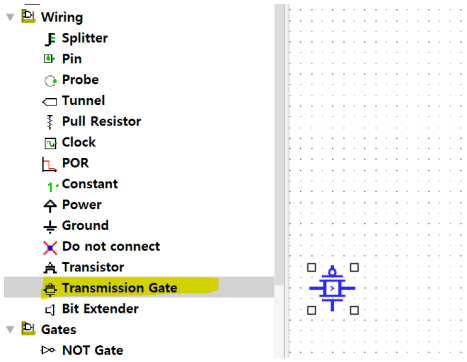
컴퓨터 구조

이성원 교수님

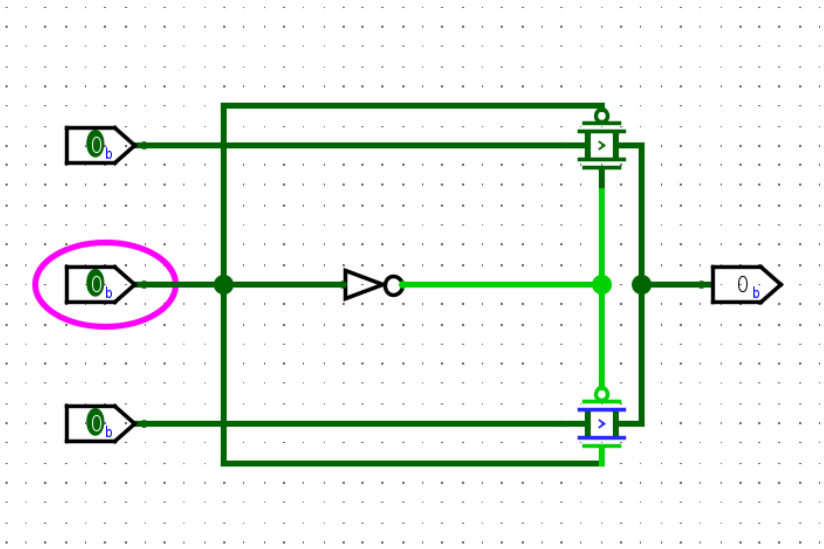
컴퓨터정보공학부 2021202003 강준우

1. What is the Transmission gate?



Transmission gate는 트렌지스터로 만들어진 gate로 스위치 같은 역할을 합니다. Pmos Nmos 트랜지스터로 만들어져 있으며, 각각의 트랜지스터의 반대되는 성질을 이용하여 흐름을 끊고 연결시킬 수 있습니다.

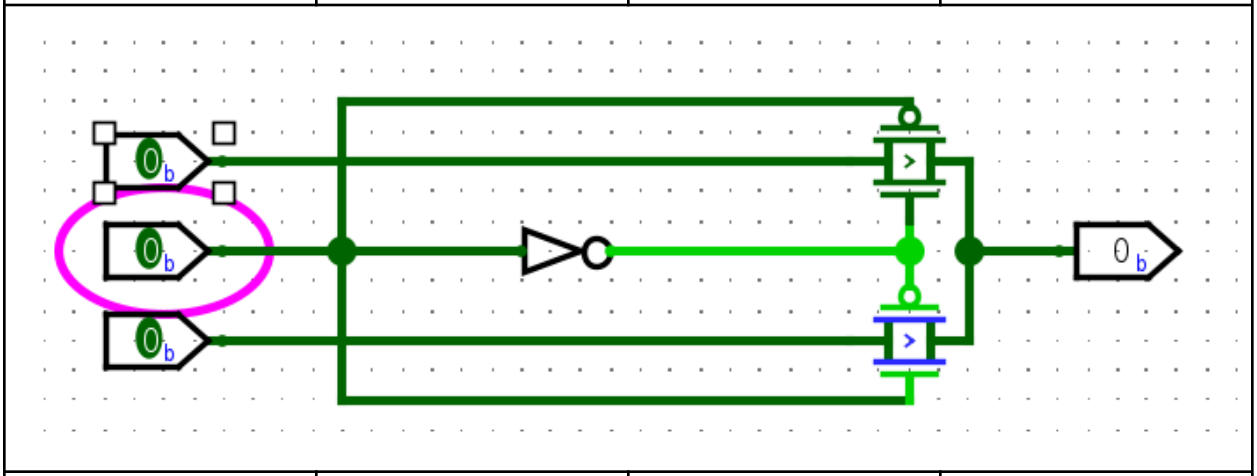
2. Implement a MUX using Transmission gates



