แก้ไขครั้งล่าสุดเมื่อวันที่ 3 ตุลาคม 2568

Chapter 8

Stack CPU

สแตกซีพียู ประกอบด้วย 3 ส่วนหลักคือ

- Memory (MEM)
- Data Stack (DS)
- Return Stack (RS)

สแตกซีพียู จะทำงานตามคำสั่งที่บรรจุไว้ในหน่วยความจำ (ช่องละ 8 บิตหรือ 1 ไบต์) โดยเริ่มจากคำสั่งที่ Program Counter (PC) ชี้อยู่ เช่น PC = 0 คือเริ่มทำคำสั่งที่อยู่ที่ช่องที่ 0 คำสั่งหนึ่งจะประกอบด้วย 2 ส่วนคือ

- รหัสดำเนินการ (opcode) ขนาด 1 ไบต์ บอกว่าเป็นคำสั่งอะไร ไบต์ถัดจากรหัสดำเนินการจะเป็นตัวถูกดำเนินการ (operand) ถ้ามี
- ตัวถูกดำเนินการ (operand) ขนาด 1 ไบต์ คำสั่งหนึ่งอาจจะต้องใช้ตัวถูกดำเนินการ (operand) จำนวน 1 หรือ 2 ตัว หรือไม่มีตัวถูกดำเนินการเลยก็ได้

เมื่อทำงานจบหนึ่งคำสั่ง PC จะเลื่อนไปชี้ที่คำสั่งถัดไป และเริ่มทำ (execute) คำสั่งนั้น เป็นเช่นนี้ไปเรื่อย ๆ

สถาปัตยกรรมชุดคำสั่ง (Instruction Set Architecture)

ชุดคำสั่งทั้งหมดมีดังนี้ (oper หรือ operand คือตัวถูกดำเนินการ, รหัสดำเนินการในตารางเป็นเลขฐานสิบหก)

รหัสดำเนินการ (Opcode)	สัญลักษณ์	คำอธิบายการทำงานของคำสั่ง
00	LIT	ds.push(oper) pc = pc + 2
01	LOAD (@)	addr = ds.pop() ds.push(mem[addr]) pc = pc + 1
02	STORE (!)	addr = ds.pop() mem[addr] = ds.pop() pc = pc + 1

03	DROP	ds.pop() pc = pc + 1
04	DUP	ds.push(ds.peek()) pc = pc + 1
05	OVER	ds.push(ds.nextToPeek()) pc = pc + 1
06	SWAP	tmp1 = ds.pop() tmp2 = ds.pop() ds.push(tmp1) ds.push(tmp2) pc = pc + 1
07 08 09 0A 0B	ADD (+) SUB (-) AND OR XOR	tmp2 = ds.pop() tmp1 = ds.pop() ds.push(tmp1 op tmp2) pc = pc + 1

0C	IF	tmp = ds.pop()
		if tmp = 0 then
		  pc = oper
		else
		  pc = pc + 2

0D	CALL	pc = oper rs.push(pc + 2)
0E	EXIT	pc = rs.pop()
0F	HALT	ซีพียูหยุดทำงาน
ยังสามารถเพิ่มคำสั่งได้อีกมาก (เหลือรหัสดำเนินการ x10 ถึง xFF)		

ใน simulator จะมีคำสั่งเพิ่มเติมคือ (โปรแกรมทั่วไปไม่ต้องใช้คำสั่งเหล่านี้)

>R ย้ายค่าจาก data stack ไป return stack

R> ย้ายค่าจาก return stack ไป data stack

ลองดูเหตุการณ์จำลองเมื่อทำแต่ละคำสั่ง → หมายถึง เลขที่อยู่ (address) ที่ PC ชี้อยู่ ก้นหลุมของสแตกคือด้านล่างในตาราง

<u>คำสั่ง LIT</u> โหลดค่าคงที่ที่บรรจุอยู่ในตัวถูกดำเนินการ (operand) ไปเก็บไว้ใน DS

ก่อนทำคำสั่ง				หลังทำคำสั่ง			
MEM		~~DS~~ Data Stack	~~RS~~ Return Stack	MEM		~~DS~~ Data Stack	~~RS~~ Return Stack
address	data			address	data		
→0	LIT			0	LIT		
1	10			1	10		
2				→2		10	

คำสั่ง LOAD (@) เอาค่าจาก MEM ไปเก็บไว้ใน DS

ก่อนทำคำสั่ง				หลังทำคำสั่ง			
MEM		~~DS~~ Data Stack	~~RS~~ Return Stack	MEM		~~DS~~ Data Stack	~~RS~~ Return Stack
address	data			address	data		
→0	LOAD			0	LOAD		
1				→1			
2				2			
3				3			
4				4			
5	10	5		5	10	10	

คำสั่ง STORE (!) เอาค่าจาก DS ไปเก็บไว้ใน MEM

ก่อนทำคำสั่ง				หลังทำคำสั่ง			
MEM		~~DS~~ Data Stack	~~RS~~ Return Stack	MEM		~~DS~~ Data Stack	~~RS~~ Return Stack
address	data			address	data		
→0	STORE			0	STORE		
1				→1			
2				2			
3				3			
4		5		4			
5		10		5	10		

