Simple CPU

0416303 楊博凱 資工 08

1 INTRODUCTION

在最後一次 LAB 中，我們需要以 decoder、ALU、register 等模組實作出一個具有簡單計算功能的 CPU。

2 EXPERIMENTAL AND COMPUTATIONAL DETAILS

2.1 Design

本次 LAB 中，decoder 我們用組合電路 case by case examination，即時的更新 ALU module 的控制訊號。在左設計中，先檢驗 instruct[19:16]屬於哪一個 case，依照指令更新 ALU 的控制訊號，例如 ： 當 instruct[19:16]=1010 時，我們就將應該被計算的數字與 XOR 指令傳入 ALU。在右設計中，每次 clock edge 時如果有非閒置的指令進入，便同步更新 register 的儲存值。

```
always@(*) begin
case(instruct[19:16])
    4'b0010: begin
        aluMode = 0;
        aluR1 = register[instruct[15:8]];
        aluR2 = register[instruct[7:0]];
    end
    4'b0011: begin
        aluMode = 0;
        aluR1 = register[instruct[15:8]];
        aluR2 = instruct[7:0];
    end
    4'b0100: begin
        aluMode = 1;
        aluR1 = register[instruct[15:8]];
        aluR2 = register[instruct[7:0]];
    end
    4'b0101: begin
        aluMode = 1;
        aluR1 = register[instruct[15:8]];
        aluR2 = instruct[7:0];
    end
    4'b1110: begin
        aluMode = 2;
        aluR1 = register[instruct[15:8]];
        aluR2 = register[instruct[7:0]];
    end
    4'b1100: begin
        aluMode = 3;
        aluR1 = register[instruct[15:8]];
        aluR2 = register[instruct[7:0]];
    end
    4'b1000: begin
        aluMode = 4;
        aluR1 = register[instruct[15:8]];
        aluR2 = register[instruct[7:0]];
    end
    4'b1010: begin
        aluMode = 5;
        aluR1 = register[instruct[15:8]];
        aluR2 = register[instruct[7:0]];
    end
    default: {aluMode,aluR1,aluR2} <= ~0;
    endcase
end
```

```
always@(posedge clk, posedge reset) begin
    if(reset) begin
        register[0] <= 0;
        register[1] <= 0;
        register[2] <= 0;
        register[3] <= 0;
    end else if(instruct[19:16])
        register[instruct[15:8]]<=aluOut;
    else
        register[instruct[15:8]] <= register[instruct[15:8]];
end
```

加法器與減法器均採用 ripple 設計， 其餘位元運算為手動展開。

```
module AND(A,B,t);
   input [7:0] A, B;
   output [7:0]t;
   assign t[0] = A[0] & B[0];
   assign t[1] = A[1] & B[1];
   assign t[2] = A[2] & B[2];
   assign t[3] = A[3] & B[3];
   assign t[4] = A[4] & B[4];
   assign t[5] = A[5] & B[5];
   assign t[6] = A[6] & B[6];
   assign t[7] = A[7] & B[7];
endmodule

module OR(A,B,t);
   input [7:0] A, B;
   output [7:0]t;
   assign t[0] = A[0] | B[0];
   assign t[1] = A[1] | B[1];
   assign t[2] = A[2] | B[2];
   assign t[3] = A[3] | B[3];
   assign t[4] = A[4] | B[4];
   assign t[5] = A[5] | B[5];
   assign t[6] = A[6] | B[6];
   assign t[7] = A[7] | B[7];
endmodule

module NOT(A,t);
   input [7:0] A;
   output [7:0]t;
   assign t[0] = ~A[0];
   assign t[1] = ~A[1];
   assign t[2] = ~A[2];
   assign t[3] = ~A[3];
   assign t[4] = ~A[4];
   assign t[5] = ~A[5];
   assign t[6] = ~A[6];
   assign t[7] = ~A[7];
endmodule

module XOR(A,B,t);
   input [7:0] A, B;
   output [7:0]t;
   assign t[0] = A[0] ^ B[0];
   assign t[1] = A[1] ^ B[1];
   assign t[2] = A[2] ^ B[2];
   assign t[3] = A[3] ^ B[3];
   assign t[4] = A[4] ^ B[4];
   assign t[5] = A[5] ^ B[5];
   assign t[6] = A[6] ^ B[6];
   assign t[7] = A[7] ^ B[7];
endmodule
```