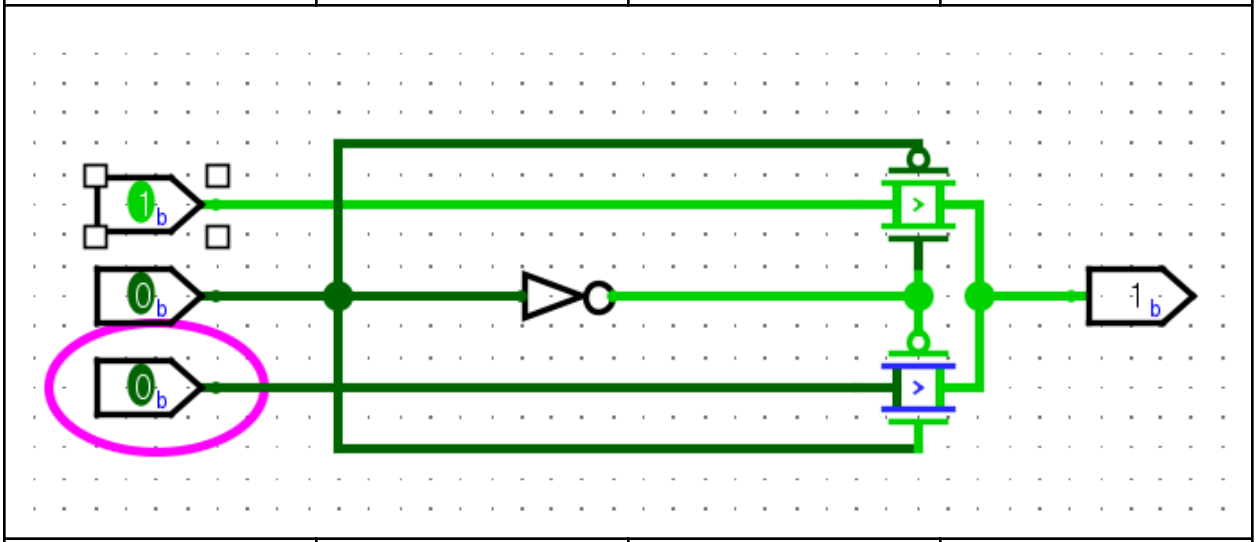
Mux truth table : 위에서부터 인풋을 각각 1 S 2라고 정의한다. 오른쪽 output에 해당하는 표