คำสั่ง DROP โยนค่าที่อยู่บนสุดของ DS ทิ้งไป

ก่อนทำคำสั่ง				หลังทำคำสั่ง			
MEM		~~DS~~ Data Stack	~~RS~~ Return Stack	MEM		~~DS~~ Data Stack	~~RS~~ Return Stack
address	data			address	data		
→0	DROP	20		0	DROP		
1		10		→1		10	

คำสั่ง DUP ทำสำเนา (duplicate) ค่าที่อยู่บนสุดของ DS

ก่อนทำคำสั่ง				หลังทำคำสั่ง			
MEM		~~DS~~ Data Stack	~~RS~~ Return Stack	MEM		~~DS~~ Data Stack	~~RS~~ Return Stack
address	data			address	data		
→0	DUP			0	DUP	10	
1		10		→1		10	

คำสั่ง OVER ทำสำเนา (duplicate) ค่าที่อยู่ก่อนบนสุดของ DS

ก่อนทำคำสั่ง				หลังทำคำสั่ง			
MEM		~~DS~~ Data Stack	~~RS~~ Return Stack	MEM		~~DS~~ Data Stack	~~RS~~ Return Stack
address	data			address	data		
→0	OVER			0	OVER	10	
1		20		→1		20	
2		10		2		10	

คำสั่ง SWAP สลับตำแหน่งสองค่าที่อยู่บนสุดของ DS

ก่อนทำคำสั่ง				หลังทำคำสั่ง			
MEM		~~DS~~ Data Stack	~~RS~~ Return Stack	MEM		~~DS~~ Data Stack	~~RS~~ Return Stack
address	data			address	data		
→0	SWAP			0	SWAP		
1		20		→1		10	
2		10		2		20	

คำสั่ง ADD (+), SUB (−), AND, OR, XOR เอาสองค่าที่อยู่บนสุดของ DS มากระทำกัน

ก่อนทำคำสั่ง				หลังทำคำสั่ง			
Memory		~~DS~~ Data Stack	~~RS~~ Return Stack	Memory		~~DS~~ Data Stack	~~RS~~ Return Stack
address	data			address	data		
→0	ADD			0	ADD		
1		20		→1			
2		10		2		30	

หมายเหตุ ถ้าเป็นคำสั่งลบ ค่าในตารางนี้จะทำให้ได้ผลลัพธ์คือ −10 ไม่ใช่ +10

คำสั่ง IF (กรณีที่ค่าบนสุดของ DS เท่ากับ 0 หรือเรียกว่า taken) ไปทำงานเมื่อเงื่อนไขเป็นจริง

ก่อนทำคำสั่ง				หลังทำคำสั่ง			
Memory		~~DS~~ Data Stack	~~RS~~ Return Stack	Memory		~~DS~~ Data Stack	~~RS~~ Return Stack
address	data			address	data		
→0	IF			0	IF		
1	5			1	5		
2				2			
3				3			
4				4			
5		0		→5			

คำสั่ง IF (กรณีที่ค่าบนสุดของ DS ไม่เท่ากับ 0 หรือเรียกว่า not taken) ไปทำงานเมื่อเงื่อนไขเป็นเท็จ

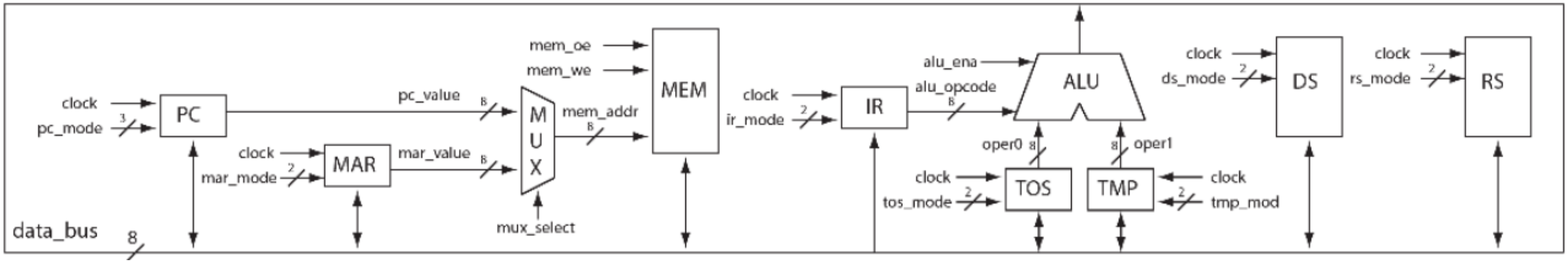
ก่อนทำคำสั่ง				หลังทำคำสั่ง			
Memory		~~DS~~ Data Stack	~~RS~~ Return Stack	Memory		~~DS~~ Data Stack	~~RS~~ Return Stack
address	data			address	data		
→0	IF			0	IF		
1	5			1	5		
2				→2			
3				3			
4				4			
5		non-zero		5			

คำสั่ง CALL ไปทำฟังก์ชันของโปรแกรม

ก่อนทำคำสั่ง				หลังทำคำสั่ง			
Memory		~~DS~~ Data Stack	~~RS~~ Return Stack	Memory		~~DS~~ Data Stack	~~RS~~ Return Stack
address	data			address	data		
→0	CALL			0	CALL		
1	5			1	5		
2				2			
3				3			
4				4			
5	EXIT			→5	EXIT		2

