

Digital System Design

Term-Project

Mu0 processor design

전자정보공학부 IT융합과 20160426 김도현

전자정보공학부 IT융합과 20160518 오승주

전자정보공학부 IT융합과 20160528 윤창섭

< 목차 >

1. 역할분담
2. 프로젝트 목표
3. FSM 설계
4. 프로세서 설계
5. 메모리 설계
6. control logic data flow 설계
7. Testbench 동작검증
8. Synthesis & RTL viewer
9. 전체 Source Code (Verilog)

1. 역할분담

1) 20160426 김도현

- Test Bench 설계
- FSM 설계
- Mu0 프로세서 설계
- Synthesis and RTL 결과 도출
- 파형 분석 및 보완
- 보고서 작성

2) 20160518 오승주

- Test Bench 설계
- FSM 설계
- Memory 설계
- Control logic과 data flow 분석
- 보고서 작성
- 오류 검사 및 보완

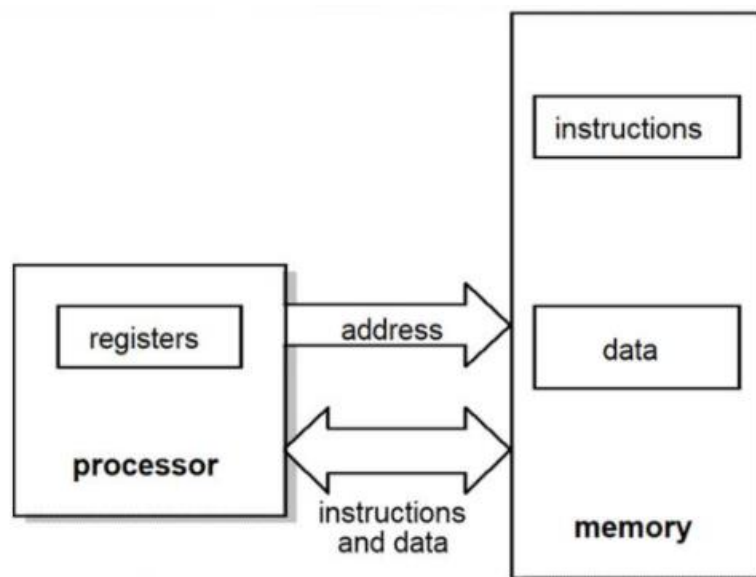
3) 20160528 윤창섭

- Test Bench 설계
- Mu0 프로세서 설계
- Memory 설계
- Control logic과 data flow 분석
- Synthesis and RTL 결과 도출
- 보고서 작성
- 파형분석 및 결과 도출

2. 프로젝트 목표

Mu0 Processor를 구현하고 $\sum_{i=s}^N i (N \geq S)$ 에 대한 Task를 명령의 형태로 memory에 저장하여 processor, memory 및 control logic간의 회로동작을 통해 1부터 10까지의 시그마 값을 구합니다.

1) Mu0 processor란?



Mu0 프로세서는 컴퓨터 시스템으로 프로세서와 메모리로 구성됩니다. 아주 단순한 프로세서로 기본적인 연산만을 수행합니다. Memory와 process는 data bus로 연결되어지며 주소를 통해 값을 읽거나 쓸 수 있습니다.

프로세서에서는 total state가 움직이는 방식을 규정하기 위해 opcode를 이용하여 Low-level 방식으로 작동합니다.



<opcode format>

총 명령어의 길이는 16bit로 앞에 4bit는 수행 가능한 명령의 종류를 나타내며 총 16가지의 명령을 구별할 수 있습니다., 뒤에 12bit는 operand로 주소를 나타낼 때 쓰입니다. 12bit라 4096가지 메모리를 구별할 수 있지만 여기선 32개까지만 사용하도록 하였습니다.

그리고 mu0 프로세서는 다음과 같이 opcode에 따른 정해진 연산을 수행합니다.

Instruction	Opcode	Effect
LDA S	0000	$ACC := mem_{16}[S]$
STO S	0001	$mem_{16}[S] := ACC$
ADD S	0010	$ACC := ACC + mem_{16}[S]$
SUB S	0011	$ACC := ACC - mem_{16}[S]$
JMP S	0100	$PC := S$
JGE S	0101	If $ACC \geq 0$ $PC := S$
JNE S	0110	If $ACC \neq 0$ $PC := S$
STP	0111	stop

< mu0에서 수행하는 기본연산 opcode >

2) Task Description

위의 기본으로 정해진 연산들을 이용해 $\sum_{i=s}^N i (N \geq S)$ Task를

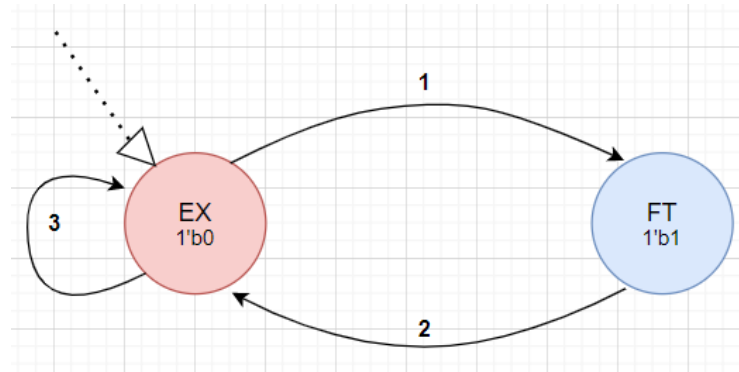
다음과 같은 순서로 설계합니다.

address		opcode	S
0	LDA S	0000	0000_0001_0000
1	STO sum	0001	0000_0001_0010
2	STO I	0001	0000_0001_0100
3	SUB N	0011	0000_0001_0001
4	JNE loop1	0110	0000_0000_0110
5	STP	0111	0000_0000_0000
6(loop1)	LDA I	0000	0000_0001_0100
7	ADD V1	0010	0000_0001_0011
8	STO I	0001	0000_0001_0100
9	ADD sum	0010	0000_0001_0010
10	STO sum	0001	0000_0001_0010
11	LDA I	0000	0000_0010_0100
12	SUB N	0011	0000_0001_0011
13	JNE loop1	0110	0000_0000_0110
14	STP	0111	0000_0000_0000
15		xxxx	x
16(S)		0000	0000_0000_0001
17(N)		0000	0000_0000_1010
18(sum)		xxxx	x
19(I)		xxxx	x
20(V1)		0000	0000_0000_0001
----		xxxx	x
31		xxxx	x

1. LDA S: 메모리에서 S번지 주소 값을 참조해 ACC에 저장한다.
2. STO sum: ACC에 저장된 값을 메모리의 sum번지에 저장한다.

3. STO i: acc에 저장된 값을 메모리의 i번지에 저장한다.
4. SUB N: acc에서 메모리의 N번째 값을 빼준다.
5. JNE loop1: acc값이 0이 아니면 pc에 loop1를 할당한다.
6. STP: 프로세서 동작을 멈춘다.
7. LDA i: acc에 메모리의 i번지 값을 저장한다.
8. ADD v1: 기존의 acc에 메모리의 v1번지 값을 저장한다.
9. STO : 메모리의 i번지 값에 acc(i+1)값이 저장된다.
10. ADD sum: 기존의 acc에 sum번지 메모리 값을 더해서 저장한다.
11. STO sum: ACC에 저장된 값을 메모리의 sum번지에 저장한다.
12. LDA i: acc에 메모리의 i번지 값을 저장한다.
13. SUB N: acc에서 메모리의 N번째 값을 빼준다.
14. JNE loop1: acc값이 0이 아니면 pc에 loop1를 할당한다.
15. STP: 프로세서 동작을 멈춘다.

