컴퓨터구조 Report

Project #0 - MU0 Processor

담당교수: 이성원 교수님

실습분반: 수 7교시

학번: 2019202050

이름: 이강현

[Introduction]

이번 프로젝트는 간단한 microprocessor로 설계된 MU0를 구현해보는 것이다. MU0는 16bit processor로서 4bit는 opcode, 12bit는 address로 구성된 16bit instruction들로 동작하며 명령어들을 간략히 설명하면 다음과 같다.

LDA S(opcode:0000): S에 해당하는 address에서 데이터를 가져와 accumulator에 저장한다.

STO S(opcode:0001): S에 해당하는 address에 accumulator의 데이터를 저장한다.

ADD S(opcode:0010): S에 해당하는 address의 값과 accumulator의 값을 더하여 accumulator에 저장한다.

SUB S(opcode:0011): S에 해당하는 address의 값을 accumulator의 값에서 뺀 후 accumulator에 저장한다.

JMP S(opcode:0100): PC = S

JGE S(opcode:0101): accumulator의 값이 0보다 크거나 같다면 PC = S

JNE S(opcode:0110), accumulator의 값이 0이 아니라면 PC = S

STP(opcode:0111): 프로그램의 실행을 종료한다.

[Assignment]

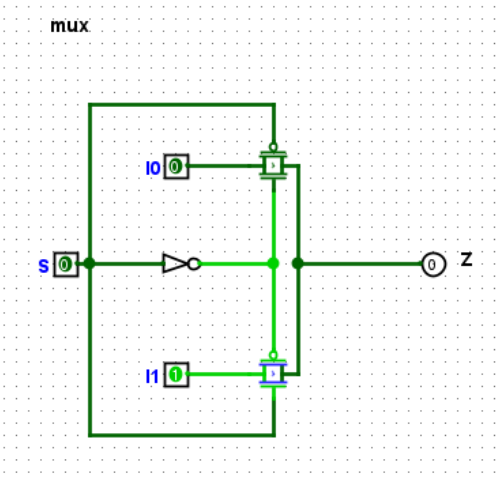
2.1

1. What is the Transmission gate?

input	output
0	0
1	1

Transmission gate란 pmos 1개와 nmos 1개를 연결한 형식으로 입력을 출력으로 전달해주는 게이트이다. Pmos는 입력이 1일 때 값을 전달하고 nmos는 입력이 0일 때 0을 전달해준다.

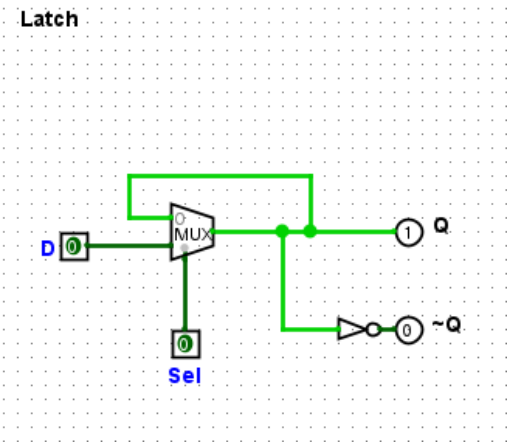
2. Implement a MUX using Transmission gates



S	Z
0	I0
1	I1

Transmission gate 두개를 사용하여 위와 같이 2 to 1 mux를 구현할 수 있다.

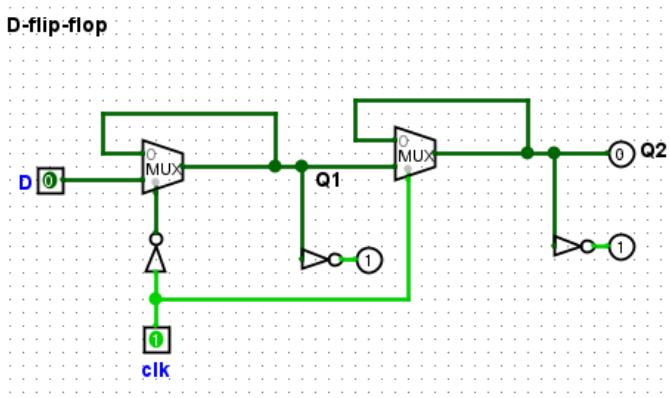
3. Implement a Latch using MUXs.



Sel	D	Q
0	x	이전값
1	0	0
1	1	1

Mux의 output값을 다시 입력으로 넣어주어 D latch를 위와 같이 구현하였다.

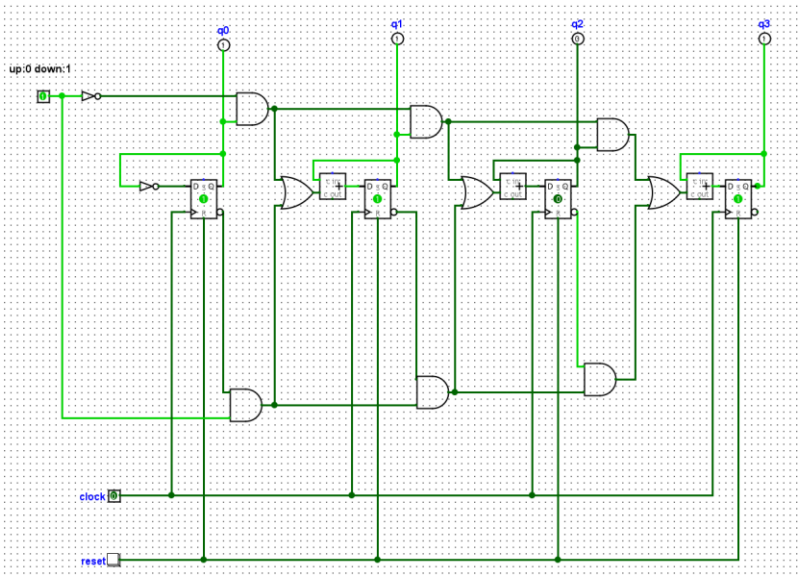
4. Implement D-FF using Latches.



clk	D	Q2
↑	0	1
↑	1	1

D-latch 두 개를 서로 연결하고 mux의 selector를 한쪽은 not게이트를 주어 상승엣지 혹은 하강엣지로 작동할 수 있게끔 하면 위와 같은 D-flip-flop을 구현할 수 있다.