คำสั่ง EXIT ออกจากฟังก์ชัน และกลับไปทำงานต่อจากที่เดิม

ก่อนทำคำสั่ง				หลังทำคำสั่ง			
Memory		~~DS~~ Data Stack	~~RS~~ Return Stack	Memory		~~DS~~ Data Stack	~~RS~~ Return Stack
address	data			address	data		
0	CALL			0	CALL		
1	5			1	5		
2				→ 2			
3				3			
4				4			
→ 5	EXIT		2	5	EXIT		



รูปที่ 9.1 วิถีข้อมูล (data path) ของสแตกซีพียู

Stack CPU Simulator

PC	Address	Memory	Data Stack	Return Stack
x	x00	LIT ▼		
	x01	x0A (+10) ▼		
	x02	HALT ▼		
	x03	! ▼		
	x04	! ▼		

Stack Simulator (8-bit)
https://cache111.com/me/2301274/Simulator_8.html

Stack Simulator (16-bit)
https://cache111.com/me/2301274/Simulator_16.html

Verilog HDL

มอดูล (module)

มอดูลคือวงจรดิจิตัลที่ทำงานอย่างใดอย่างหนึ่ง มีการรับ/ส่งข้อมูลกับมอดูลอื่น ๆ ผ่าน พอร์ต (port) ซึ่งมี 3 แบบ คือ

input	เป็นพอร์ตที่ใช้รับค่าเข้ามาในมอดูล
output	เป็นพอร์ตที่ใช้ส่งค่าออกจากมอดูล
inout	เป็นทั้งอินพุตและเอาต์พุตพอร์ต

ตัวอย่างการประกาศมอดูล

```
module and(a, b, c); // AND  gate
        input    a, b;
        output   c;
        ..........
endmodule
```

```
module adder(oper1, oper2, sum);          // 8-bit adder
        input    [7:0]    oper1;
        input    [7:0]    oper2;
        output   [7:0]    sum;
        ..........
endmodule
module sram(cs, oe, we, addr, data);      // static random access memory
        input    cs, oe, we;              // chip select, output enable, write enable
        input    [7:0]    addr;           // address bus
        inout    [7:0]    data;           // data bus
endmodule
```

Comment

ใช้ // และ /* */ เพื่อบอกว่าข้อความหลัง // และข้อความใน /* */ เป็น comment

Constant

การระบุค่าคงที่ให้ระบุ 1) จำนวนบิต 2) ค่าของแต่ละบิต (0, 1, z) เช่น

1'b0	ขนาด 1 บิต ค่าของแต่ละบิตคือ 0
1'b1	ขนาด 1 บิต ค่าของแต่ละบิตคือ 1
4'b1010	ขนาด 4 บิต ค่าของแต่ละบิตคือ 1 0 1 0
8'b0000_1111	ขนาด 8 บิต ค่าของแต่ละบิตคือ 0 0 0 0 1 1 1 1
1'bz	ขนาด 1 บิต ค่าของแต่ละบิตคือ z
8'bzzzz_zzzz	ขนาด 8 บิต ค่าของแต่ละบิตคือ z z z z z z z z

Wire

เส้นลวดในมอดูลประกาศไว้หลังจากการประกาศพอร์ต พอร์ตที่ประกาศไว้เป็นเส้นลวดโดยอัตโนมัติ ไม่ต้องประกาศซ้ำ ตัวอย่างการประกาศเส้นลวด เช่น

```
wire      a, b;
wire      [7:0]     c;          // c[7] เป็น MSB และ c[0] เป็น LSB
```

การอ้างถึงเส้นลวด เช่น a, b, c[7], c[0], c[7:0], c[7:4] เป็นต้น

Assign

การกำหนดค่าให้เส้นลวด เช่น

```
assign a = 1'b1;
assign c[7:0] = a[7:0] + 1'b1;
assign c[7:0] = a[7:0] + b[7:0];
assign c[7:0] = { c[6:0], 1'b0 };    // shift left
assign c = (s == 1'b0 ? a : b);      // 2-way mux
assign e = (s0 == 1'b0 ? (s1 == 1'b0 ? a : b) : (s1 == 1'b0 ? c : d));  // 4-way mux
```

เขียนแบบนี้เรียกว่า ternary conditional operator

หรือเขียนแบบนี้ก็ได้

```
assign e =   ({s0, s1} == 2'b00 ? a :          // 4-way mux
             ({s0, s1} == 2'b01 ? b :
             ({s0, s1} == 2'b10 ? c : d)));
```

Reg

เรจิสเตอร์เป็นอุปกรณ์ที่จำค่าได้ เช่น สั่งให้จำค่าในเส้นลวด เมื่อเวลาผ่านไปค่าในเส้นลวดเปลี่ยนไป แต่ค่าในเรจิสเตอร์ยังเป็นค่าเดิม

```
reg       a, b;
reg       [7:0]     c;          // c[7] เป็น MSB และ c[0] เป็น LSB
```

