Part III: Simple MIPS Processor with Additional Datapath

control_unit module

ทำการเพิ่ม sll โดย sll อยู่ใน instruction r-type มีเลข opcode เป็น 0 และเลข function เป็น 0 จึงกำหนดให้หาก case opcode เป็น 6'b000000 กำหนดให้เส้น RegDst,Regwrite เป็น 1 นอกนั้นเป็น 0

ทำการเพิ่ม lb โดย lb อยู่ใน instruction i-type มีเลข opcode เป็น 32 บิท จึงกำหนดให้หาก case opcode เป็น 6'b100000 กำหนดให้เส้น ALUsrc,MemtoReg,RegWrite,MemRead,lb เป็น 1 นอกนั้นเป็น 0

```
47      always @( posedge rst or posedge clk) begin
48          if(rst) begin
49              for(i=0;i<32;i=i+1) begin
50                  mem[i] = 0;
51              end
52          end
53          else if(reg_write == 1)
54              if(lb!=1)
55                  mem[write_register] <= write_data;
56              else
57                  case(alu_result)
58                      2'b00 : mem[write_register] <= 8'h00000000 + write_data[7:0];
59                      2'b01 : mem[write_register] <= 8'h00000000 + write_data[15:8];
60                      2'b10 : mem[write_register] <= 8'h00000000 + write_data[23:16];
61                      2'b11 : mem[write_register] <= 8'h00000000 + write_data[31:24];
62                  endcase
63      end
64  endmodule
```

register module

ในการ load byte ทำการเพิ่มเงื่อนไขเช็คว่าเมื่อสาย regwrite เป็น 1 ถ้า lb เป็น 0 ให้ทำการเขียน data ลงใน register ได้เลย แต่ถ้า lb เป็น 1 ให้ทำการ load byte เป็นจำนวน 4 byte โดยเขียน data ทั้งหมดครั้งละ 8 bit จำนวน 4 ครั้งจนครบ 32 bit (1 byte = 8 bit)

Part IV: Simple MIPS Processor with Subroutine

control_unit module

ทำการเพิ่ม jr โดย jr อยู่ใน instruction r-type มีเลข opcode เป็น 0 และเลข function เป็น 8 จึงกำหนดให้หาก case opcode เป็น 6'b000000 และ func เป็น 6'b001000 กำหนดให้เส้น jr เป็น 1 นอกนั้นเป็น 0

ทำการเพิ่ม jal โดย jal อยู่ใน instruction j-type มี opcode เป็น 3 จึงกำหนดให้หาก case opcode เป็น 6'b000011 กำหนดให้เส้น Regwrite,Jump,Jal เป็น 1 นอกนั้นเป็น 0

ทำการเพิ่ม j โดย j อยู่ใน instruction j-type มี opcode เป็น 2 จึงกำหนดให้หาก case opcode เป็น 6'b000010 กำหนดให้เส้น Jump เป็น 1 นอกนั้นเป็น 0

top_module module

```
77          .read_register_1(instruction[25:21]),
78          .read_register_2(instruction[20:16]),
79          .write_register(RegDst ? instruction[15:11] : Jal ? 5'b11111 : instruction[20:16]),
80          .write_data(write_data),
81          .lb(LoadByte),
```

ในการ Jump Register ทำการกำหนดสาย write_register ขึ้นมาโดย RegDst คือ instruction 15:11(rd) หรือ กำหนดตัวที่ jump คือ register ตัวที่ 31 (5'b11111)

```
114     wire [31:0] tmp_new_address;
115     assign tmp_new_address = current_address + 4;
116     assign new_address = Jump ? {tmp_new_address[31:28], (instruction[25:0] << 2)} :
117         ((Branch && zero) ? tmp_new_address + (imm_signed_extended << 2) : Jr ? read_data_1 : tmp_new_address);
118
119     /* END */
120
121     wire [31:0] data_from_mem;
122     assign write_data = MemtoReg ? data_from_mem : Jal ? tmp_new_address : alu_result;
123
```

ทำการเพิ่มสาย tmp_new_address โดยหลักการเหมือนเป็นตัว PC counter คือการนำ address ปัจจุบันบวก 4 เพื่อไปยัง address ที่ต้องการ jump ไป และทำการกำหนด new address ถ้าหาก Jump ให้ instruction 25:0 shift left ไป 2 และ PC counter จะบวกเพิ่มไป 4 ไปเก็บไว้ใน tmp_new_address ที่เป็น address ที่ต้องการ jump ไป ถ้าหากเป็น Beq จะนำ tmp_new_address ไปรวมกับ instruction 15:0 (เลข immediate)ที่ shift left ไป 2 เป็น new address ถ้าหากเป็น Jr ให้สาย read_data_1 เป็น new address

ทำการเพิ่มสาย data_from_mem คือ data ที่มาจาก memory โดยจะกลับมาเขียนที่ registers state โดยกำหนด write_data หากเมื่อ Memtoreg เปลี่ยน mux เป็น 1 ให้เขียน data_from_mem หากเป็น Jal ให้เขียน tmp_new_address นอกเหนือจากนั้นเขียน alu_result