
// re-normalize
if (o_overflow_w) begin
    o_exponent_w = o_exponent_w + 8'b1;
    o_fraction_w = o_fraction_w >> 1;
end
```

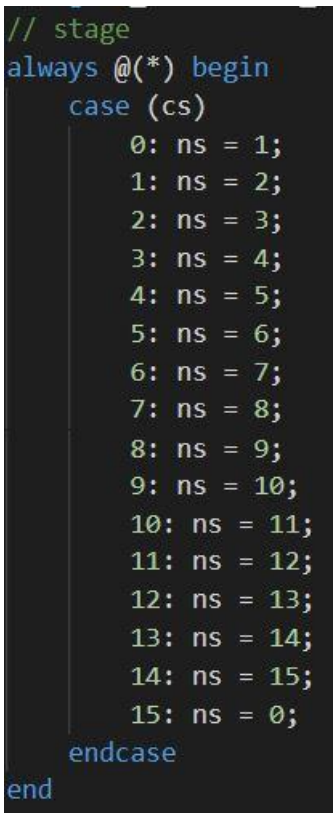
(7)最後將 sign、exponent、fraction 三部分合併。

(fraction 要去掉先前為了方便計算所加上去的個位數，只記錄小數位)

```
// combine
o_data_w = {o_sign_w, o_exponent_w, o_fraction_w[22:0]};
o_valid_w = 1;
```

3. CPU

先將 CPU 分成 16 個 stage。



```
// stage
always @(*) begin
    case (cs)
        0: ns = 1;
        1: ns = 2;
        2: ns = 3;
        3: ns = 4;
        4: ns = 5;
        5: ns = 6;
        6: ns = 7;
        7: ns = 8;
        8: ns = 9;
        9: ns = 10;
        10: ns = 11;
        11: ns = 12;
        12: ns = 13;
        13: ns = 14;
        14: ns = 15;
        15: ns = 0;
    endcase
end
```

然後在 sequential part 處理 output 和 reset，以及 update 所有 register x。

(clock on 時輸出；reset 在 i_rst_n 結束時把所有 output 歸 0，同時將 program counter 跟 pc offset 初始化為 0 和 4)

```
// sequential circuit
always @(posedge i_clk or negedge i_rst_n) begin
    if (~i_rst_n) begin
        o_i_valid_addr_r <= 0;
        o_i_addr_r <= 0;
        o_d_data_r <= 0;
        o_d_addr_r <= 0;
        o_d_MemRead_r <= 0;
        o_d_MemWrite_r <= 0;
        o_finish_r <= 0;
        cs <= 0;
        pc <= 0;
        for (i = 0; i < 32; i = i + 1) begin
            x_r[i] <= 0;
            x_w[i] <= 0;
        end
        pc_offset <= 12'd4;
    end
    else begin
        o_i_valid_addr_r <= o_i_valid_addr_w;
        o_i_addr_r <= o_i_addr_w;
        o_d_data_r <= o_d_data_w;
        o_d_addr_r <= o_d_addr_w;
        o_d_MemRead_r <= o_d_MemRead_w;
        o_d_MemWrite_r <= o_d_MemWrite_w;
        o_finish_r <= o_finish_w;
        cs <= ns;
        for (i = 0; i < 32; i = i + 1) begin
            x_r[i] <= x_w[i];
        end
    end
end
```

combinational part 則處理各個情況，總共有 5 個情況要處理。

(1)cs = 0，表示新的一輪開始:

通知 instruction memory，讓他知道 program counter 是多少。

(一開始為 0，表示從頭開始執行)

```
reg [12:0] pc;
```

同時也初始化 o_d_MemRead_w 跟 o_d_MemWrite_w 為 0，通知 data memory。

最後初始化 o_finish_w 為 0 表示程式還在執行。

```
// start
if (cs == 0) begin
    o_i_valid_addr_w = 1;
    o_i_addr_w = pc;
    o_d_MemRead_w = 0;
    o_d_MemWrite_w = 0;
    o_finish_w = 0;
end
```

(2)再來等到收到 instruction 時，開始處理各個指令操作

```
// deal with instruction
else if (i_i_valid_inst) begin
```

① Stop(eof)

設定 o_finish_w 為 1 表示程式結束。