สามารถประกาศพอร์ต output ให้เป็นเรจิสเตอร์ได้ (เป็นทั้งเรจิสเตอร์และเอาต์พุต)

Always

คำสั่ง alwalys เป็นการสั่งให้เรจิสเตอร์จำค่า เช่น

```
always @(posedge clock)  // ทำทุก ๆ ครั้งที่ขอบขาขึ้นของ clock (เปลี่ยนค่าจาก 0 เป็น 1)
        begin
                . . . . . .
        end
```

สัญญาณควบคุม

เมื่อต้องสร้างสัญญาณควบคุมจำนวนมาก วิธีเขียนง่าย ๆ ในบรรทัดเดียวคือ

```
{ a[1:0], b[1:0], c[1:0], d, e } = 8'b01_10_11_10;
```

จะสั้นกว่าและดูง่ายกว่าเขียนทีละบรรทัดแบบนี้

```
a[1:0] = 2'b01;

b[1:0] = 2'b10;

c[1:0] = 2'b11;

d = 1'b1;

e = 1'b0;
```

Program Counter (PC)

PC เป็นเรจิสเตอร์ขนาด 8 บิต เก็บเลขที่อยู่ (address) ของคำสั่งที่กำลังถูกกระทำ (execute) PC ทำงานตามจังหวะสัญญาณนาฬิกา (clock) ที่ขอบขาขึ้น (positive edge) ของสัญญาณนาฬิกา (ยกเว้น mode = 2)

ถ้า mode = 0	รีเซตค่า PC เป็น 0
ถ้า mode = 1	เซตค่า PC ให้เท่ากับค่าที่อยู่ใน data bus
ถ้า mode = 2	เขียนค่า PC ลงไปใน data bus
ถ้า mode = 3	ไม่ต้องทำอะไร (ให้ค่า z ที่ data bus)
ถ้า mode = 4	เพิ่มค่า PC ไปอีกหนึ่งหน่วย

```
module pc(clock, pc_mode, data_bus, pc_value);

  input           clock;
  input           [2:0]          pc_mode;
  inout           [7:0]          data_bus;
  output          [7:0]          pc_value;
  reg             [7:0]          pc_value;
  reg             [7:0]          temp; // ไม่ได้สร้าง register จริง ๆ

  // คำสั่งนี้ไม่รอสัญญาณนาฬิกา
  assign data_bus[7:0] = (pc_mode[2:0] == 3'b010 ? pc_value[7:0] : temp[7:0]);
```

```
always @(posedge clock)
        begin
                case(pc_mode[2:0])
                        3'b000:
                                begin
                                        pc_value[7:0] = 8'b0000_0000;
                                        temp[7:0] = 8'bzzzz_zzzz;
                                end
                        3'b001:
                                begin
                                        pc_value[7:0] = data_bus[7:0];
                                        temp[7:0] = 8'bzzzz_zzzz;
                                end
                        3'b010:
                                begin
                                        // do nothing (โหมดนี้ไม่ทำงานตามสัญญาณนาฬิกา)
                                end
                        3'b011:
                                begin
                                        temp[7:0] = 8'bzzzz_zzzz;
                                end
                        3'b100:
                                begin
                                        pc_value[7:0] = pc_value[7:0] + 1'b1;
                                        temp[7:0] = 8'bzzzz_zzzz;
                                end
                endcase
        end
endmodule
```

Memory Address Register (MAR)

MAR เป็นเรจิสเตอร์ขนาด 8 บิต เก็บเลขที่อยู่ (address) ที่จะอ่านเขียนหน่วยความจำ MAR ทำงานตามจังหวะสัญญาณนาฬิกา (clock) ที่ขอบขาขึ้น (positive edge) ของสัญญาณนาฬิกา (ยกเว้น mode = 2)

ถ้า mode = 0 รีเซตค่า MAR เป็น 0

ถ้า mode = 1 เซตค่า MAR ให้เท่ากับค่าที่อยู่ใน data bus

ถ้า mode = 2 เขียนค่า MAR ลงไปใน data bus

ถ้า mode = 3 ไม่ต้องทำอะไร (ให้ค่า z ที่ data bus)

```
module register(clock, register_mode, data_bus, register_value);

  input           clock;
  input           [1:0]           register_mode;
  inout           [7:0]           data_bus;
  output  [7:0]           register_value;
  reg             [7:0]           register_value;
  reg             [7:0]           temp;

  // คำสั่งนี้ไม่รอสัญญาณนาฬิกา
  assign data_bus[7:0] = (register_mode[1:0] == 2'b10 ? register_value[7:0] : temp[7:0]);
```

```
always @(posedge clock)
        begin
                case(register_mode[1:0])
                        2'b00:
                                begin
                                        register_value[7:0] = 8'b0000_0000;
                                        temp[7:0] = 8'bzzzz_zzzz;
                                end
                        2'b01:
                                begin
                                        register_value[7:0] = data_bus[7:0];
                                        temp[7:0] = 8'bzzzz_zzzz;
                                end
                        2'b10:
                                begin
                                        // do nothing (โหมดนี้ไม่ทำงานตามสัญญาณนาฬิกา)
                                end
                        2'b11:
                                begin
                                        temp[7:0] = 8'bzzzz_zzzz;
                                end
                endcase
        end
endmodule
```