S	1	2	output

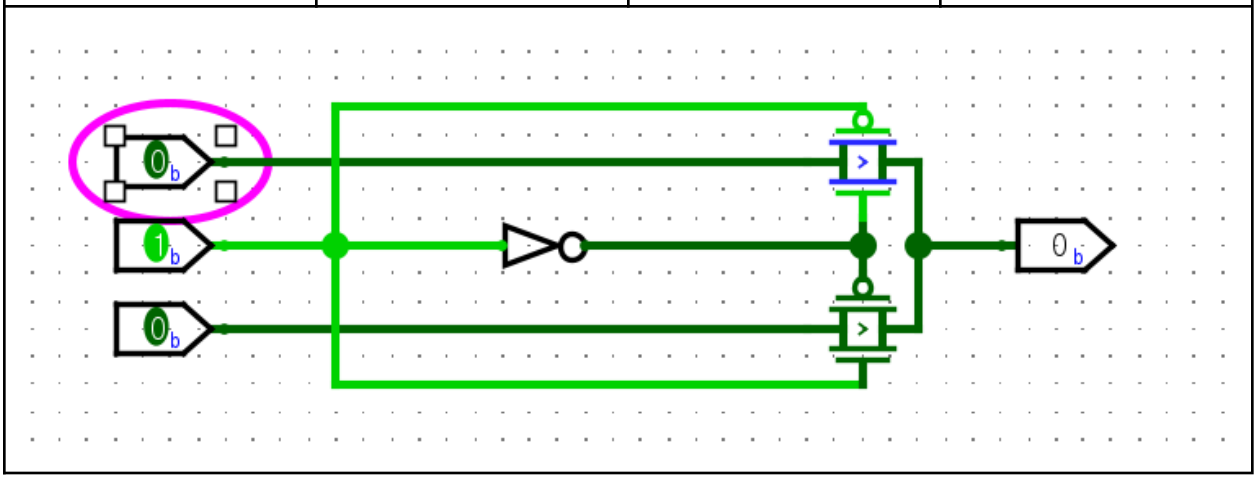
0	0	x	0



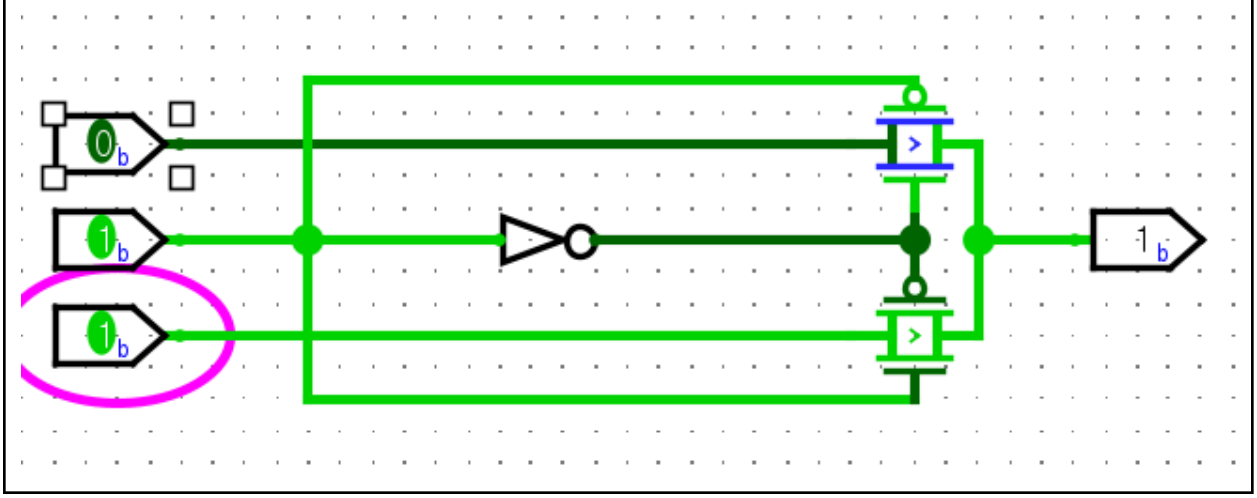
0	1	x	1



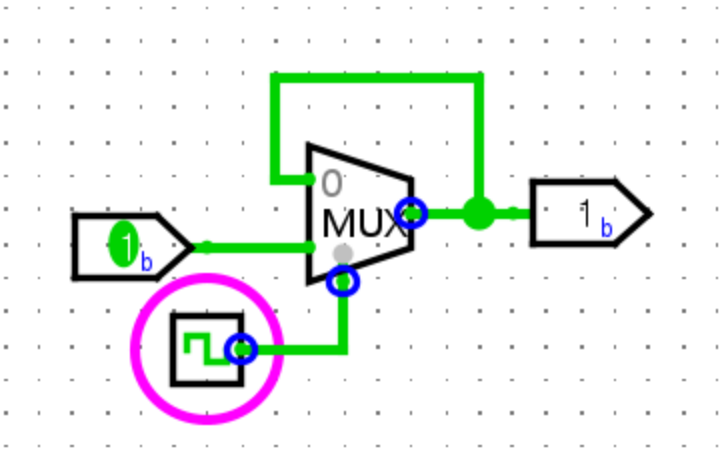
1	x	0	0



1	x	1	1

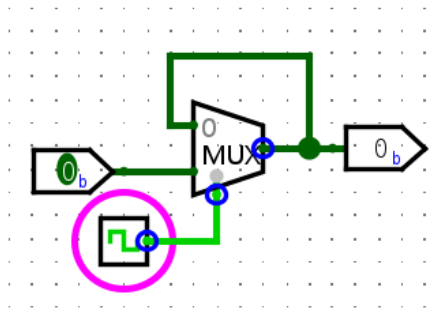
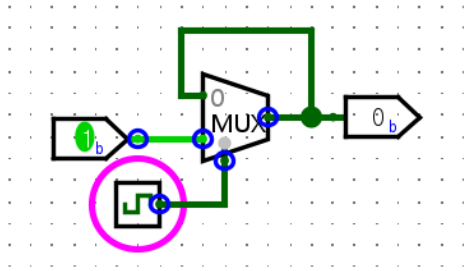