5. Implement Synchronous Up/Down Counter using D-FFs and adders.



counter	q3	q2	q1	q0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1

8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

up카운트라면 0, down카운트라면 1로 설정 후 클록의 상승엣지마다 1씩 값을 상승시킨다. 위는 카운트마다 변하는 q0,q1,q2,q3의 값들을 나타낸 것이고 down 카운터는 위의 순서의 역순으로 진행된다. adder를 통해 이전 플립플롭의 값과 현재 플립플롭의 값을 더해주어 현재 플립플롭에 넣어주는 방식을 통해 카운터를 구성하였다. sum 값을 이용하기 때문에 현재 플립플롭이 0이라면 1로 1이라면 캐리발생이므로 0으로 바뀌기 때문에 입력값의 반전효과를 얻을 수 있다.

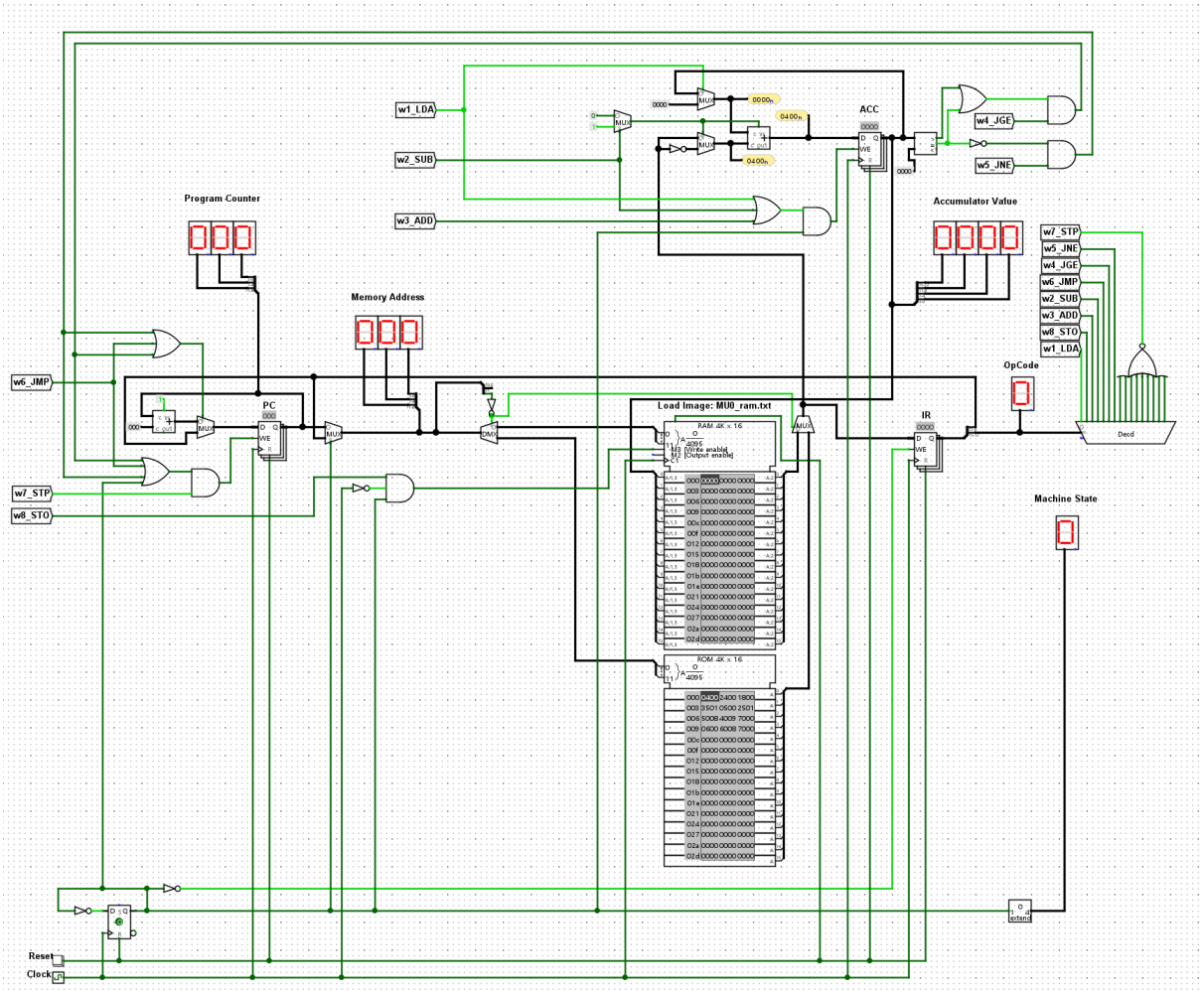
6. Compare Synchronous counter and Asynchronous counter such as a ripple counter.

위에서 설계한 카운터는 동기식 카운터로서 모든 플립플롭들이 하나의 클록과 연결되어있어 동시에 값이 바뀐다는 특징(동기화)이 있으나 리플 카운터라고도 불리는 비동기식 카운터는 하나의 클록에 모두 연결되지 않은 형식을 띈다.

7. What makes that Asynchronous counter is rarely used?

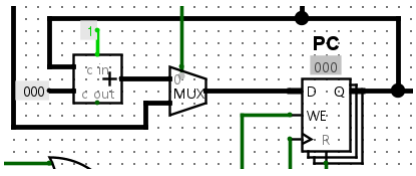
비동기 카운터는 하나의 클록에 의해 각 플립플롭들이 동기화되어있지 않아 각 플립플롭을 통과할 때마다 전달지연이 발생한다는 단점 때문에 잘 사용하지 않는다.

2.2



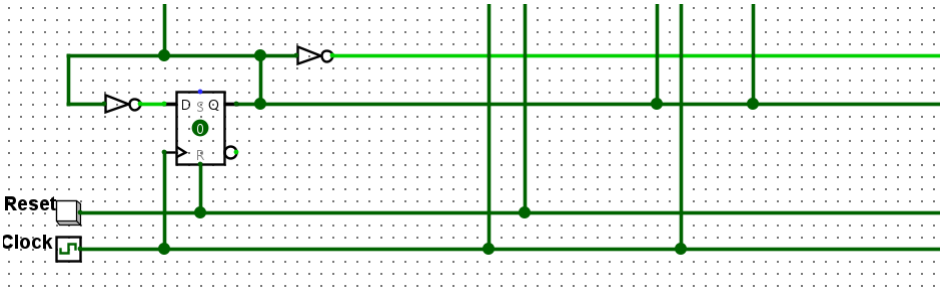
위는 구현한 MU0의 회로도이다.

MU0에서 사용된 주요 회로에 관한 설명이다.

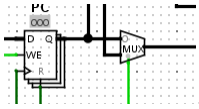


우선 fetch 과정중 pc=pc+1을 하거나 jump를 하게될 때 사용되는 부분이다. adder의 carry in부분에 1이라는 상수를 넣어주어 pc+1을 만들고 mux를 통과시켜 다음 명령을 실행할지 jump할지 결정하게 된다. 위에서 구현한 counter에서 값이 변하게 동작한 adder의 역할과 유사하다.