Multiplexor (MUX)

ใช้เลือกระหว่าง PC และ MAR เพื่อกำหนดเลขที่อยู่ (address) ที่จะอ่าน/เขียนหน่วยความจำ

ถ้า select = 0 เลือกค่า PC

ถ้า select = 1 เลือกค่า MAR

ในภาษา Verilog เขียนอธิบายดังนี้

```
module mux(sel, in0, in1, out);

        input                   sel;
        input       [7:0]       in0;
        input       [7:0]       in1;
        output      [7:0]       out;

        assign out[7:0] = (sel == 1'b0 ? in0[7:0] : in1[7:0]);
endmodule
```

Memory (MEM)

MEM เป็นหน่วยความจำขนาด 256 x 8 บิต คือมี 256 ช่อง ช่องละ 8 บิต การอ่าน/เขียนหน่วยความจำใช้สัญญาณ OE (output enable) และ WE (write enable) เหมือนเอสแรมปกติ

ถ้า mem_oe = 0 และ mem_we = 1 เขียนค่าที่อยู่ในช่องที่ mem_addr ชี้อยู่ลง data bus (read)

ถ้า mem_oe = 1 และ positive edge ของ mem_we เขียนค่าใน data bus ลงช่องที่ mem_addr ชี้อยู่ (write)

ถ้า mem_oe = 1 และ mem_we = 1 ไม่ต้องทำอะไร ให้ค่า high impedance (z) ที่ data bus

Instruction Register (IR)

IR เป็นเรจิสเตอร์ขนาด 8 บิต เก็บรหัสดำเนินการ (opcode) ของคำสั่งที่ถูกระทำ (execute) อยู่ ในภาษา Verilog เขียนอธิบายเหมือน MAR

Top of Stack (TOS)

TOS เป็นเรจิสเตอร์ขนาด 8 บิต เก็บของชิ้นบนสุดที่อยู่ใน data stack ของชิ้นอื่นๆ ที่เหลือจะเก็บไว้ในมอดูล DS สาเหตุที่ทำเช่นนี้เพราะว่า TOS มักจะถูกป้อนเข้าไปให้คำนวณใน ALU บ่อยๆ ดังนั้นจึงเก็บไว้ในเรจิสเตอร์ที่แยกต่างหากออกมาจาก DS ในภาษา Verilog เขียนอธิบายเหมือน MAR

Temporary (TMP)

TMP เป็นเรจิสเตอร์ขนาด 8 บิต ใช้เก็บตัวถูกดำเนินการ (operand) ตัวที่สองไว้ชั่วคราว ตัวถูกดำเนินการตัวแรกอยู่ใน TOS ในภาษา Verilog เขียนอธิบายเหมือน MAR

Arithmetic Logic Unit (ALU)

ALU เป็นวงจรคำนวณฟังก์ชันพื้นฐานเช่น add (+), sub (-), and, or, xor เป็นต้น ขึ้นอยู่กับ alu_opcode

ถ้า alu_ena = 0 จะเขียนผลลัพธ์ลง data bus

ถ้า alu_ena = 1 จะเขียนค่า z (high impedance)

ในภาษา Verilog เขียนอธิบายดังนี้

```
module alu(ena, opcode, oper0, oper1, data_bus);

  input          ena;
  input   [7:0]  opcode;
  input   [7:0]  oper0;
  input   [7:0]  oper1;
  inout   [7:0]  data_bus;
  wire    [7:0]  temp;

  assign data_bus[7:0] = (ena == 1'b0 ? temp[7:0] : 8'bzzzz_zzzz);

  assign temp[7:0] =
         (opcode[7:0] == 8'b0000_0111 ? oper1[7:0] + oper0[7:0] :    // ADD
         (opcode[7:0] == 8'b0000_1000 ? oper1[7:0] - oper0[7:0] :    // SUB
         (opcode[7:0] == 8'b0000_1001 ? oper1[7:0] & oper0[7:0] :    // AND
         (opcode[7:0] == 8'b0000_1010 ? oper1[7:0] | oper0[7:0] :    // OR
         (opcode[7:0] == 8'b0000_1011 ? oper1[7:0] ^ oper0[7:0] :    // XOR
             8'b0000_0000 )))));
endmodule
```

Data Stack (DS) และ Return Stack (RS)

DS และ RS เป็นสแตกที่จุของได้ 256 ชิ้น แต่ละชิ้นมีขนาด 8 บิต ทำงานตามจังหวะสัญญาณนาฬิกา (clock) ที่ขอบขาขึ้น (positive edge) ของสัญญาณนาฬิกา

ถ้า mode = 0 ทำให้สแตกว่าง (empty stack)

ถ้า mode = 1 push ค่าจาก data bus ลงไปในสแตก

ถ้า mode = 2 pop ค่าจากสแตกและเขียนลง data bus

ถ้า mode = 3 ไม่ต้องทำอะไร (ให้ค่า z ที่ data bus)