3. Mu0 Processor FSM 설계



① EX->FT

input				Output									
opcode	acc_z	acc_msb	comp_sn	a_sel	b_sel	acc_ce	pc_ce	ir_ce	acc_oe	alu_fs		mem_rq	rnw
0000(LDA)	x	x	x	1	1	1	0	0	0	011	B	1	1
0001(STO)	x	x	x	1	x	0	0	0	1	x	x	1	0
0010(ADD)	x	x	x	1	1	1	0	0	0	001	A+B	1	1
0011(SUB)	x	x	x	1	1	1	0	0	0	010	A-B	1	1
1000(LDSA)	x	x	x	1	1	1	0	0	0	011	B	1	1
1001(LDN)	x	x	x	1	1	1	0	0	0	011	B	1	1
1100(STS)	x	x	x	1	x	0	0	0	0	x	x	1	0

② FT->EX

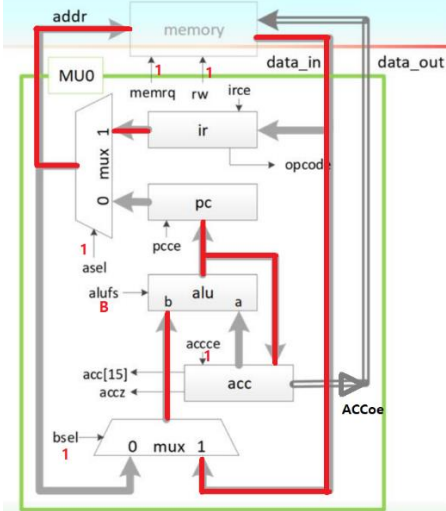
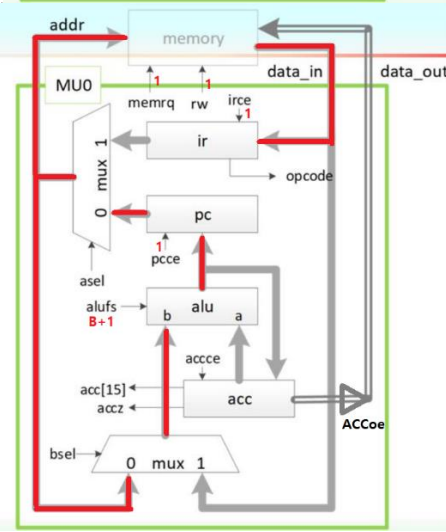
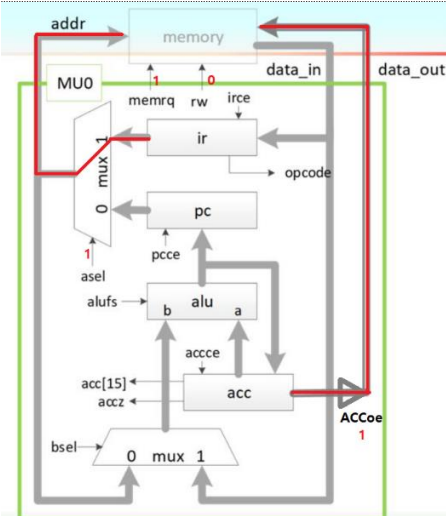
input				Output									
opcode	acc_z	acc_msb	comp_sn	a_sel	b_sel	acc_ce	pc_ce	ir_ce	acc_oe	alu_fs		mem_rq	rnw
0000(LDA)	x	x	x	0	0	0	1	1	0	100	B+1	1	1
0001(STO)	x	x	x	0	0	0	1	1	0	100	B+1	1	1
0010(ADD)	x	x	x	0	0	0	1	1	0	100	B+1	1	1
0011(SUB)	x	x	x	0	0	0	1	1	0	100	B+1	1	1
1000(LDSA)	x	x	x	0	0	0	1	1	0	100	B+1	1	1
1001(LDN)	x	x	x	0	0	0	1	1	0	100	B+1	1	1
1100(STS)	x	x	x	0	0	0	1	1	0	100	B+1	1	1

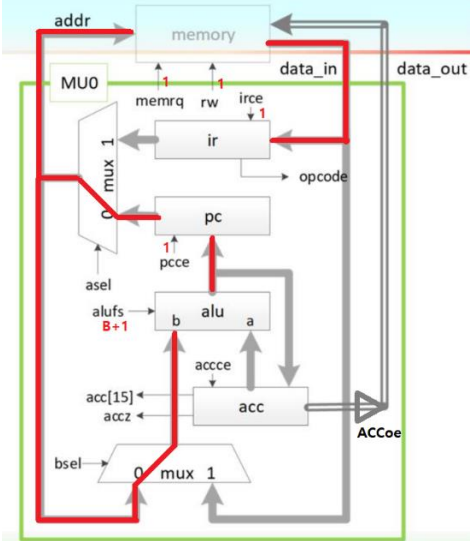
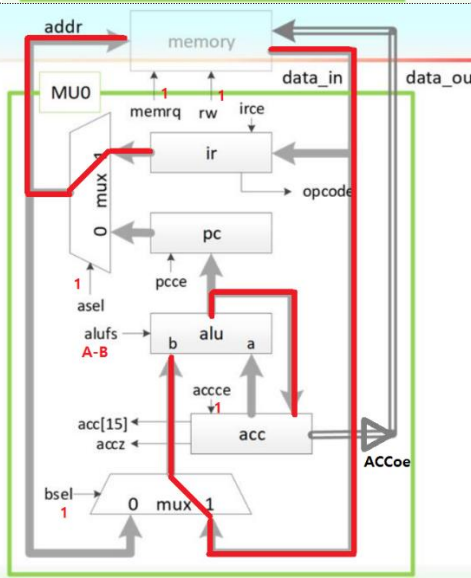
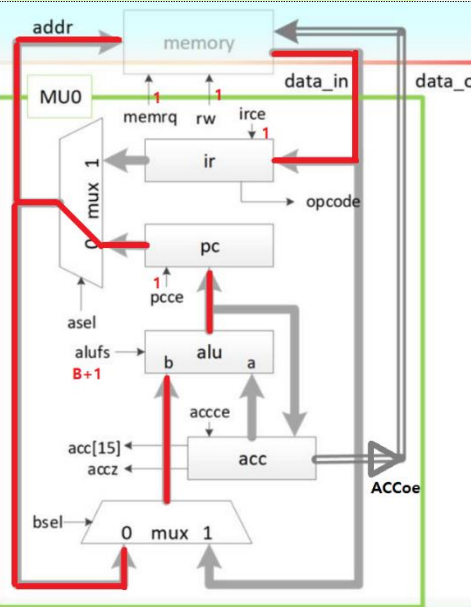
③ EX->EX