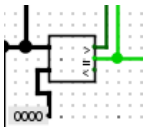
a	b	carry in	sum	carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



다음은 reset과 clock위에 존재하는 d flip flop부분이다. 해당부분은 accumulator나 ram에 값을 저장할 때 원하는 타이밍에 저장하기 위해 사용된다. 이를 테면 accumulator에 연결된 선은 instruction의 주소값과 주소값을 참조하여 가져온 값이 모두 존재할 수 있게 된다. 따라서 write enable의 셋 타이밍을 조절하지 않는다면 값을 저장하지 않고 주소값을 저장하여 올바른 값을 얻지 못할 것이다.

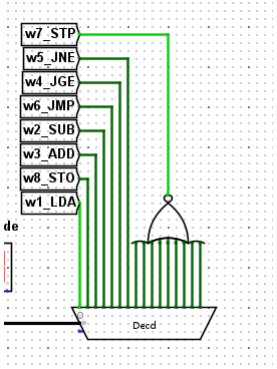


또한 왼쪽과 같이 PC에서 나온 명령어가 업데이트 될 때까지 저장장치의 입력으로 들어가선 안되고 업데이트 될 동안 주소값을 참조하여 저장장치에서 데이터를 뽑아내야 하기 때문에 그 시간을 벌기 위해서도 사용된다.



다음은 comparator이다. 이는 MU0 회로에서 JMP명령어의 conditional한 경우를 고려하기 위해 accumulator의 값을 확인하기 위해 사용되는 회로이다.

위는 입력값 0과 비교하여 0보다 클때는 >부분이 1로 같을때는 =부분이 1로 작을때는 <부분이 1로 바뀐다. <부분을 사용되지 않으므로 끊어져있다.

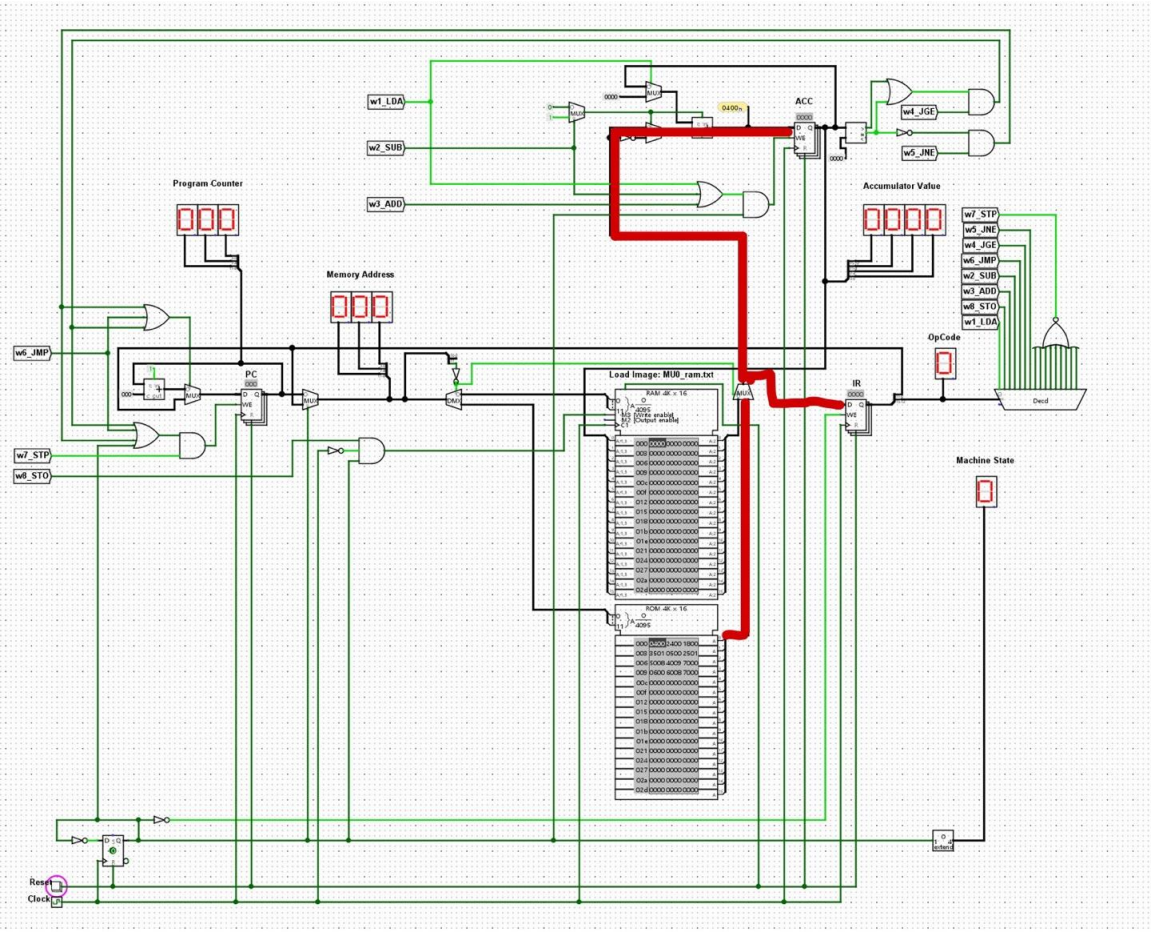


input	output			
00	1	0	0	0
01	0	1	0	0
10	0	0	1	0
11	0	0	0	1

마지막으로 decorder부분이다. decorder는 2bit라고 생각할 때 위와 같은 진리표를 가진다. 따라서 MU0에서는 opcode를 decorder를 통해 4비트 명령어를 16개의 출력으로 바꾸어 tunnel을 활용하여 명령이 작동하게끔 한다. MU0에서 명령해석기로서 작동함을 알 수 있다.

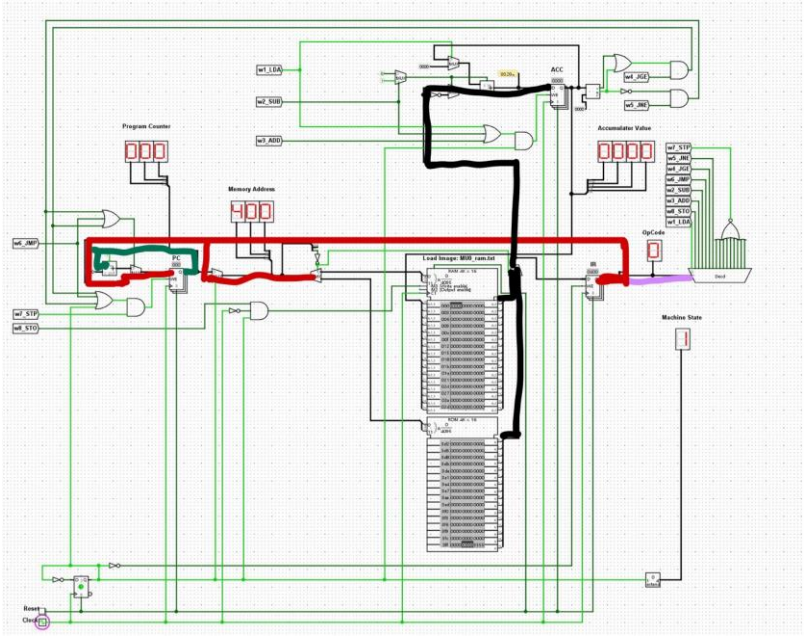
이제부터 MU0회로의 동작 방식을 설명한다.