ขอย้ำว่าพอทำเป็นฮาร์ดแวร์แล้วตัวบนสุด (top of stack) ของ data stack จะอยู่ที่เรจิสเตอร์ TOS เสมอ ตัวที่เหลือจะเก็บในมอดูล DS ส่วน return stack เก็บข้อมูลทั้งหมดไว้ในมอดูล RS ในภาษา Verilog เขียนอธิบายดังนี้ (มอดูล DS และ RS ทำงานเหมือนกัน)

```
1   module ds(clock, mode, data_bus);
2
3       input   clock;
4       input   [1:0]   mode;
5       inout   [7:0]   data_bus;
6
7       reg     [7:0]   tos;
8       reg     [7:0]   data[0:255];
9       reg     [7:0]   temp;
10
11      assign data_bus[7:0] = (mode[1:0] == 2'b10 ? data[tos[7:0]][7:0] : temp[7:0]);
```

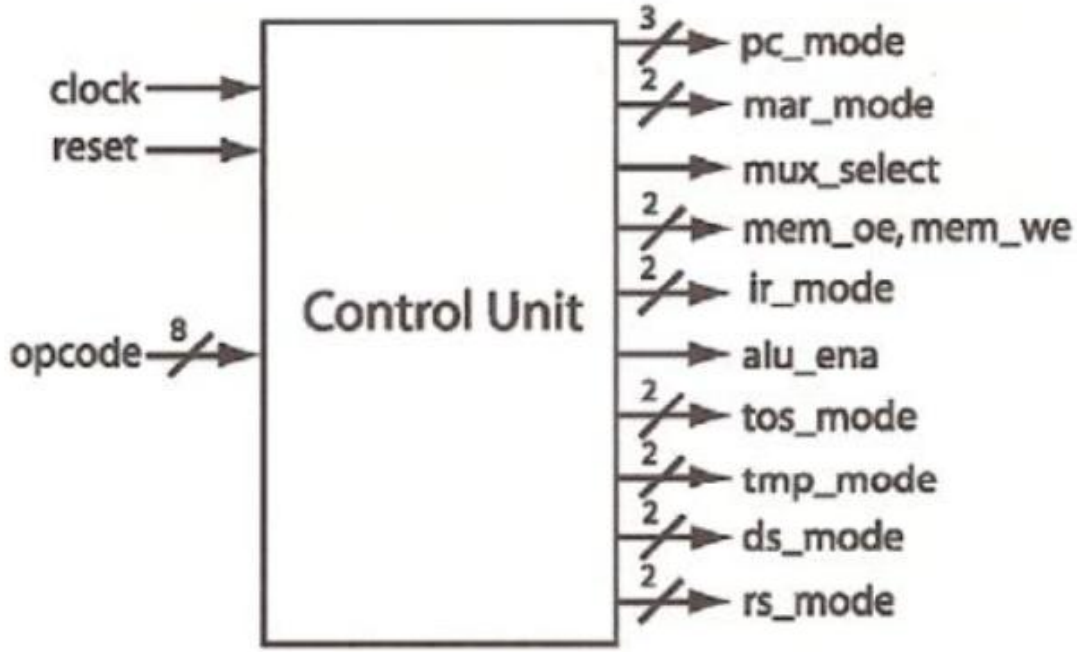
mode = 2 เขียน tos ลง data bus ทันที (asyn)

```
13      always @(posedge clock)
14          begin
15              case(mode[1:0])
16                  2'b00:
17                      begin
18                          tos[7:0] = 8'b1111_1111;
19                          temp[7:0] = 8'bzzzz_zzzz;
20                      end
21                  2'b01:
22                      begin
23                          tos[7:0] = tos[7:0] + 1'b1 ;
24                          data[tos[7:0]][7:0] = data_bus[7:0];
25                          temp[7:0] = 8'bzzzz_zzzz;
26                      end
27                  2'b10:
28                      begin
29                          tos[7:0] = tos[7:0] - 1'b1;
30                      end
31                  2'b11:
32                      begin
33                          temp[7:0] = 8'bzzzz_zzzz;
34                      end
35              endcase
36          end
37
38  endmodule
```

pop ตามสัญญาณนาฬิกา (ทำที่ positive edge)

Control Unit

Control unit เป็นเครื่องจำกัดสถานะ (FSM) ที่สร้างสัญญาณไปควบคุมมอดูลต่างๆ ให้ทำงานได้ถูกต้อง (ดูรูปที่ 9.2) ที่ขอบขาขึ้นของสัญญาณนาฬิกา ถ้า reset = 0 จะรีเซตสถานะของ FSM ให้อยู่ที่สถานะเริ่มต้น สาย opcode นั้นลากมาจาก IR เพราะ FSM จำเป็นต้องรู้ว่าซีพียูกำลังทำคำสั่ง (instruction) อะไร เพื่อที่จะได้สร้างสัญญาณควบคุมได้ถูกต้อง สัญญาณที่ส่งไปควบคุมมอดูลต่างๆ อยู่ทางฝั่งขวาของ control unit ปกติจะไม่นิยมเขียน control unit ลงในวิถีข้อมูล (data path) เพราะจะทำให้สายไฟพันกันจนดูไม่รู้เรื่อง