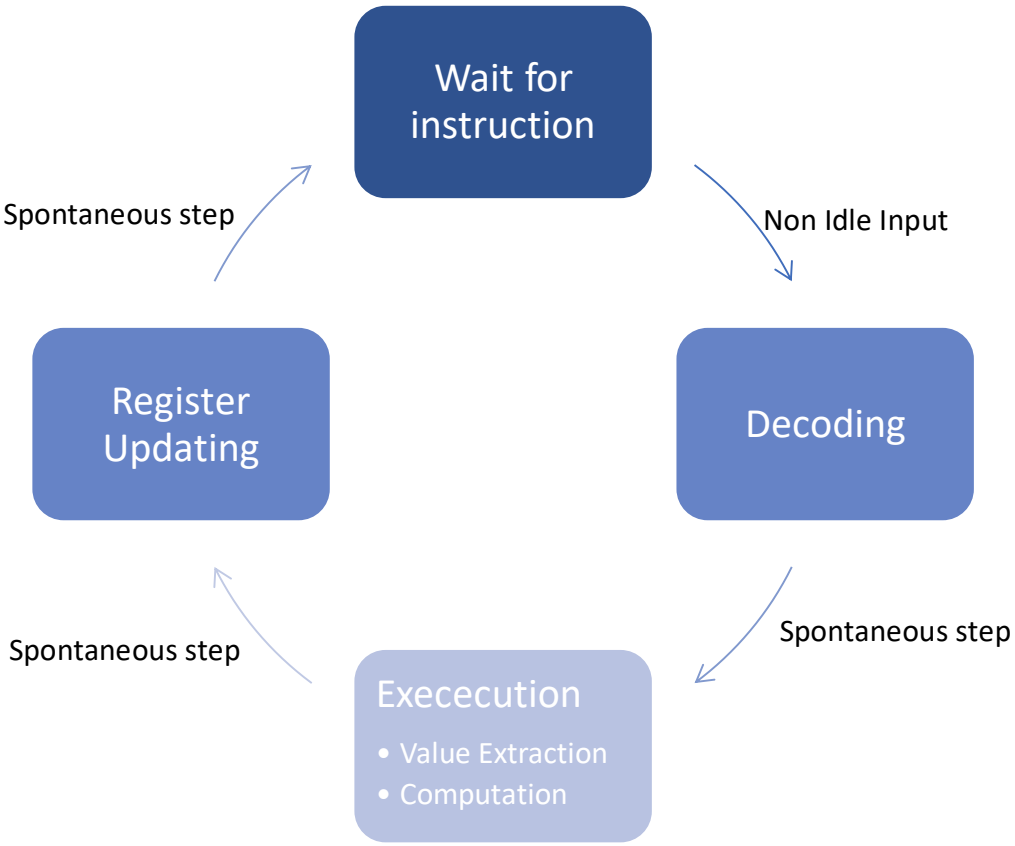
```
module FA(A,B,S);
   input [7:0] A, B;
   output [7:0] S;
   wire [6:0] t;
   wire Cout;
   FA_1bit FA0(.A(A[0]), .B(B[0]), .Cin(0), .S(S[0]), .Cout(t[0]));
   FA_1bit FA1(.A(A[1]), .B(B[1]), .Cin(t[0]), .S(S[1]), .Cout(t[1]));
   FA_1bit FA2(.A(A[2]), .B(B[2]), .Cin(t[1]), .S(S[2]), .Cout(t[2]));
   FA_1bit FA3(.A(A[3]), .B(B[3]), .Cin(t[2]), .S(S[3]), .Cout(t[3]));
   FA_1bit FA4(.A(A[4]), .B(B[4]), .Cin(t[3]), .S(S[4]), .Cout(t[4]));
   FA_1bit FA5(.A(A[5]), .B(B[5]), .Cin(t[4]), .S(S[5]), .Cout(t[5]));
   FA_1bit FA6(.A(A[6]), .B(B[6]), .Cin(t[5]), .S(S[6]), .Cout(t[6]));
   FA_1bit FA7(.A(A[7]), .B(B[7]), .Cin(t[6]), .S(S[7]), .Cout(Cout));
endmodule

module FA_1bit(A, B, Cin, S, Cout);
   input A, B, Cin;
   output S, Cout;
   assign S = Cin ^ A ^ B;
   assign Cout = (A & B) | (Cin & B) | (Cin & A);
endmodule
```

```
module FS(A,B,S);
   input [7:0] A, B;
   output [7:0] S;
   wire [7:0] t;
   wire Cout;
   FS_1bit FS0(.A(A[0]), .B(B[0]), .Cin(0), .S(S[0]), .Cout(t[0]));
   FS_1bit FS1(.A(A[1]), .B(B[1]), .Cin(t[0]), .S(S[1]), .Cout(t[1]));
   FS_1bit FS2(.A(A[2]), .B(B[2]), .Cin(t[1]), .S(S[2]), .Cout(t[2]));
   FS_1bit FS3(.A(A[3]), .B(B[3]), .Cin(t[2]), .S(S[3]), .Cout(t[3]));
   FS_1bit FS4(.A(A[4]), .B(B[4]), .Cin(t[3]), .S(S[4]), .Cout(t[4]));
   FS_1bit FS5(.A(A[5]), .B(B[5]), .Cin(t[4]), .S(S[5]), .Cout(t[5]));
   FS_1bit FS6(.A(A[6]), .B(B[6]), .Cin(t[5]), .S(S[6]), .Cout(t[6]));
   FS_1bit FS7(.A(A[7]), .B(B[7]), .Cin(t[6]), .S(S[7]), .Cout(Cout));
endmodule

module FS_1bit(A, B, Cin, S, Cout);
   input A, B, Cin;
   output S, Cout;
   assign S = Cin ^ A ^ B;
   assign Cout = ((~A) & B) | (Cin & B) | (Cin & (~A));
endmodule
```

```
module ALU(
    input[2:0] mode,
    input[7:0] reg1,
    input[7:0] reg2,
    output reg[7:0] out
);
    // Interface:
    // mode 0 - Add
    // mode 1 - Sub
    // mode 2 - And
    // mode 3 - Or
    // mode 4 - Not
    // mode 5 - Xor

    wire[7:0] addOut;
    FA fa0(reg1,reg2,addOut);

    wire[7:0] subOut;
    FS fs0(reg1,reg2,subOut);

    wire[7:0] andOut;
    AND and0(reg1,reg2,andOut);

    wire[7:0] orOut;
    OR or0(reg1,reg2,orOut);

    wire[7:0] notOut;
    NOT not0(reg1,notOut);

    wire[7:0] xorOut;
    XOR xor0(reg1,reg2,xorOut);

    always@(*)case(mode)
    0: out =addOut;
    1: out =subOut;
    2: out =andOut;
    3: out =orOut;
    4: out =notOut;
    5: out =xorOut;
    default: out =0;
    endcase

endmodule
```