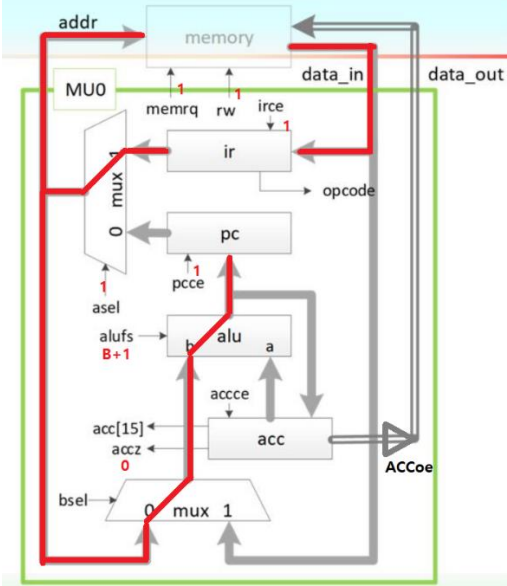
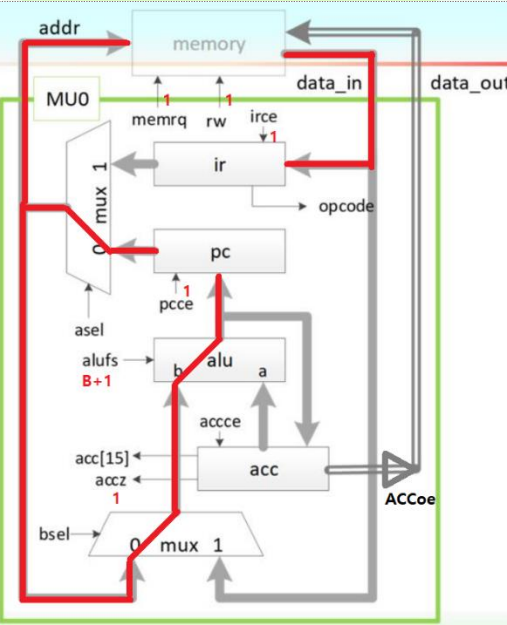
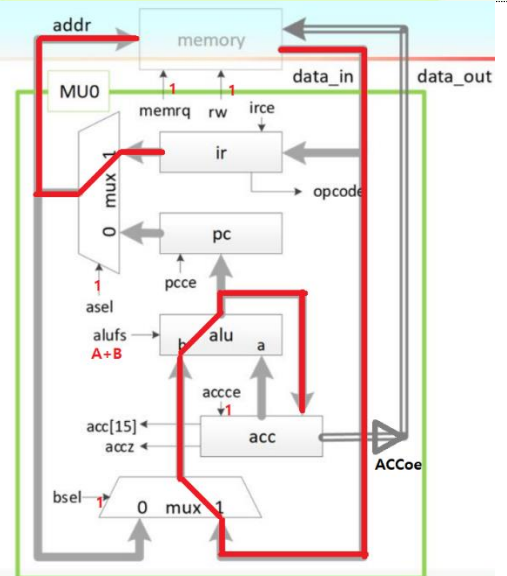
input				Output									
opcode	acc_z	acc_msb	comp_sn	a_sel	b_sel	acc_ce	pc_Ce	ir_Ce	acc_oe	alu_fs		mem_rq	rnw
0100(JMP)	x	x	x	1	0	0	1	1	0	100	B+1	1	1
0101(JGE)	x	0	x	1	0	0	1	1	0	100	B+1	1	1
0101(JGE)	x	1	x	0	0	0	1	1	0	100	B+1	1	1
0110(JNE)	0	x	x	1	0	0	1	1	0	100	B+1	1	1
0110(JNE)	1	x	x	0	0	0	1	1	0	100	B+1	1	1
0111(STOP)	x	x	x	1	x	0	0	0	0	x	x	0	1
1010(SUM)	x	x	1	0	0	0	1	1	0	100	B+1	1	1
1010(SUM)	x	x	0	1	0	0	1	1	0	100	B+1	1	1
1011(JLT)	x	x	1	0	0	0	1	1	0	100	B+1	1	1
1011(JLT)	x	x	0	1	0	0	1	1	0	100	B+1	1	1

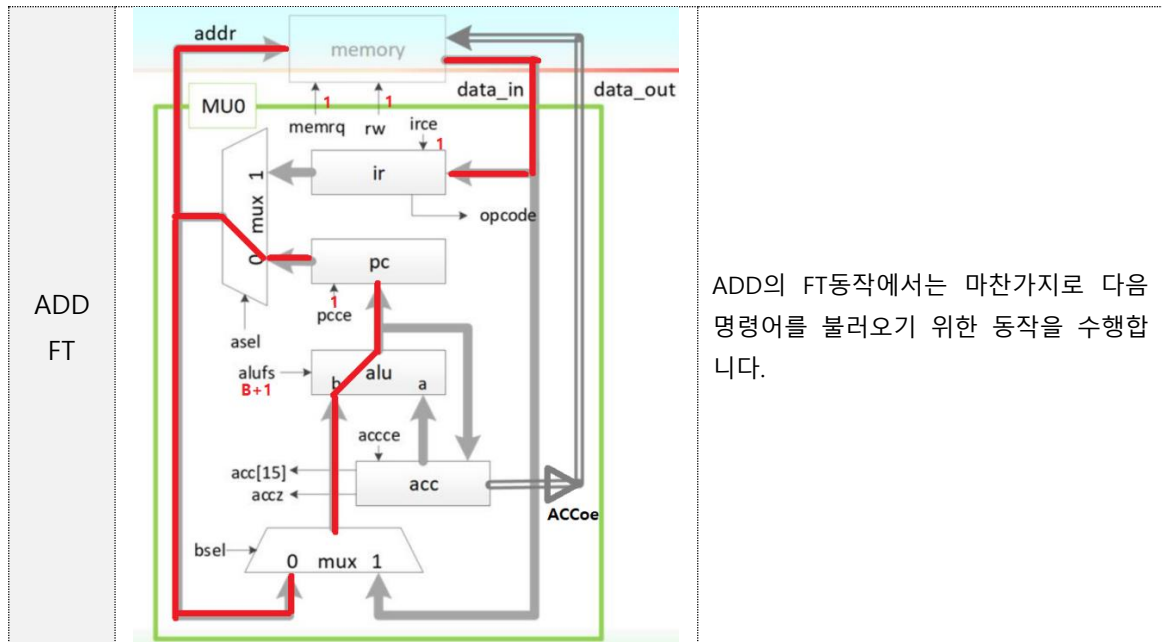
4. Mu0 Processor 설계

프로세서의 동작을 정의하기 위해 각 명령어 별로 data flow를 알 필요성이 있다고 생각하여 각 명령어 별로 어떤 식으로 동작되는지를 도식화 하여 표현하였습니다.