รูปที่ 9.2 Control unit ของสแตกซีพียู

```
module control(clock, reset, data_bus, oper0, pc_mode, mar_mode, mux_select, mem_oe,
mem_we, ir_mode, alu_ena, tos_mode, tmp_mode, ds_mode, rs_mode);

  input                 clock;
  input                 reset;
  input   [7:0]         data_bus;
  input   [7:0]         oper0;
  output  [2:0]         pc_mode;
  output  [1:0]         mar_mode;
  output                mux_select;
  output                mem_oe;
  output                mem_we;
  output  [1:0]         ir_mode;
  output                alu_ena;
  output  [1:0]         tos_mode;
  output  [1:0]         tmp_mode;
  output  [1:0]         ds_mode;
  output  [1:0]         rs_mode;

  reg     [7:0]         state;
  reg     [18:0]        cont;

  assign {pc_mode[2:0], mar_mode[1:0], mux_select, mem_oe, mem_we, ir_mode[1:0],
alu_ena, tos_mode[1:0], tmp_mode[1:0], ds_mode[1:0], rs_mode[1:0]} = cont[18:0];
```

```
always @(posedge clock)
      begin
         if (reset == 1'b0)
            begin
               state[7:0] = 8'b0000_0000;
            end
         else
            begin
               case(state[7:0])
                  8'b0000_0000 : // reset
                     begin
                        cont[18:0] = 19'b000_00_0_1_1_00_1_00_00_00_00;
                        state[7:0] = 8'b0000_0001;
                     end
                  8'b0000_0001 : // fetch
                     begin
                        cont[18:0] = 19'b011_11_0_0_1_01_1_11_11_11_11;
                        state[7:0] = 8'b0000_0010;
                     end
```

```
8'b0000_0010 : // decode, pc = pc + 1
   begin
      cont[18:0] = 19'b100_11_0_1_1_11_1_11_11_11_11;
      case (data_bus[7:0]) // ตัวเลขในวงเล็บคือจำนวน clock cycle ที่ใช้
         8'b0000_0000 : state[7:0] = 8'b0000_0011; // LIT (2)
         8'b0000_0001 : state[7:0] = 8'b0000_0101; // LOAD (2)
         8'b0000_0010 : state[7:0] = 8'b0000_0111; // STORE (3)
         8'b0000_0011 : state[7:0] = 8'b0000_1010; // ADD (2)
         8'b0000_0100 : state[7:0] = 8'b0000_1010; // SUB (2)
         8'b0000_0101 : state[7:0] = 8'b0000_1010; // AND (2)
         8'b0000_0110 : state[7:0] = 8'b0000_1010; // OR (2)
         8'b0000_0111 : state[7:0] = 8'b0000_1010; // XOR (2)
         8'b0000_1000 : state[7:0] = 8'b0000_1100; // DROP (1)
         8'b0000_1001 : state[7:0] = 8'b0000_1101; // DUP (1)
         8'b0000_1010 : state[7:0] = 8'b0000_1110; // OVER (4)
         8'b0000_1011 : state[7:0] = 8'b0001_0010; // SWAP (3)
         8'b0000_1100 :
            begin
               if (oper0[7:0] != 8'b0000_0000)
                  state[7:0] = 8'b0001_0101; // IF(1) branch not taken
               else
                  state[7:0] = 8'b0001_0110; // IF(2) branch taken
            end
         8'b0000_1101 : state[7:0] = 8'b0001_1000; // CALL (4)
         8'b0000_1110 : state[7:0] = 8'b0001_1100; // EXIT (1)
         8'b0000_1111 : state[7:0] = 8'b1111_1111; // HALT
      endcase
   end
```

```
8'b0000_0011 : // lit (1) : ds.push(tos)
   begin
      cont[18:0] = 19'b011_11_0_1_1_11_1_10_11_01_11;
      state[7:0] = 8'b0000_0100;
   end
8'b0000_0100 : // lit (2) : tos = mem[pc], pc = pc + 1
   begin
      cont[18:0] = 19'b100_11_0_0_1_11_1_01_11_11_11;
      state[7:0] = 8'b0000_0001;
   end
8'b0000_0101 : // load (1) : mar = tos
   begin
      cont[18:0] = 19'b011_01_0_1_1_11_1_10_11_11_11;
      state[7:0] = 8'b0000_0110;
   end
8'b0000_0110 : // load (2) : tos = mem[mar]
   begin
      cont[18:0] = 19'b011_11_1_0_1_11_1_01_11_11_11;
      state[7:0] = 8'b0000_0001;
   end
```

```
8'b0000_0111 : // store (1) : mar = tos
   begin
      cont[18:0] = 19'b011_01_0_1_1_11_1_10_11_11_11;
      state[7:0] = 8'b0000_1000;
   end
8'b0000_1000 : // store (2) : data_bus = ds.pop()
   begin
      cont[18:0] = 19'b011_11_1_1_0_11_1_11_11_10_11;