먼저 명령어를 살펴보기 전 이 회로에서 FETCH와 EXECUTE가 어떻게 이루어지는지 확인해보자. 세 단계로 나누어서 설명한다.

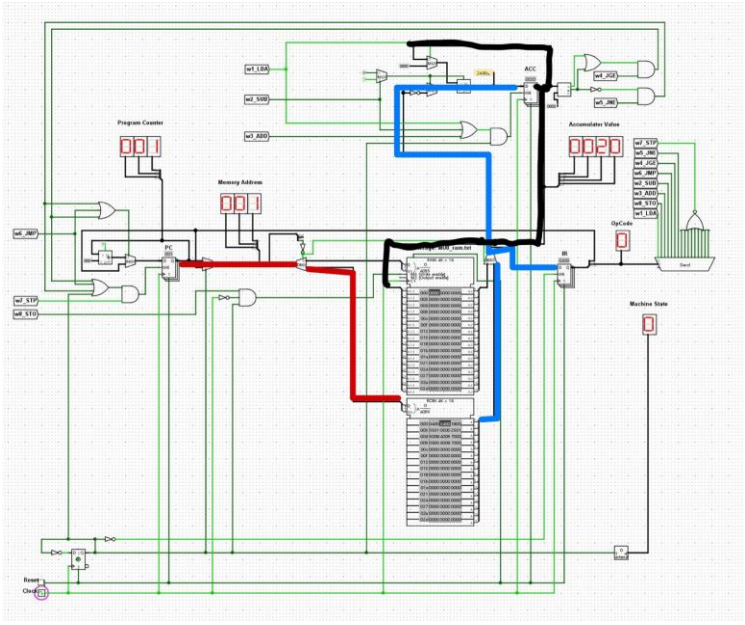


첫번째 단계는 초기상태이다. Reset를 하게 되면 모든 회로가 0으로 초기화되므로

비동기 저장장치는 000주소의 첫번째 명령어를 가리키게 된다. 이때 빨간선으로 표기된 부분에 저장장치의 000주소의 데이터가 위치하게된다. 하지만 아래의 플립플롭에 의해 accumulator의 write enable단자는 control되기 때문에 accumulator의 입력단자에 있는 데이터는 쓰여지지 않는다.



클록의 상승엣지에서 빨간선에 있던 데이터는 IR를 통과하게 되어 상위 4비트는 opcode이므로 디코더로 가게되어 명령어를 해석하고 하위 12비트는 위의 사진과 같이 이동하여 PC의 값을 바꿀 준비를 하는 부분과 저장장치의 해당 주소를 가리키게끔하도록 저장장치의 입력부분으로 연결되어 있다. 이 단계에서 또한 하단의 플립플롭에 의한 control로 저장장치에 입력되는 데이터는 mux로 조절되어 pc의 값이 아닌 IR에서 나온 12비트가 다시 저장장치에 입력으로 들어가게 된다. 주소에 해당하는 데이터는 검은선으로 표기되었다. 이번 단계에서 검은선의 데이터는 이전단계와 다르게 accumulator의 write enable이 1이되어 accumulator에 쓰여지게 된다.



다음으로 검은선의 데이터는 명령어에 따라 accumulator의 add sub명령어를 위해 피연산자로서 역할을 하거나 저장장치의 입력으로서 sto명령시 저장된다. 혹은 lda 명령이 들어오게 되면 파란선의 데이터가 쓰여지게 되어 사용되지 않을 수도 있다.

또한 이단계에서는 이전단계에서 PC값이 1증가한 데이터 혹은 jump를 위해 가져온 주소가 저장장치의 입력으로 이동한다.

그 이후는 이전단계, 명령어를 해석하고 다음 명령어를 결정하는 단계와 현재 단계, 데이터를 accumulator에 쓰거나 accumulator에서 나온 데이터를 저장장치에 저장하거나 그리고 다음 명령어가 입력되어 명령해석을 준비하는 단계 두 단계가 각각 한 클록 사이클을 가지며 번갈아가며 반복동작한다.

이제 각 명령어가 어떻게 작동하는지 설명한다. 명령어 순서는 모든 명령어의 실행과정을 확인할 수 있게 모든 명령어를 사용하게 끔 구성하였으며 명령어부분은

```
000  0400 2400 1800 3501  0500 2501 5008 4009
008  7000 0600 6008 7000  0000 0000 0000 0000
```

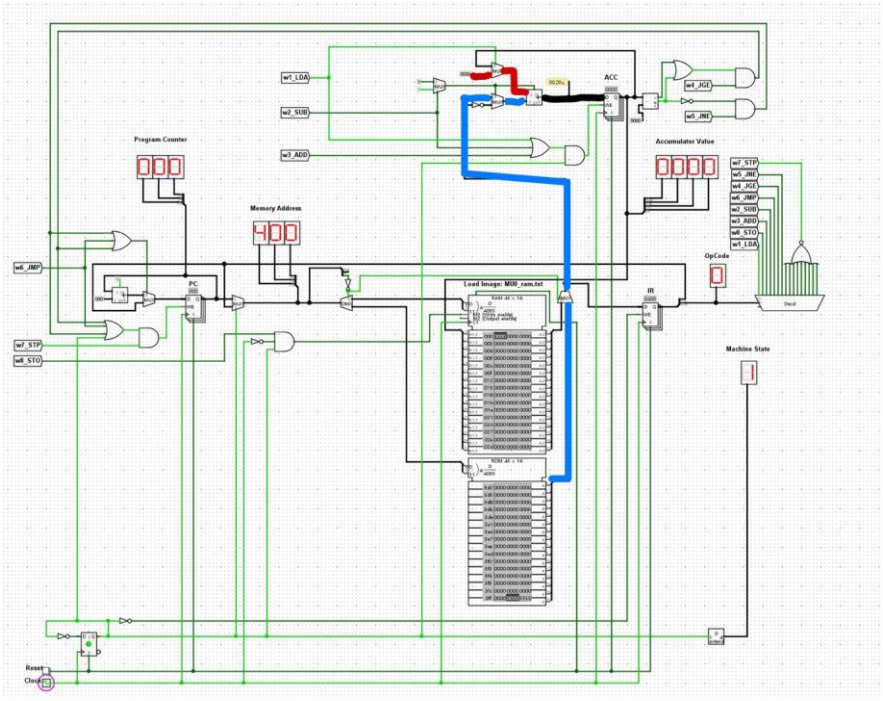
사진과 같고

명령어에서 사용할 데이터가 저장된 부분은 다음과 같다.