LDA EX		<p>LDA는 메모리의 S 주소에서 Data를 불러와 acc 레지스터에 저장하는 동작을 수행합니다. 먼저 첫 사이클인 EX에서는 S의 주소(addr)가 memory에서 참조되어 data를 불러온 후 b_mux, alu를 거쳐 acc에 저장되어집니다.</p>
LDA FT		<p>LDA에서 값을 acc에 load하였으므로 다음 명령어를 불러와 IR 레지스터에 저장하고 PC를 갱신하는 작업을 해야 합니다. PC에 저장될 때에는 다음 명령이어야 하므로 B+1의 작업을 수행합니다.</p>
STO EX		<p>STO 명령은 acc의 값을 memory에 값을 저장하는 동작을 수행합니다. 첫 사이클에서는 ACCoe가 1이 되어 data_out이 memory에 저장되는 동작을 수행합니다. 여기서 rw는 0(write)가 됩니다.</p>

<p>STO FT</p>		<p>STO의 두번째 사이클인 FT에서는 명령을 수행한 후에 다음 명령을 IR 레지스터에 저장하고 PC를 갱신하는 역할을 수행합니다. LDA FT 동작과 동일합니다.</p>
<p>SUB EX</p>		<p>SUB 명령은 acc 레지스터에 저장된 값에다가 memory 주소 s에 있는 값을 불러와 뺄셈을 수행하는 역할을 합니다. 그러기 위해선 alufs 동작이 A-B가 되도록 하여 acc에 저장된 값(a)과 메모리에서 불러온 값(b)를 빼어 다시 acc에 저장합니다.</p>
<p>SUB FT</p>		<p>SUB FT 동작에서는 PC를 갱신합니다. LDA와 STO 명령과 같이 갱신하는 동작은 동일합니다.</p>

<p>JNE ACCz0</p>		<p>JNE 동작에서는 ACCz0의 값이 중요합니다. ACCz0은 acc에 있는 값이 모두 0 일 때 1이 되는데, S와 N이 같은 지 다른 지를 검증하기 위해 사용되어집니다. 0일 때는 다르다고 판단하여 현재 레지스터인 IR에서 동작을 수행합니다.</p>
<p>JNE ACCz1</p>		<p>JNE 동작에서는 ACCz0의 값이 중요합니다. ACCz0은 acc에 있는 값이 모두 0 일 때 1이 되는데, S와 N이 같은 지 다른 지를 검증하기 위해 사용되어집니다. 1일 때는 같은 값이라고 판단하여 다음 동작을 수행합니다. 다음 동작은 PC에 있는 값이 불러와 다음 동작으로 넘어가게 됩니다.</p>
<p>ADD EX</p>		<p>ADD 명령어는 ir에 있는 명령어의 S주소에 있는 값을 불러와 acc의값과 더하여 그 결과를 다시 acc에 저장하는 역할을 수행합니다.</p>



<Verilog code_mu0_process>

```

1 module mu0_process(clk, rst, memrq, rnw, addr, data_in, data_out);
2
3     parameter ADDR_LENGTH = 12;
4     parameter DATA_LENGTH = 16;
5
6     parameter EX = 1'b0, FT=1'b1;
7     parameter LDA=4'b0000, STO=4'b0001, ADD=4'b0010, SUB=4'b0011,
8       JMP=4'b0100, JGE=4'b0101, JNE=4'b0110, STP=4'b0111;
9
10    input clk, rst;
11    input [DATA_LENGTH-1 : 0] data_in;
12
13    output reg memrq, rnw;
14    output reg[ADDR_LENGTH-1 : 0] addr;
15    output reg [DATA_LENGTH-1 : 0] data_out;
16
17    reg[DATA_LENGTH-1 : 0] alu, acc, b_out, ir;
18    reg[ADDR_LENGTH-1 : 0] pc;
19    reg ps, ns;
20    reg[3:0] opcode;
21    reg[2:0] ALUfs;
22    reg Asel, Bsel, ACCce, PCce, IRce, ACCoe;
23    reg acc_msb, accz;
24
25    assign data_out = ACCoe ? acc : 16'bz;
26
27    always @(posedge clk, posedge rst) begin
28        if(rst) ps<=EX;
29        else ps<=ns;
30    end
31
32    always @(*) begin
33        if (ps==EX)
34            casex(opcode)
35                4'b00xx : ns<=FT;
36                4'b01xx : ns<=EX;
37                default : ns<=EX;
38            endcase
39        else ns<=EX;
40    end

```

(Line): Remark

(3~4): Address를 12bit, Data 길이를 16bit로 설정하도록 parameter로 정의합니다.

(6): Execute과 Fetch state를 각각 0, 1로 설정합니다.

(7~8): 8개의 명령어에 대해 opcode를 할당합니다.

(10~15): mu0_processor의 input과 output값을 각각 선언합니다.

(17~23): mu0_processor 내부 unit에 필요한 pin들을 선언합니다.

(27~40): processor의 FSM을 coding합니다. (initial state= Execute)

```

41  always @(*) begin
42      if (ps==EX)
43          case(opcode)
44              LDA : {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw}
45                  <= 11'b1_1_1_0_0_0_011_1_1;
46              STO : {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw}
47                  <= 11'b1_x_0_0_0_1_000_1_0;
48              ADD : {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw}
49                  <= 11'b1_1_1_0_0_0_001_1_1;
50              SUB : {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw}
51                  <= 11'b1_1_1_0_0_0_010_1_1;
52              JMP : {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw}
53                  <= 11'b1_0_0_1_1_0_100_1_1;
54              JGE : if(acc_msb) {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw}
55                      <= 11'b0_0_0_1_1_0_100_1_1;
56                      else {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw}
57                          <= 11'b1_0_0_1_1_0_100_1_1;
58              JNE : if(accz) {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw}
59                      <= 11'b0_0_0_1_1_0_100_1_1;
60                      else {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw}
61                          <= 11'b1_0_0_1_1_0_100_1_1;
62              STP : {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw}
63                  <= 11'b1_x_0_0_0_0_000_0_1;
64              default : {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw}
65                      <= 11'b0_0_1_1_1_0_000_1_1;
66          endcase
67      else if (ps==FT) {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw}
68          <= 11'b0_0_0_1_1_0_100_1_1;
69      else {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw}
70          <= 11'b0_0_0_1_1_0_000_1_1;
71  end
72  //IR
73  always @ (posedge clk) begin
74
75      if(rst)  ir<=0;
76      else begin
77          if(IRce) begin ir<=data_in; opcode<=data_in[15:12];end
78          else ir<=ir;
79      end
80  end

```

(41~71): Present state가 Execute일 때, 각각의 명령어에 대해 processor 내부 unit들이 어떻게 동작하는지 FSM의 input, output Table값을 참조하여 작성합니다. 이 때, Load, Store, Add, Subtract 명령어는 Execute, Fetch state에 따라 다르게 동작하므로 present state가 fetch일 경우에도 else if 문으로 구분하여 작성합니다.