      state[7:0] = 8'b0000_1001;
   end
8'b0000_1001 : // store (3) : write, tos = ds.pop()
   begin
      cont[18:0] = 19'b011_11_0_1_1_11_1_01_11_10_11;
      state[7:0] = 8'b0000_0001;
   end
8'b0000_1010 : // add, sub, and, or, xor (1) : tmp = ds.pop()
   begin
      cont[18:0] = 19'b011_11_0_1_1_11_1_11_01_10_11;
      state[7:0] = 8'b0000_1011;
   end
8'b0000_1011 : // add, sub, and, or, xor (2) : tos = tos op tmp
   begin
      cont[18:0] = 19'b011_11_0_1_1_11_0_01_11_11_11;
      state[7:0] = 8'b0000_0001;
   end
8'b0000_1100 : // drop (1) : tos = ds.pop()
   begin
      cont[18:0] = 19'b011_11_0_1_1_11_1_01_11_10_11;
      state[7:0] = 8'b0000_0001;
   end
```

```
8'b0000_1101 : // dup (1) : ds.push(tos)
   begin
      cont[18:0] = 19'b011_11_0_1_1_11_1_10_11_01_11;
      state[7:0] = 8'b0000_0001;
   end
8'b0000_1110 : // over (1) : tmp = ds.pop()
   begin
      cont[18:0] = 19'b011_11_0_1_1_11_1_11_01_10_11;
      state[7:0] = 8'b0000_1111;
   end
8'b0000_1111 : // over (2) : ds.push(tmp)
   begin
      cont[18:0] = 19'b011_11_0_1_1_11_1_11_10_01_11;
      state[7:0] = 8'b0001_0000;
   end
8'b0001_0000 : // over (3) : ds.push(tos)
   begin
      cont[18:0] = 19'b011_11_0_1_1_11_1_10_11_01_11;
      state[7:0] = 8'b0001_0001;
   end
8'b0001_0001 : // over (4) : tos = tmp
   begin
      cont[18:0] = 19'b011_11_0_1_1_11_1_01_10_11_11;
      state[7:0] = 8'b0000_0001;
   end
```

```
8'b0001_0010 : // swap (1) : tmp = tos
   begin
      cont[18:0] = 19'b011_11_0_1_1_11_1_10_01_11_11;
      state[7:0] = 8'b0001_0011;
   end
8'b0001_0011 : // swap (2) : tos = ds.pop()
   begin
      cont[18:0] = 19'b011_11_0_1_1_11_1_01_11_10_11;
      state[7:0] = 8'b0001_0100;
   end
8'b0001_0100 : // swap (3) : ds.push(tmp)
   begin
      cont[18:0] = 19'b011_11_0_1_1_11_1_11_10_01_11;
      state[7:0] = 8'b0000_0001;
   end
```

```
8'b0001_0101 : // IF (not taken 1) : pc = pc + 1, tos = ds.pop()
   begin
      cont[18:0] = 19'b100_11_0_1_1_11_1_01_11_10_11;
      state[7:0] = 8'b0000_0001;
   end
8'b0001_0110 : // IF (taken 1) : pc = mem[pc]
   begin
      cont[18:0] = 19'b001_11_0_0_1_11_1_11_11_11_11;
      state[7:0] = 8'b0001_0111;
   end
8'b0001_0111 : // IF (taken 2) : tos = ds.pop()
   begin
      cont[18:0] = 19'b011_11_0_1_1_11_1_01_11_10_11;
      state[7:0] = 8'b0000_0001;
   end
```

```
8'b0001_1000 : // CALL (1) : mar = pc
   begin
      cont[18:0] = 19'b010_01_0_1_1_11_1_11_11_11_11;
      state[7:0] = 8'b0001_1001;
   end
8'b0001_1001 : // CALL (2) : pc = pc + 1
   begin
      cont[18:0] = 19'b100_11_0_1_1_11_1_11_11_11_11;
      state[7:0] = 8'b0001_1010;
   end
8'b0001_1010 : // CALL (3) : rs.push(pc)
   begin
      cont[18:0] = 19'b010_11_0_1_1_11_1_11_11_11_01;
      state[7:0] = 8'b0001_1011;
   end
8'b0001_1011 : // CALL (4) : pc = mem[mar]
   begin
      cont[18:0] = 19'b001_11_1_0_1_11_1_11_11_11_11;
      state[7:0] = 8'b0000_0001;
   end
```

```
                8'b0001_1100 : // EXIT (1) : pc = rs.pop()
                   begin
                      cont[18:0] = 19'b001_11_0_1_1_11_1_11_11_11_10;
                      state[7:0] = 8'b0000_0001;
                   end
                8'b1111_1111 : // halt
                   begin
                      cont[18:0] = 19'b011_11_0_1_1_11_1_11_11_11_11;
                      state[7:0] = 8'b1111_1111;
                   end
             endcase
          end
       end
endmodule
```

สังเกตว่าแต่ละคำสั่งใช้จำนวน clock cycle ไม่เท่ากัน คำสั่งที่ซับซ้อนต้องใช้จำนวนรอบสัญญาณนาฬิกามากกว่าจะทำเสร็จ ในขณะที่คำสั่งง่ายๆ ใช้เพียง 1 หรือ 2 รอบสัญญาณนาฬิกาก็ทำเสร็จแล้ว ข้อสังเกตอีกอย่างคือ การเปลี่ยนสถานะและเอาต์พุตของ FSM ที่เขียนด้วย Verilog อยู่ภายในคำสั่ง always @(posedge clock) ดังนั้นทั้งสถานะและเอาต์พุตจะเปลี่ยนค่าที่ขอบขาขึ้นของสัญญาณนาฬิกาเท่านั้น