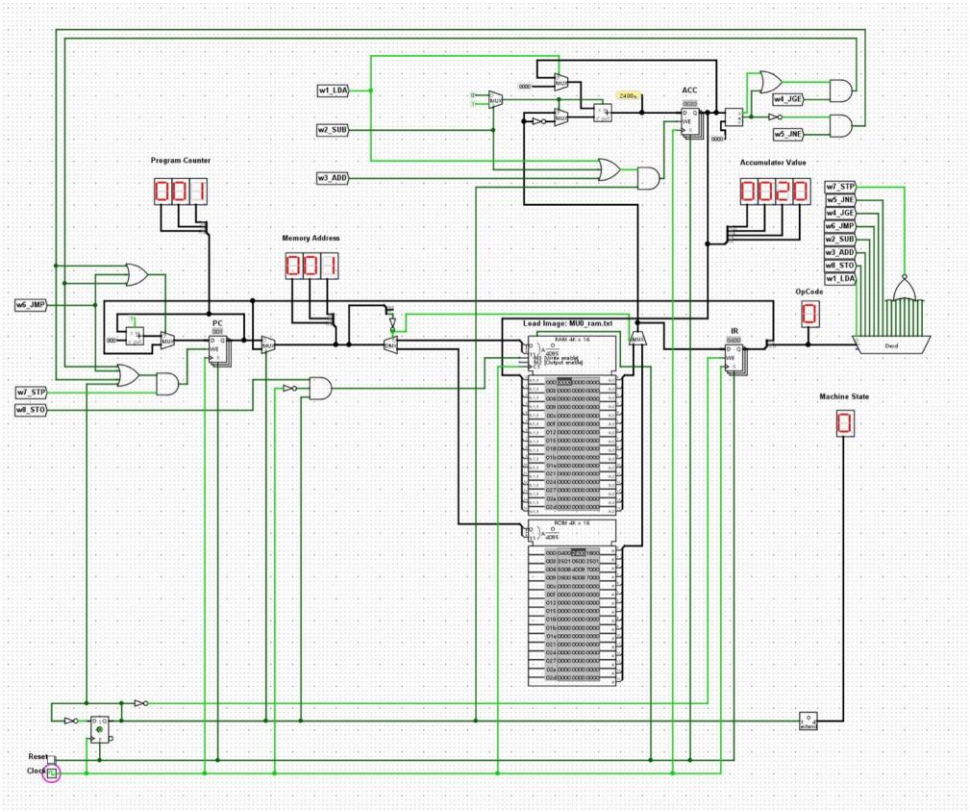
```
400  0020 5555 00aa 0055  0000 0000 0000 0000

500  7fff 0001 0000 0000  0000 0000 0000 0000
```

<LDA> 0400

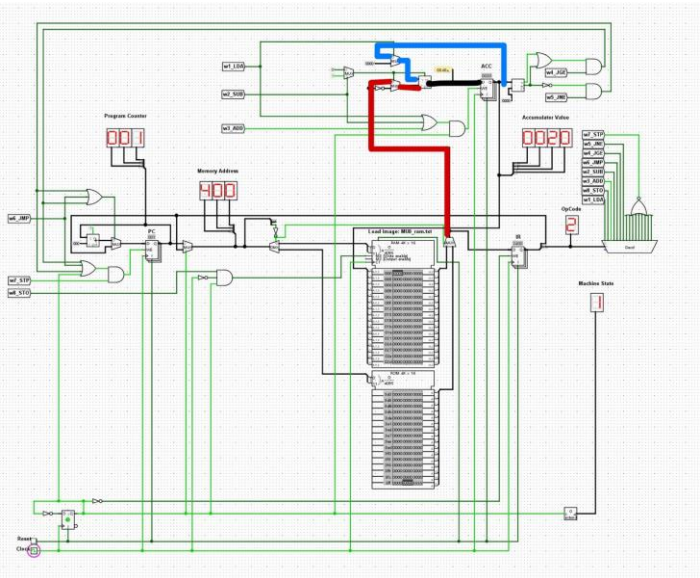


LDA명령어가 해석되면 저장장치는 주소에 맞는 데이터를 파란선으로 보내주고 accumulator의 입력과 연결된 adder로 들어가는 입력을 결정하는 MUX가 0000을 보내어 파란선의 데이터와 더해진다. 0000과 더해지므로 파란선은 adder를 지나도 데이터는 변함이 없고 이는 검은선으로 accumulator에 대기하고 있게 된다.



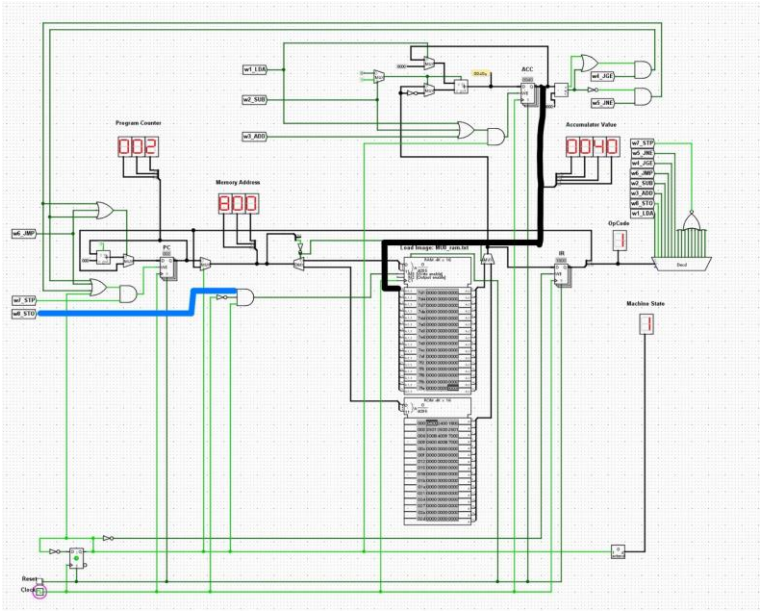
따라서 0400이라는 instruction을 실행한 결과 400의 주소에 있는 20이 저장됨을 알 수 있다.