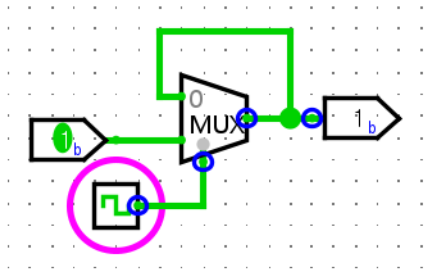


3. Implement a Latch using MUXs.

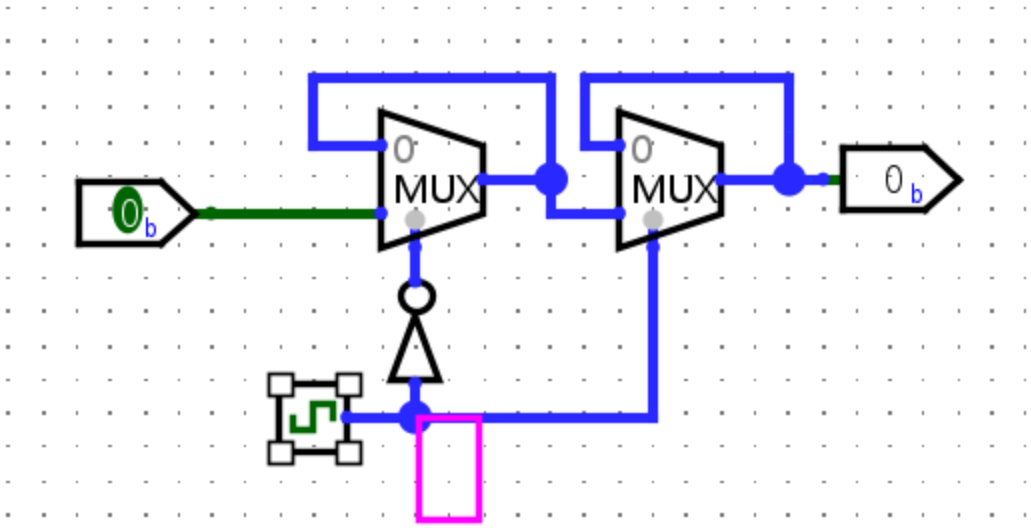


mux를 이용하여 다음과 같이 clock을 이용하여 latch를 구현할 수 있다. 해당 회로는 인풋 데이터를 메인 스위치와 같이 응용한 모습입니다.
Truth table: input clock output으로 정의한다. 현재 수행을 n으로 정의합니다.