(72~80): Instruction Register의 동작을 작성합니다.

```

165 //ACC
166 always @ (posedge clk) begin
167     if(rst) acc<=0;
168     else begin
169         if(acc_ce) begin
170             acc<=alu;
171             acc_msb<=alu[15];
172
173             if(alu==0) acc_z<=1;
174             else acc_z<=0;
175         end
176         else acc<=acc;
177     end
178 end
179 //LDSA
180 always @ (posedge clk) begin
181     if(rst) s<=0;
182     else begin
183         if(!inc_en) begin
184             if(s_en) begin
185                 s<=alu;
186             end
187             else s<=s;
188         end
189         else s<=s+16'h0001;
190     end
191 end
192 //SUM
193 always @ (posedge clk) begin
194     if(sum_rst) sum<=0;
195     else begin
196         if(sum_en) sum<=sum+s;
197         else sum<=sum;
198     end
199 end

```

(165~178): Accumulator의 동작을 작성합니다.

ACCce(change enable)이 1일 경우에만 ACC에 ALU의 값을 할당하고, ACC의 15번째 bit(최상위 bit)를 저장합니다.

(179~191): LDSA의 동작을 작성합니다. inc_en값이 0일 경우에 ALU 또는 S값을 할당하고 enable값이 1인 경우 s값을 1증가시킵니다.

(192~199): SUM의 동작을 작성합니다. sum_en값이 1일 경우 sum에 s를 더한 값을 할당합니다.

(201~210): LDN의 동작을 작성합니다. enable값이 1일 경우, n에 ALU값을 할당합니다.

(212~222): ALU의 동작을 작성합니다. 표와 같이 ALU_fs값에 따라 수행하는 동작이 달라집니다.

Operation	ALU_fs
0	000
A+B	001
A-B	010
B	011
B+1	100

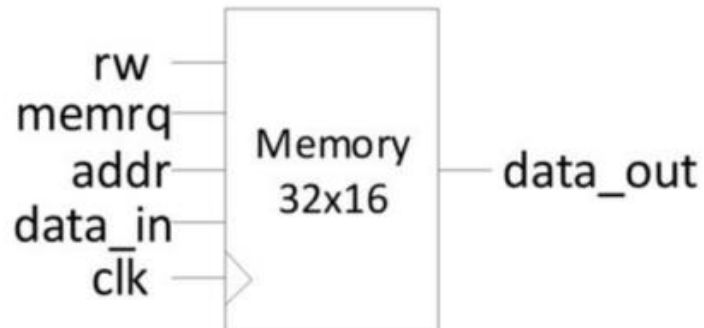
```

201 //LDN
202 always @ (posedge clk) begin
203     if(rst) n<=0;
204     else begin
205         if(n_en) begin
206             n<=alu;
207         end
208         else n<=n;
209     end
210 end
211
212 //ALU
213 always @ (*) begin
214     case(alu_fs)
215         3'b000 : alu<=0; //0
216         3'b001 : alu<=acc+b_out; //A+B
217         3'b010 : alu<=acc-b_out; //A-B
218         3'b011 : alu<=b_out; //B
219         3'b100 : alu<=b_out+16'b0000_0000_0000_0001; //B+1
220         default : alu<=0;
221     endcase
222 end
223
224 //A-MUX
225 always @ (*) begin
226     if(a_sel) a_out<=ir[11:0];
227     else a_out<=pc;
228 end
229
230 //B-MUX
231 always @ (*) begin
232     if(b_sel) b_out<=databus;
233     else b_out<={4'b0000, a_out}; //later test except 0000
234 end
235
236 endmodule

```

(224~234): A, B mux의 동작을 작성합니다. 각각의 select 신호에 대해 1일 때 instruction 또는 data값을 할당합니다.

5. Mu0 Memory 설계



메모리는 instruction과 data를 저장한다.

프로세서는 메모리에 저장된 명령어를 가져와서 수행하기까지 한 state에서 두 state가 소요된다.

1) Input

- clk: positive-edge일때 신호
- memrq: 1bit의 memory enable
- Rw: 1bit의 read(1), write(0)
- Addr: 12bit의 메모리 주소
- Data_in: 16bit의 데이터입력

2) Output

- Data_out: 16bit의 출력

3) Memory Size

- 16bit size 32개 memory

<Verilog code_mu0_memory>

```

1 module mu0_memory(clk, memrq, rnw, addr, data_in, data_out);
2
3     parameter ADDR_LENGTH = 12;
4     parameter DATA_LENGTH = 16;
5     parameter MEM_LENGTH = 32;
6
7     input clk, memrq, rnw;
8     input [ADDR_LENGTH-1 : 0] addr;
9     input [DATA_LENGTH-1 : 0] data_out;
10
11     output reg [DATA_LENGTH-1 : 0] data_in;
12     reg [DATA_LENGTH-1 : 0] memory [MEM_LENGTH-1 : 0];
13
14     assign data_in = (memrq && rnw) ? memory[addr] : 16'bz;
15
16     always@(posedge clk) begin
17         if(memrq) begin
18             if(!rnw) memory[addr] <= data_out;
19             else data_in <= memory[addr];
20         end
21         else memory[addr] <= memory[addr];
22     end
23 endmodule

```

(Line): Remark

(3~5): Address(12bit),
Data(16bit)와 Memory 개수(32
개)를 parameter로 선언합니다.

(7~12): Mu0 memory의 input,
output port를 선언하고
memory를 16bitx32 size로 선
언합니다.

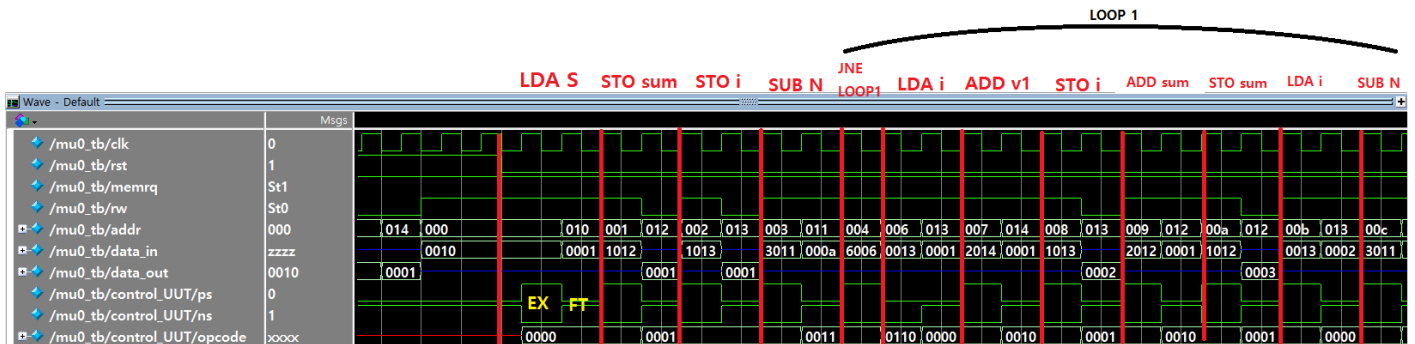
(16~22): data_out일 때
memory에 값을 입력, data_in
일 때 memory로부터 값을 가
져오는 동작을 작성합니다.