ALU module interface 一開始就即時計算好所有運算的值。並以 3 位元控制訊號控制 mode，再根據不同 mode 進入不同 case，並揀取 ALU 所需傳出的 output。

2.2 Finite State Machine & Event Diagram

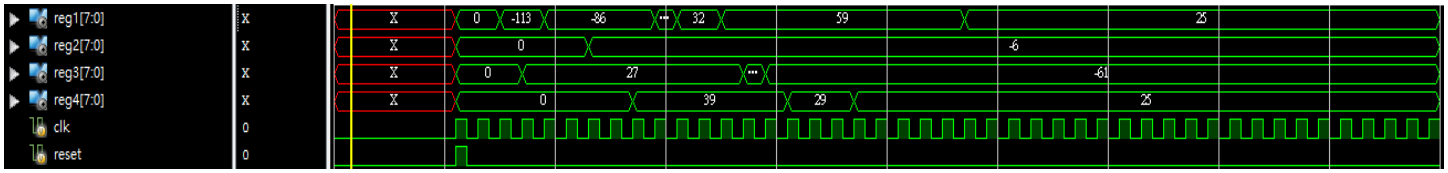


3 RESULTS AND DISCUSSION

3.1 Discussion

在本次 LAB 中，發現到若有切分 pipeline，則會造成運算無法在一個 clock 內完成。為了避免同步電路產生的延遲問題，我們多半採用組合電路以達到 real time response 的效果。我覺得這次作業 CPU 指令全部要求即時不合理，因為 real time 處理常常會犧牲掉處理器的時脈，對 CPU 是一個不利的設計。

3.2 Timing Diagram



4 CONCLUSIONS

By this final project，I realized that a simple CPU is a circuit that has a stream of instructions fed to it and those instructions determine what it will do, based on the operations we give to it. In my code, at first I let the decoder decode the instruction which can select the numbers to be calculated and the mode that the ALU should do with these two numbers. Next, it send the two numbers and mode number to ALU, telling ALU it can starts to calculate. Upon receiving the mode and numbers, ALU calculate all the possible answers to each mode and select the correct output by mode number. At last, we update the answer to our register and continue to go on to the next instruction.