<ADD> 2400



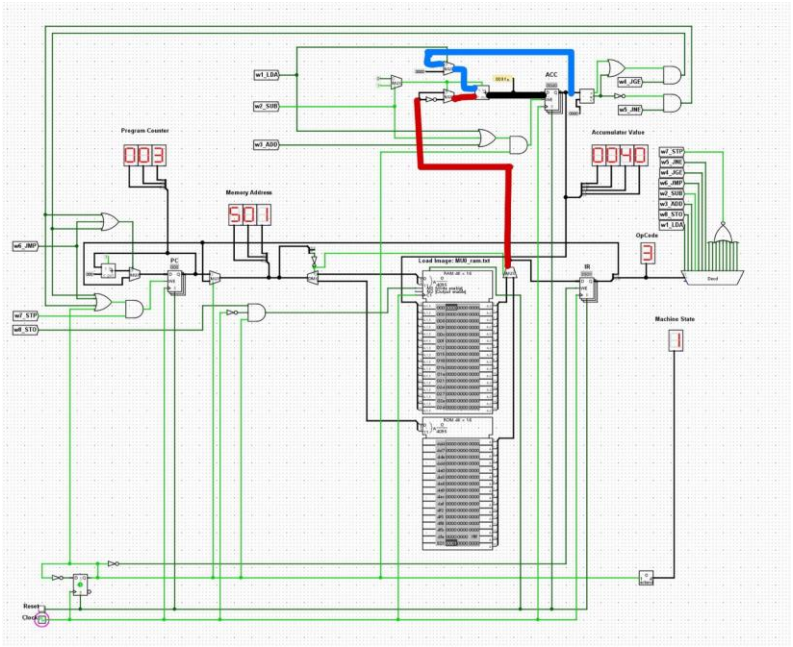
ADD명령이 해석되면 위의 사진과 같이 accumulator의 현재 데이터를 나타내는 파란선이 MUX에서 선택되어 adder에서 현재 저장장치가 가리키고 있는 데이터와 서로 더해진다. 그 후 검은선에서 대기하고 있다가 accumulator에 다음 상승엣지에 기록된다.

<STO> 1800



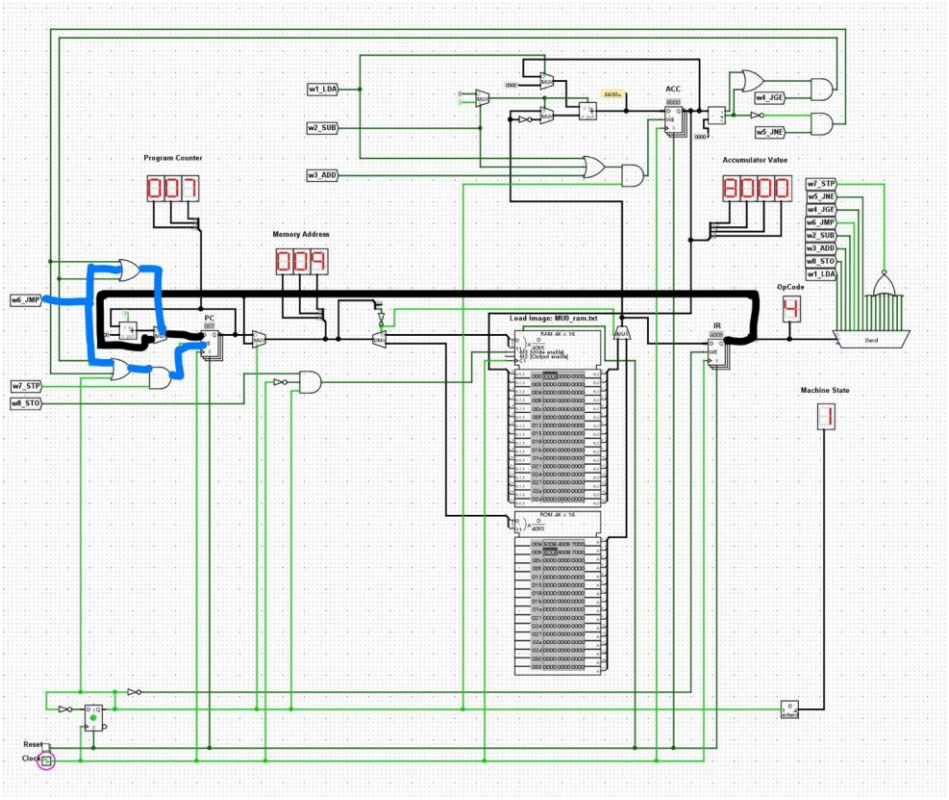
STO 명령이 해석된 후 accumulator의 데이터는 검은선으로 저장장치의 입력에 대기하고 있고 파란선의 단자가 1이되었기 때문에 and 게이트의 입력단자에는 clock과 not게이트로 연결된 부분을 제외하고 모두 1이다. Clock과 연결된 부분이 다시 0으로 변하게 되면 저장장치의 write enable을 1로 만들고 이때 상승엣지를 지나지 않아 accumulator의 값은 유지되나 비동기 저장장치는 write enable이 1이되었으므로 이 타이밍에 저장을 하게 된다.

<SUB> 3501



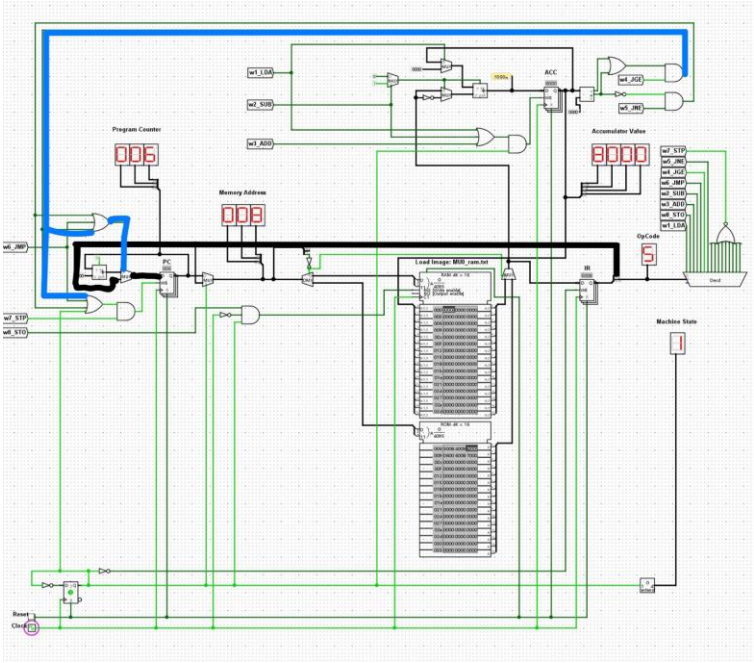
SUB 명령어는 ADD와 거의 비슷하게 동작한다. 한가지 다른 점은 SUB 명령어는 accumulator에 더해줄 값을 not게이트를 통해 반전시키고 1을 더해주는 데이터 처리 과정을 거친다.