```
// eof
if (i_i_inst == 32'b11111111111111111111111111111111) begin
    o_finish_w = 1;
end
```

② SD、ADDI

store:

設定 o_d_MemWrite_w 為 1，讓 data memory 知道要進行寫入了。

第 rs1 個 register x 的值加上 immediate 為 address。

第 rs2 個 register x 的值為要儲存的資料。

addi:

第 rs1 個 register x 的值加上 immediate，儲存到第 rd 個 register x。

```
// store
else if (i_i_inst[6:0] == 7'b0100011 && i_i_inst[14:12] == 3'b011) begin
    o_d_MemWrite_w = 1;
    o_d_addr_w = x_w[i_i_inst[19:15]] + {i_i_inst[31:25], i_i_inst[11:7]};
    o_d_data_w = x_w[i_i_inst[24:20]];
end
// addi
else if (i_i_inst[6:0] == 7'b0010011 && i_i_inst[14:12] == 3'b000) begin
    x_w[i_i_inst[11:7]] = x_w[i_i_inst[19:15]] + i_i_inst[31:20];
end
```

③ LD(load)

因為 data memory 要過 7 個 cycle 才會回傳 data，所以先用 rd_index 紀錄要存在第幾個 register x。

設定 o_d_MemRead_w 為 1，讓 data memory 知道要進行讀取了。

第 rs1 個 register x 的值加上 immediate 為 address。

(注意:由於 data memory 不是馬上回傳資料回來，所以這裡只有處理

load address，資料回傳才將第 rd_index 的 register x 做 update，詳細情況請見(5))

```
// load
else if (i_i_inst[6:0] == 7'b0000011 && i_i_inst[14:12] == 3'b011) begin
    rd_index = i_i_inst[11:7];
    o_d_MemRead_w = 1;
    o_d_addr_w = x_w[i_i_inst[19:15]] + i_i_inst[31:20];
end
```

④ ADD、SUB

add:

第 rs1 個 register x 的值加上第 rs2 個 register x 的值，儲存到第 rd 個 register x。

sub:

第 rs1 個 register x 的值減掉第 rs2 個 register x 的值，儲存到第 rd 個 register x。

```
// add
else if (i_i_inst[6:0] == 7'b0110011 && i_i_inst[14:12] == 3'b000 && i_i_inst[31:25] == 7'b0000000) begin
    x_w[i_i_inst[11:7]] = x_w[i_i_inst[19:15]] + x_w[i_i_inst[24:20]];
end
// sub
else if (i_i_inst[6:0] == 7'b0110011 && i_i_inst[14:12] == 3'b000 && i_i_inst[31:25] == 7'b0100000) begin
    x_w[i_i_inst[11:7]] = x_w[i_i_inst[19:15]] - x_w[i_i_inst[24:20]];
end
```

⑤ AND、OR、XOR

and:

第 rs1 個 register x 的值和第 rs2 個 register x 的值做 and bit operation，結果儲存到第 rd 個 register x。

or:

第 rs1 個 register x 的值和第 rs2 個 register x 的值做 or bit operation，結果儲存到第 rd 個 register x。

xor:

第 rs1 個 register x 的值和第 rs2 個 register x 的值做 xor bit operation，結果儲存到第 rd 個 register x。

$\mathrm{xor}(a,b) = a\bar{b} + \bar{a}b$

```
// and
else if (i_i_inst[6:0] == 7'b0110011 && i_i_inst[14:12] == 3'b111 && i_i_inst[31:25] == 7'b0000000) begin
    x_w[i_i_inst[11:7]] = x_w[i_i_inst[19:15]] & x_w[i_i_inst[24:20]];
end
// or
else if (i_i_inst[6:0] == 7'b0110011 && i_i_inst[14:12] == 3'b110 && i_i_inst[31:25] == 7'b0000000) begin
    x_w[i_i_inst[11:7]] = x_w[i_i_inst[19:15]] | x_w[i_i_inst[24:20]];
end
// xor
else if (i_i_inst[6:0] == 7'b0110011 && i_i_inst[14:12] == 3'b100 && i_i_inst[31:25] == 7'b0000000) begin
    x_w[i_i_inst[11:7]] = (x_w[i_i_inst[19:15]] & ~x_w[i_i_inst[24:20]]) | (~x_w[i_i_inst[19:15]] & x_w[i_i_inst[24:20]]);
end
```

⑥ ANDI、ORI、XORI

andi:

第 rs1 個 register x 的值和 immediate 做 and bit operation，結果儲存到第 rd 個 register x。

or:

第 rs1 個 register x 的值和 immediate 做 or bit operation，結果儲存到第 rd 個 register x。

xor:

第 rs1 個 register x 的值和 immediate 做 xor bit operation，結果儲存到第 rd 個 register x。

$\mathrm{xor}(a,b) = a\bar{b} + \bar{a}b$