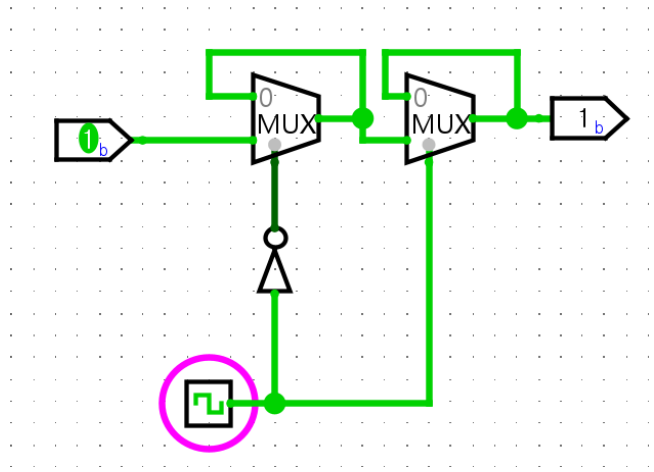
clock	input	output
0	x	n-1 data

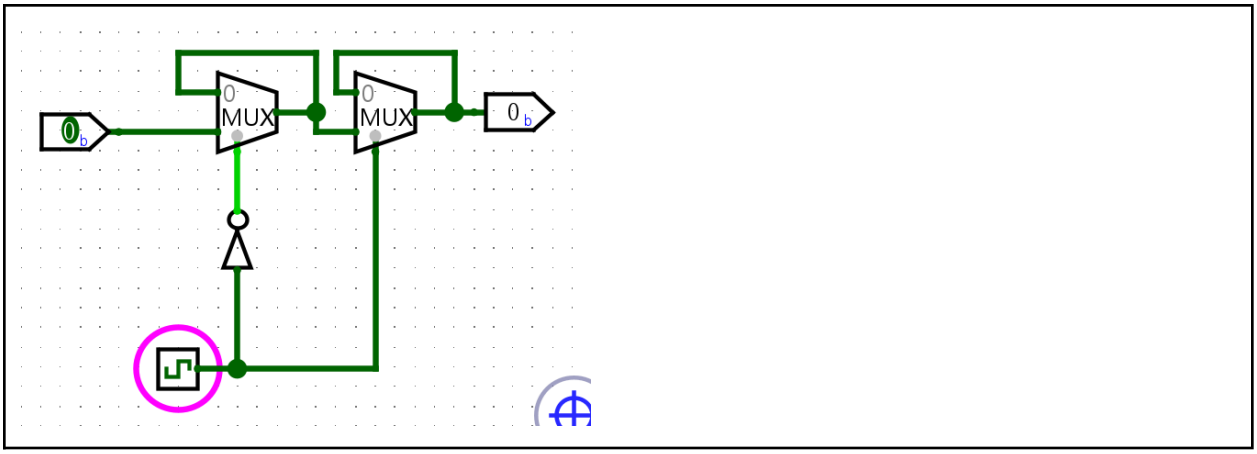
<table>
<tr><td colspan="3"> 
</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td colspan="3"> 
</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="3"> 
</td></tr>
</table>