<JMP> 4009



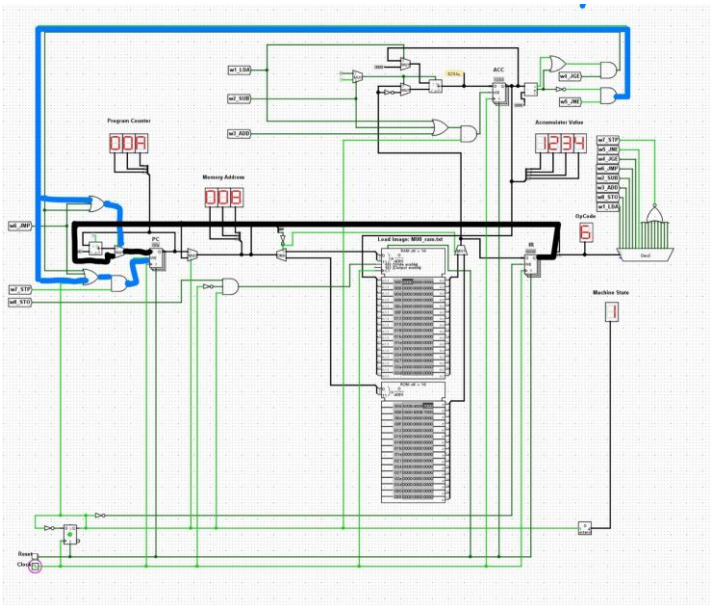
JMP 명령어는 unconditional이므로 무조건 해당 주소로 점프하게 된다. 따라서 다음 명령어를 결정하는 단계에서 pc에 1을 더한 값이 아닌 주소가 mux에게 선택되어 pc에 쓰여지게 된다.

<JGE> 5008



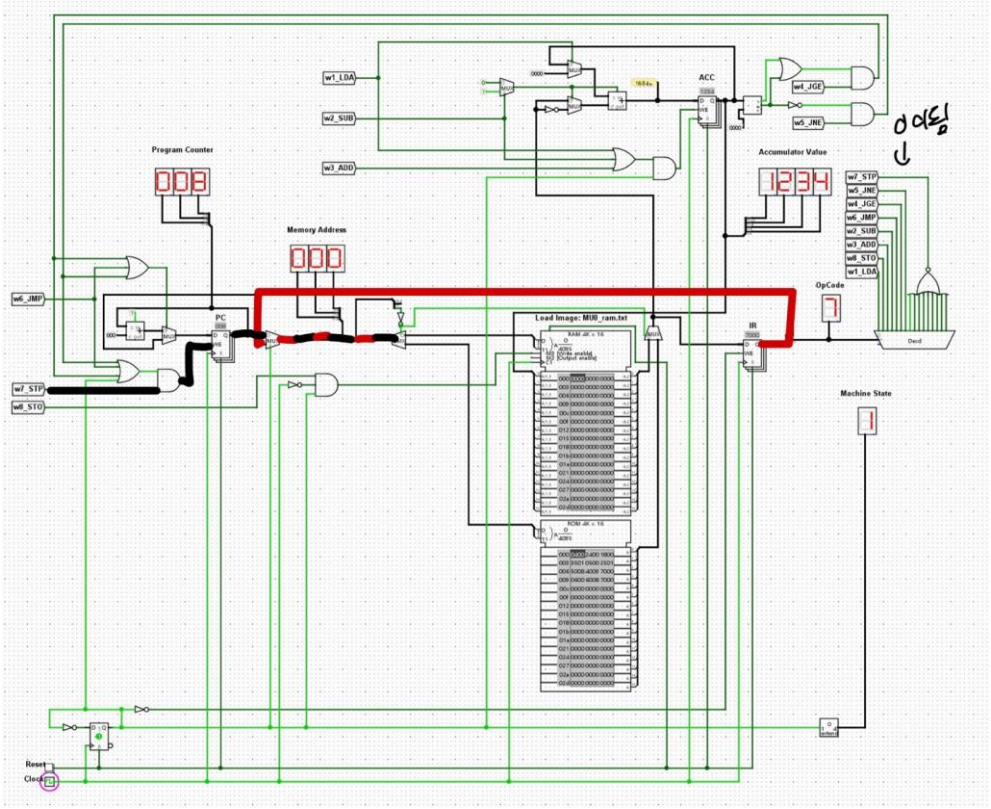
JGE 명령은 accumulator의 값이 0보다 크거나 같을 때 점프하기 때문에 accumulator의 comparator와 and게이트로 묶여 PC에 들어갈 값을 결정하는 MUX의 selector와 연결되게 된다. 위의 사진의 결과는 accumulator의 첫번째 값이 8 즉 1000이므로 음수이기 때문에 파란선이 0이고 mux는 PC+1을 PC로 보내게 된다.

<JNE> 6008



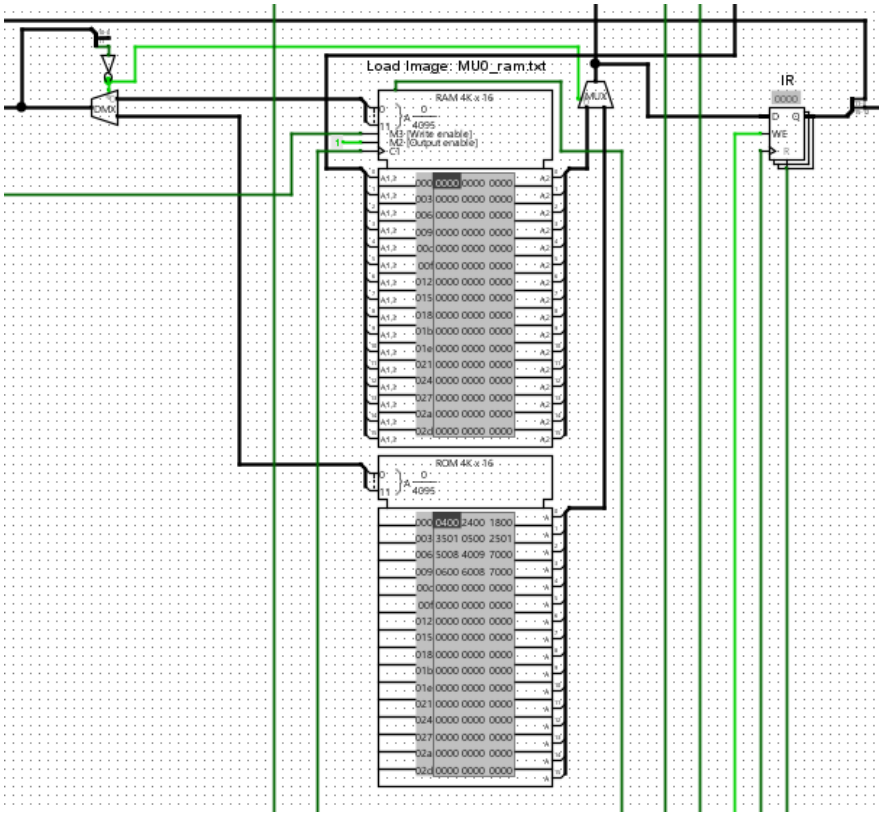
JNE 또한 점프조건이 존재하며 acuumulator의 값이 0이 아닐 때 점프하게 된다. 위의 사진에서 accumulator의 값은 0이 아니므로 파란선은 1이 되게 되고 점프할 주소인 검정선이 PC에 쓰여지게 된다.