```
// andi
else if (i_i_inst[6:0] == 7'b0010011 && i_i_inst[14:12] == 3'b111) begin
    x_w[i_i_inst[11:7]] = x_w[i_i_inst[19:15]] & i_i_inst[31:20];
end
// ori
else if (i_i_inst[6:0] == 7'b0010011 && i_i_inst[14:12] == 3'b110) begin
    x_w[i_i_inst[11:7]] = x_w[i_i_inst[19:15]] | i_i_inst[31:20];
end
// xori
else if (i_i_inst[6:0] == 7'b0010011 && i_i_inst[14:12] == 3'b100) begin
    x_w[i_i_inst[11:7]] = (x_w[i_i_inst[19:15]] & ~i_i_inst[31:20]) | (~x_w[i_i_inst[19:15]] & i_i_inst[31:20]);
end
```

⑦ SLLI、SRLI

slli:

第 rs1 個 register x 的值向左 shift shamt 位，結果儲存到第 rd 個 register x。

srli:

第 rs1 個 register x 的值向右 shift shamt 位，結果儲存到第 rd 個 register x。

```
// slli
else if (i_i_inst[6:0] == 7'b0010011 && i_i_inst[14:12] == 3'b001 && i_i_inst[31:25] == 7'b0000000) begin
    x_w[i_i_inst[11:7]] = x_w[i_i_inst[19:15]] << i_i_inst[24:20];
end
// srli
else if (i_i_inst[6:0] == 7'b0010011 && i_i_inst[14:12] == 3'b101 && i_i_inst[31:25] == 7'b0000000) begin
    x_w[i_i_inst[11:7]] = x_w[i_i_inst[19:15]] >> i_i_inst[24:20];
end
```

⑧ BNE、BEQ

bne:

如果第 rs1 個 register x 的值不等於第 rs2 個 register x 的值，program counter 的 offset 變成 immediate。

beq:

如果第 rs1 個 register x 的值等於第 rs2 個 register x 的值，program counter 的 offset 變成 immediate。

(注意:此處 instruction 裡並沒有 immediate[0]，immediate[0]恆等於 0)

```
// bne
else if (i_i_inst[6:0] == 7'b1100011 && i_i_inst[14:12] == 3'b001) begin
    if (x_w[i_i_inst[19:15]] != x_w[i_i_inst[24:20]]) begin
        pc_offset = {i_i_inst[31], i_i_inst[7], i_i_inst[30:25], i_i_inst[11:8], 1'b0};
    end
end
// beq
else if (i_i_inst[6:0] == 7'b1100011 && i_i_inst[14:12] == 3'b000) begin
    if (x_w[i_i_inst[19:15]] == x_w[i_i_inst[24:20]]) begin
        pc_offset = {i_i_inst[31], i_i_inst[7], i_i_inst[30:25], i_i_inst[11:8], 1'b0};
    end
end
```

(3)cs = 15，表示這一輪結束了:

在進到下一輪前處理 program counter

program counter 要加上 offset。

預設的 offset 為 4(表示執行下一行指令)，如果 offset 不為 4 表示這輪有 branch，branch 處理完將 offset 恢復預設 4。

```
// deal with pc
else if (cs == 15) begin
    o_i_valid_addr_w = 0;
    pc = pc + $signed(pc_offset);
    if (pc_offset != 12'd4) begin
        pc_offset = 12'd4;
    end
    o_d_MemRead_w = 0;
    o_d_MemWrite_w = 0;
end
```

(4)其餘 stage 就不做任何事

```
else begin
    o_i_valid_addr_w = 0;
    o_d_MemRead_w = 0;
    o_d_MemWrite_w = 0;
end
```

(5)load 的值回來了，update 第 rd_index 的 register x 的值

```
// receive loaded data
if (i_d_valid_data) begin
    x_w[rd_index] = i_d_data;
end
```