6. control logic 회로설계

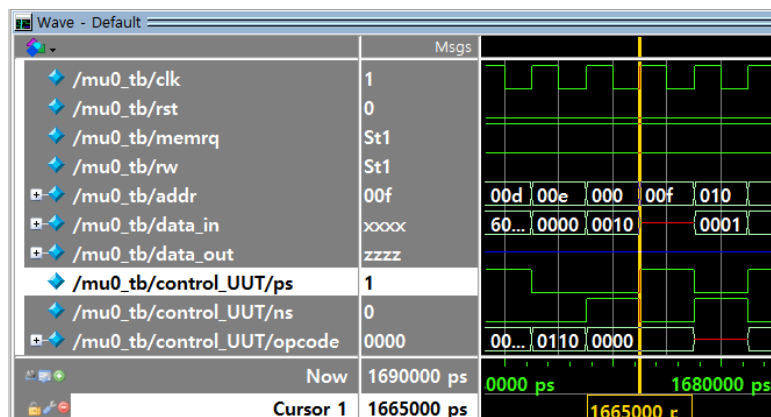
Inputs						Outputs									
Instruction	Opcode	Ex/ft	ACC15	ACCz	Reset	Bsel	PCce	ACCoe	MEMrq	Ex/ft	ALUfs	RnW	Asel	ACCce	IRce
Reset	xxxx	1	x	x	x	0	0	1	1	1	0	= 0	1	1	0
LDA S	0000	0	0	x	x	1	1	1	0	0	0	= B	1	1	1
	0000	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
STO S	0001	0	0	x	x	1	x	0	0	0	1	x	1	0	1
	0001	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
ADD S	0010	0	0	x	x	1	1	1	0	0	0	A+B	1	1	1
	0010	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
SUB S	0011	0	0	x	x	1	1	1	0	0	0	A-B	1	1	1
	0011	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
JMP S	0100	0	x	x	x	1	0	0	1	1	0	B+1	1	1	0
JGE S	0101	0	x	x	0	1	0	0	1	1	0	B+1	1	1	0
	0101	0	x	x	1	0	0	0	1	1	0	B+1	1	1	0
JNE S	0110	0	x	0	x	1	0	0	1	1	0	B+1	1	1	0
	0110	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
STOP	0111	0	x	x	x	1	x	0	0	0	0	x	0	1	0

프로세서 모듈과 메모리 모듈 사이를 연결해주기 위한 하나의 시스템으로 프로세서에서 수행될 연산의 opcode값이 정해지면 위와 같은 기능(output)을 수행합니다.

7. Testbench 동작 검증



Rst이 0으로 바뀌면서 동작이 실행됩니다. 첫 명령은 메모리 0번지에 있는 LDA S 명령이 실행됩니다. 총 2cycle에 걸쳐 Ex, Ft 동작이 실행된 후 다음 동작인 STO sum이 실행됩니다. 그렇게 순차적으로 SUB N 명령까지 실행되고 나면 JNE LOOP1 명령을 만나는데 이 jump 계열 명령은 한 사이클에 이루어지며 LOOP1 번지로 가서 명령을 인출합니다. LOOP 안에 명령들이 조건(S=N)에 만족할 때까지 Loop를 수행하게 됩니다. Test bench에서 의도한대로 ps, ns가 FSM에 맞게 출력되었고, 각 사이클별로 명령들이 맞게 동작함을 확인할 수 있었습니다.



총 1665ns 시점에서 모든 Loop가 돌아지며 결과값이 도출되어집니다. (한 사이클 당 10ns) 결과값을 확인한 결과

Memory Data - /mu0_tb/memort_UUT/memory - Default									
0000001f	x	x	x	x	x	x	x	x	x
00000017	x	x	x	1	10	55	10	1	
0000000f	x	0	24582	12305	19	4114	8210	4115	
00000007	8212	19	0	24582	12305	4115	4114	16	

메모리 영역 확인했을 때 sum의 메모리 주소인 18번째자리에 1~10까지의 합인 55가 도출됨을 알 수 있었습니다. (decimal로 변환하여 표현하였습니다.)

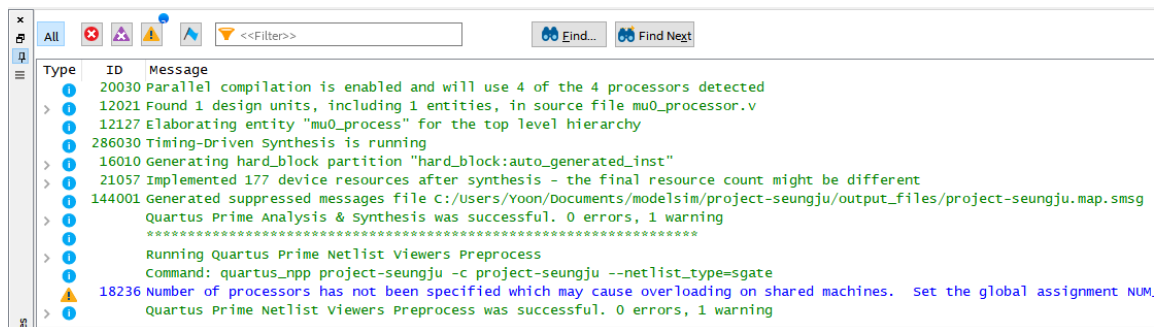
8. Synthesis & RTL viewer

설계한 processor가 Synthesis한 지 검증하기 위해 Quartus로 돌려본 결과 몇 군데 오류가 뜨는 것을 알 수 있었다. 그러기 위해 밑에 부분의 코드를

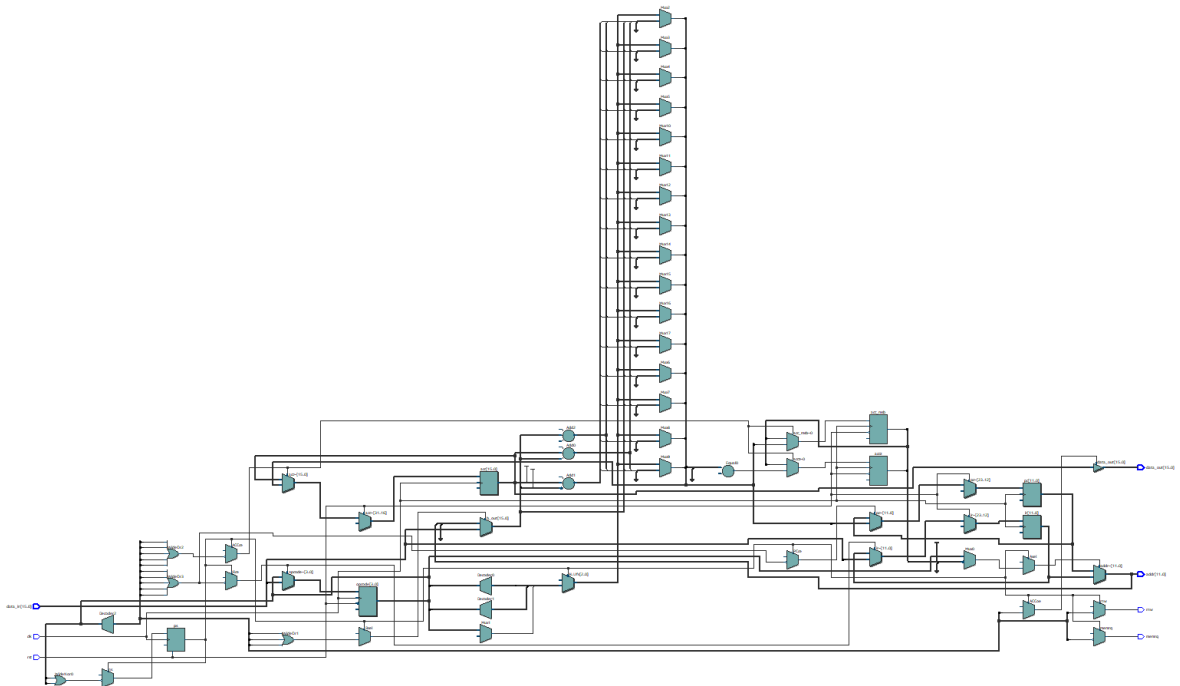
```
assign data_out = ACCCoe ? acc : 16'bz;
```