<JNE> 7000



마지막으로 STP 명령어이다. 명령어가 해석되면 nor게이트와 연결되어 있어 다른 명령어 수행 시 항상 1을 유지하던 STP단자가 0으로 꺼지게 된다. 그 결과는 검정선으로 PC의 write enable의 값을 결정하던 값이 0으로 변하여 PC에 값을 쓸 수 없게 되었다는 것을 의미하고 이는 수행할 다음 명령어의 값을 바꿀 수 없음을 의미하기 때문에 프로그램이 정지하게 되는 것이다. 저장장치에 연결되어 있는 부분은 현재 PC의 값과 7000 중 000을 번갈아 가며 읽기만 반복하게 된다.

명령어 설명을 마치고 추가적으로 메모리의 설계를 제안서에 제시된 방법으로 설계해 보았다.



설계방법은 0부터 7FF까지는 비휘발성 메모리 ROM을 사용하고 800부터 FFF까지는 휘발성 메모리 RAM을 사용해야하므로 ROM과 RAM을 준비하고 저장장치로 들어가는 주소값을 빼내어 splitter를 이용하여 첫번째 비트만 추출하였다. 추출한 비트를 Demultiplexer에 selector로서 연결하고 이것이 0일때는 ROM으로 연결되고 1일때는 RAM으로 연결되어 주소에 맞게 저장장치를 사용하였다. 또한 출력시에도 사용한 저장장치의 출력값을 사용해야만 하므로 mux를 이용하여 출력값을 동일한 selector로 선택하게끔 구현하였다. 휘발성 메모리 RAM은 reset단자와 clear pin을 연결해주어 reset시 0으로 초기화되게끔 하였다.

위에서 구성한 명령어 시나리오를 ROM에 LOAD한 후 reset단자를 눌러 RAM을 초기화 시키고 명령어를 끝까지 수행하게 되면 아래와 같다.

```
800  0000 0000 0000 0000  0000 0000 0000 0000
808  0000 0000 0000 0000  0000 0000 0000 0000
810  0000 0000 0000 0000  0000 0000 0000 0000
818  0000 0000 0000 0000  0000 0000 0000 0000
820  0000 0000 0000 0000  0000 0000 0000 0000
```

<reset한 후 초기 상태>

```
800  0040 0000 0000 0000  0000 0000 0000 0000
808  0000 0000 0000 0000  0000 0000 0000 0000
810  0000 0000 0000 0000  0000 0000 0000 0000
818  0000 0000 0000 0000  0000 0000 0000 0000
```

<프로그램 종료 후>

STO명령에서 주소 800에 40이 저장되었음을 알 수 있다.

다음은 주어진 MU0_ram.txt파일의 명령어를 순차적으로 실행하였을 때 명령어가 어셈블리 언어로 어떻게 변화하는지 또한 accumulator값은 어떻게 변화하는지를 설명해보도록 한다.

Memory Address	instruction	opcode	Access Address	Accumulator	PC	Execution Order(decimal)
0000	0400	LDA	400	20	1	1
0001	2400	ADD	400	40	2	2
0002	1800	STO	800	40	3	3
0003	0401	LDA	401	5555	4	4
0004	2402	ADD	402	55FF	5	5
0005	3403	SUB	403	55AA	6	6
0006	1801	STO	801	55AA	7	7
0007	4010	JMP	10	55AA	10	8
0008	7000	STP	0	x	x	x
0009	0600	LDA	600	1234	A	15
000A	3600	SUB	600	0	B	16
000B	0616	LDA	616	0	C	17
000C	2600	ADD	600	1234	D	18
000D	1803	STO	803	1234	E	19
000E	6000	JNE	0	1234	0	20
000F	7000	STP	0	x	x	x
0010	0500	LDA	500	7FFF	11	9
0011	2501	ADD	501	8000	12	10
0012	5008	JGE	8	8000	13	11
0013	3501	SUB	501	7FFF	14	12
0014	1802	STO	802	7FFF	15	13
0015	5009	JGE	9	7FFF	9	14
0016	7000	STP	0	x	x	x

아래는 참조한 memory address이다.

0400	0020
0401	5555
0402	00AA
0403	0055
0500	7FFF
0501	0001

0600	1234
0601	0000

7f8 0000 0000 0000 0000 0000 0000 0000 0000
800 0040 55aa 7fff 1234 0000 0000 0000 0000
808 0000 0000 0000 0000 0000 0000 0000 0000 <실행 후 저장된 값>

X는 실행되지 않았음을 의미하고 PC값으로 다음 실행될 값을 알 수 있으며 추가적으로 execution order로 진행 순서를 표기하였다. 명령이 진행되면서 accumulator의 값은 명령이 원하는 대로 잘 바뀌고 또 메모리에 잘 저장되었음을 알 수 있다.

<Conclusion>

첫번째 프로젝트를 진행하면서 logisim에 익숙하지 않은터라 회로를 그리는거 자체에도 시간이 많이 걸렸고 회로 하나하나에 대한 해석이 쉽지 않았다. 특히 clock에 가장 가깝게 연결되어있는 플립플롭에 대한 필요성이 의문이었고 이는 clock을 많이 눌러보며 어떠한 타이밍에 회로가 원하는 데이터가 뭔지 입력에 대기하고 있는 데이터는 쓰여야하는지 버려져야하는지를 생각했고 클록을 많이 동작시켜보면서 그 필요성을 이해할 수 있게 되었다. 프로젝트를 통해 더 logisim을 이용한 회로 그리기가 익숙해졌고 회로를 해석하는 다양한 접근이 가능해진 것 같다.