always문으로 따로 써주어 Synthesis 가능한 코드로 변경 후 합성을 진행하였다.

```
always @(*) begin
    data_out = ACCCoe ? acc : 16'bz;
end
```



알맞게 합성이 진행되었으며 아래 logic처럼 잘 합성되는 것을 알 수 있었다.



<Verilog code_mu0_tb>

```
1 `timescale 1ns/100ps
2 module mu0_tb;
3     //opcode
4     parameter LDA = 4'b0000, STO = 4'b0001, ADD = 4'b0010, SUB = 4'b0011;
5     parameter JMP=4'b0100, JGE = 4'b0101, JNE = 4'b0110, STP = 16'b0111_0000_0000_0000;
6     //address
7     parameter S = 12'h010, N=12'h011, sum=12'h012, i=12'h013, v1=12'h014;
8     parameter loop1 =12'h006;
9     //parameter
10    parameter ADDR_LENGTH = 12;
11    parameter DATA_LENGTH = 16;
12
13    reg clk, rst;
14    wire memrq, rw;
15    wire [ADDR_LENGTH-1:0] addr;
16    wire [DATA_LENGTH-1:0] data_in;
17    wire [DATA_LENGTH-1:0] data_out;
18
19    mu0_memory memort_UUT (.clk(clk), .memrq(memrq), .rnw(rw), .addr(addr),
20        .data_in(data_in), .data_out(data_out));
21    mu0_process control_UUT (.clk(clk), .rst(rst), .memrq(memrq), .rnw(rw),
22        .addr(addr), .data_in(data_in), .data_out(data_out));
23
24    initial begin
25        clk=0; rst=1;
26        #200 rst=0;
27    end
28
29    always begin #5 clk = ~clk; end
```

(Line): Remark

(4~11): 8 개의 명령어 opcode 와 S, N, sum, i, v1, loop1 의 주소값을 할당합니다. Address 를 12bit, Data 를 16bit 로 설정합니다.

(13~17): wave 파형에 출력될 input, output port 를 선언합니다.

(19~22): Mu0 processor 와 Mu0 Memory 를 testbench 에 연결합니다.

(24~29): 초기 clk 과 rst 신호를 설정합니다. 최초 200ns 까지 rst 신호를 1 로 주어 processor 의 동작을 막고, memory 에 값을 전부 할당한 뒤, 200ns 이후 rst 을 0 으로 설정하여 processor 와 memory 사이의 동작 결과를 관측합니다. clk cycle 은 10ns 을 주기로 설정합니다.

```

31  initial begin // instruction
32      force data_out = {LDA, S}; force addr=0; force rw=0; force memrq=1;
33      #10 release data_out; release addr;
34      force data_out = {STO, sum}; force addr=1;
35      #10 release data_out; release addr;
36      force data_out = {STO, i}; force addr=2;
37      #10 release data_out; release addr;
38      force data_out = {SUB, N}; force addr=3;
39      #10 release data_out; release addr;
40      force data_out = {JNE, loop1}; force addr=4;
41      #10 release data_out; release addr;
42      force data_out = {STP, 12'b0000_0000_0000}; force addr=5;
43      #10 release data_out; release addr;
44      force data_out = {LDA, i}; force addr=6;
45      #10 release data_out; release addr;
46      force data_out = {ADD, v1}; force addr=7;
47      #10 release data_out; release addr;
48      force data_out = {STO, i}; force addr=8;
49      #10 release data_out; release addr;
50      force data_out = {ADD, sum}; force addr=9;
51      #10 release data_out; release addr;
52      force data_out = {STO, sum}; force addr=10;
53      #10 release data_out; release addr;
54      force data_out = {LDA, i}; force addr=11;
55      #10 release data_out; release addr;
56      force data_out = {SUB, N}; force addr=12;
57      #10 release data_out; release addr;
58      force data_out = {JNE, loop1}; force addr=13;
59      #10 release data_out; release addr;
60      force data_out = {STP, 12'b0000_0000_0000}; force addr=14;
61      #10 release data_out; release addr;
62      //S, N, data input
63      force data_out = 16'h0001; force addr=S;
64      #10 release data_out; release addr;
65      force data_out = 16'h000A; force addr=N;
66      #10 release data_out; release addr;
67      force data_out = 16'h0001; force addr=v1;
68      #10 release data_out; release addr; release rw; release memrq; end
69  endmodule

```

(31~68): Task 실행 과정을 작성합니다. 작성한 code 는 i 가 S 일 때부터 N 이 될 때 까지의 합을 구하는 Task 가 됩니다. testbench 에 module 2 개를 연결하였기 때문에 force & release 를 이용한 방식으로 값을 할당하고 해제하고 있습니다. Task 를 수행하는 동안 총합 170ns 가 걸리는데, 200ns 이전까지는 rst=1 로 processor 의 개입을 막고 있기 때문에 memory 에 값을 정상적으로 넣을 수 있습니다.

9. 전체 Source Code (Verilog)

Table 1. mu0_process

```
module mu0_process(clk, rst, memrq, rnw, addr, data_in, data_out);

    parameter ADDR_LENGTH = 12;
    parameter DATA_LENGTH = 16;

    parameter EX = 1'b0, FT=1'b1;
    parameter LDA=4'b0000, STO=4'b0001, ADD=4'b0010, SUB=4'b0011,
              JMP=4'b0100, JGE=4'b0101, JNE=4'b0110, STP=4'b0111;

    input clk, rst;
    input [DATA_LENGTH-1 : 0] data_in;

    output reg memrq, rnw;
    output reg[ADDR_LENGTH-1 : 0] addr;
    output reg [DATA_LENGTH-1 : 0] data_out;

    reg[DATA_LENGTH-1 : 0] alu, acc, b_out, ir;
    reg[ADDR_LENGTH-1 : 0] pc;
    reg ps, ns;
    reg[3:0] opcode;
    reg[2:0] ALUfs;
    reg Asel, Bsel, ACCce, PCce, IRce, ACCoe;
    reg acc_msb, accz;

    assign data_out = ACCoe ? acc : 16'bz;

    always @(posedge clk, posedge rst) begin
        if(rst) ps<=EX;
        else ps<=ns;
    end

    always @(*) begin
        if (ps==EX)
            casex(opcode)
                4'b00xx : ns<=FT;
                4'b01xx : ns<=EX;
                default : ns<=EX;
            endcase
        else ns<=EX;
    end

    always @(*) begin
        if (ps==EX)
            case(opcode)
                LDA : {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw} <= 11'b1_1_1_0_0_0_011_1_
_1;
                STO : {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw} <= 11'b1_x_0_0_0_1_1_0_0;
                ADD : {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw} <= 11'b1_1_1_0_0_0_001_1_
_1;
                SUB : {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw} <= 11'b1_1_1_0_0_0_010_1_
_1;
                JMP : {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw} <= 11'b1_0_0_1_1_0_100_1_
_1;
                JGE : if(acc_msb) {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw} <= 11'b0_0_0_
1_1_0_100_1_1;
                        else {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw} <= 11'b1_0_0_1_1_0_
100_1_1;
                JNE : if(accz) {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw} <= 11'b0_0_0_1_
1_0_100_1_1;
                        else {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw} <= 11'b1_0_0_1_1_0_
100_1_1;
                STP : {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw} <= 11'b1_x_0_0_0_0_1_0_0_
_1;
                default : {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw} <= 11'b0_0_1_1_1_0_0_
00_1_1;
            endcase
        else if (ps==FT) {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw} <= 11'b0_0_0_1_1_0_100_
1_1;
        else {Asel, Bsel, ACCce, PCce, IRce, ACCoe, ALUfs, memrq, rnw} <= 11'b0_0_0_1_1_0_000_1_1;
    end

    //IR
    always @ (posedge clk) begin

        if(rst) ir<=0;
        else begin
```

```

        if(IRce) begin ir<=data_in; opcode<=data_in[15:12];end
        else ir<=ir;
    end

end

//PC
always @ (posedge clk) begin
    if(rst) pc<=0;
    else begin
        if(PCce) pc<=alu[11:0];
        else pc<=pc;
    end
end

//ACC
always @ (posedge clk) begin
    if(rst) acc<=0;
    else begin
        if(ACCce) begin
            acc<=alu;
            acc_msb<=alu[15];
            if(alu==0) accz<=1;
            else accz<=0;
        end
        if(ACCoe) data_out <= acc;
        else data_out <= data_out;
        end
        else acc<=acc;
        $display("acc value : %b",acc);
    end
end

//ALU
always @ (*) begin
    case(ALUfs)
        3'b000 : alu<=0; //0
        3'b001 : alu<=acc+b_out; //A+B
        3'b010 : alu<=acc-b_out; //A-B
        3'b011 : alu<=b_out; //B
        3'b100 : alu<=b_out+16'b0000_0000_0000_0001; //B+1
        default : alu<=0;
    endcase
end

//A-MUX
always @ (*) begin
    if(Asel) addr<=ir[11:0];
    else addr<=pc;
end

//B-MUX
always @ (*) begin
    if(Bsel) b_out<=data_in;
    else b_out<={4'b0000, addr}; //later test except 0000
end

endmodule

```

Table 2. mu0_memory

```

module mu0_memory(clk, memrq, rnw, addr, data_in, data_out);

    parameter ADDR_LENGTH = 12;
    parameter DATA_LENGTH = 16;
    parameter MEM_LENGTH = 32;

    input clk, memrq, rnw;
    input [ADDR_LENGTH-1 : 0] addr;
    input [DATA_LENGTH-1 : 0] data_out;

    output reg [DATA_LENGTH-1 : 0] data_in;
    reg [DATA_LENGTH-1 : 0] memory [MEM_LENGTH-1 : 0];

    assign data_in = (memrq && rnw) ? memory[addr] : 16'bz;

    always@(posedge clk) begin
        if(memrq) begin
            if(!rnw) memory[addr] <= data_out;
            else data_in <= memory[addr];end
        else memory[addr] <= memory[addr];
    end
endmodule

```

```

end
endmodule

```

Table 3. mu0_tb

```

`timescale 1ns/100ps
module mu0_tb;
    //opcode
    parameter LDA = 4'b0000, STO = 4'b0001, ADD = 4'b0010, SUB = 4'b0011;
    parameter JMP=4'b0100, JGE = 4'b0101, JNE = 4'b0110, STP = 16'b0111_0000_0000_0000;
    //address
    parameter S = 12'h010, N=12'h011, sum=12'h012, i=12'h013, v1=12'h014;
    parameter loop1 =12'h006;
    //parameter
    parameter ADDR_LENGTH = 12;
    parameter DATA_LENGTH = 16;

    reg clk, rst;
    wire memrq, rw;
    wire [ADDR_LENGTH-1:0] addr;
    wire [DATA_LENGTH-1:0] data_in;
    wire [DATA_LENGTH-1:0] data_out;

    mu0_memory memort_UUT (.clk(clk), .memrq(memrq), .rnw(rw), .addr(addr), .data_in(data_in), .data_out
(data_out));
    mu0_process control_UUT (.clk(clk), .rst(rst), .memrq(memrq), .rnw(rw), .addr(addr), .data_in(data_in
), .data_out(data_out));

    initial begin
        clk=0; rst=1;
        #200 rst=0;
    end

    always begin #5 clk = ~clk; end

    initial begin
        // instruction
        force data_out = {LDA, S}; force addr=0; force rw=0; force memrq=1;
        #10 release data_out; release addr;
        force data_out = {STO, sum}; force addr=1;
        #10 release data_out; release addr;
        force data_out = {STO, i}; force addr=2;
        #10 release data_out; release addr;
        force data_out = {SUB, N}; force addr=3;
        #10 release data_out; release addr;
        force data_out = {JNE, loop1}; force addr=4;
        #10 release data_out; release addr;
        force data_out = {STP, 12'b0000_0000_0000}; force addr=5;
        #10 release data_out; release addr;
        force data_out = {LDA, i}; force addr=6;
        #10 release data_out; release addr;
        force data_out = {ADD, v1}; force addr=7;
        #10 release data_out; release addr;
        force data_out = {STO, i}; force addr=8;
        #10 release data_out; release addr;
        force data_out = {ADD, sum}; force addr=9;
        #10 release data_out; release addr;
        force data_out = {STO, sum}; force addr=10;
        #10 release data_out; release addr;
        force data_out = {LDA, i}; force addr=11;
        #10 release data_out; release addr;
        force data_out = {SUB, N}; force addr=12;
        #10 release data_out; release addr;
        force data_out = {JNE, loop1}; force addr=13;
        #10 release data_out; release addr;
        force data_out = {STP, 12'b0000_0000_0000}; force addr=14;
        #10 release data_out; release addr;
        //S, N, data input
        force data_out = 16'h0001; force addr=S;
        #10 release data_out; release addr;
        force data_out = 16'h000A; force addr=N;
        #10 release data_out; release addr;
        force data_out = 16'h0001; force addr=v1;
        #10 release data_out; release addr; release rw; release memrq;
    end
endmodule

```