4. Implement D-FF using Latches.



앞서 구현한 Dlatch를 1개더 연결하여 DFF를 구현했습니다,
Truth table : 입력과 출력 clock을 기준으로 표를 구현

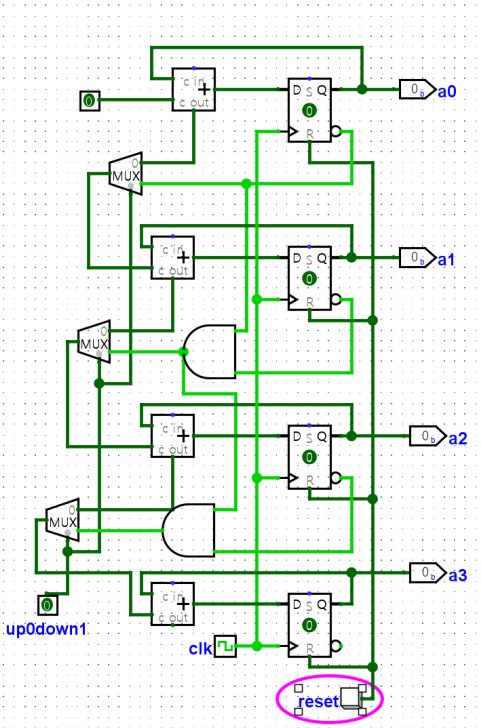
clock	input	output
up	1	1
		
down	0	0

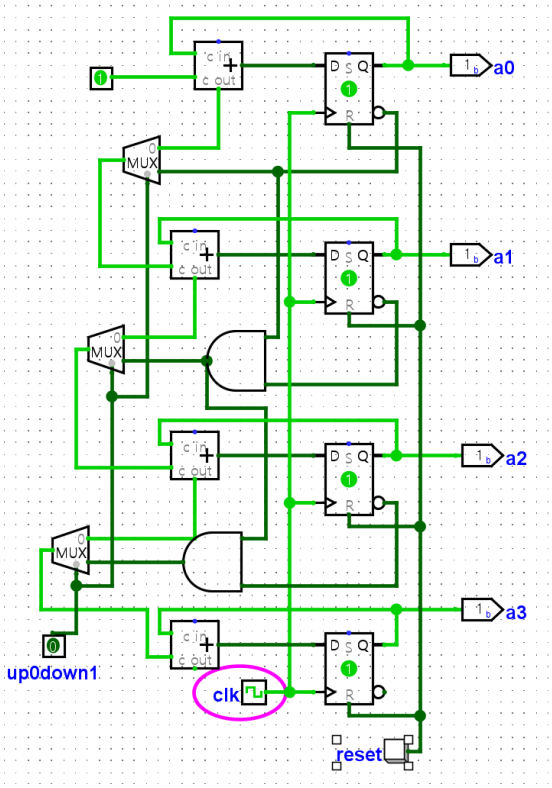


클럭이 상승일 때 인풋 데이터가 아웃풋으로 반영되는 모습을 확인할 수 있습니다.

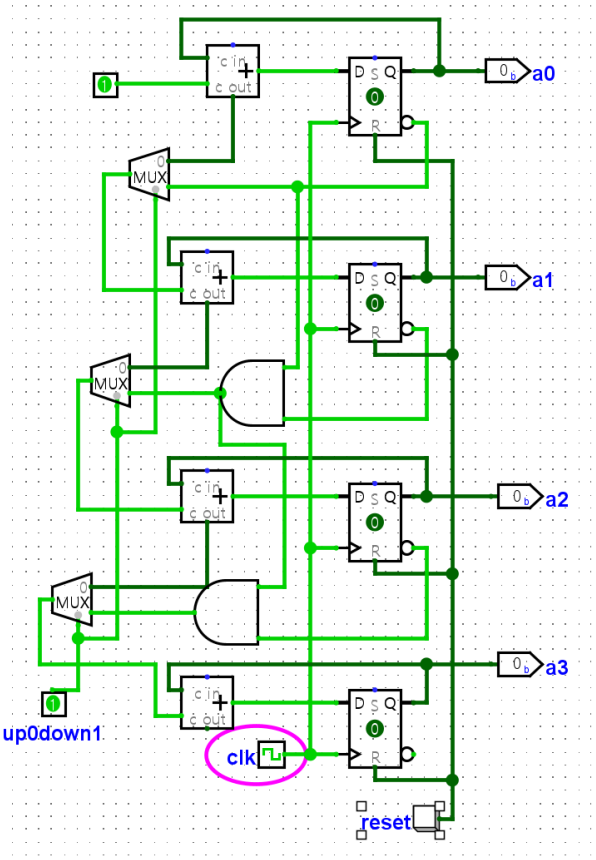
5. Implement Synchronous Up/Down Counter using D-FFs and adders.

해당 회로도는 DFF와 에더를 통해서 0000부터 1111까지 변화하는 카운터를 구현한 모습입니다.
회로도에는 왼쪽 위에 enable입력과 오른쪽 아래에 리셋을 통해서 전제 조건을 조절합니다.
추가로 UP/Down을 통해서 오름차순과 내림차순 카운터를 조절할 수 있습니다.





Up(0) enable(1)



Down(1) , enable(1)

truthtable : UP일때 표현 , down일때는 반대로 작용한다.

a3	a2	a1	a0	clk
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8

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

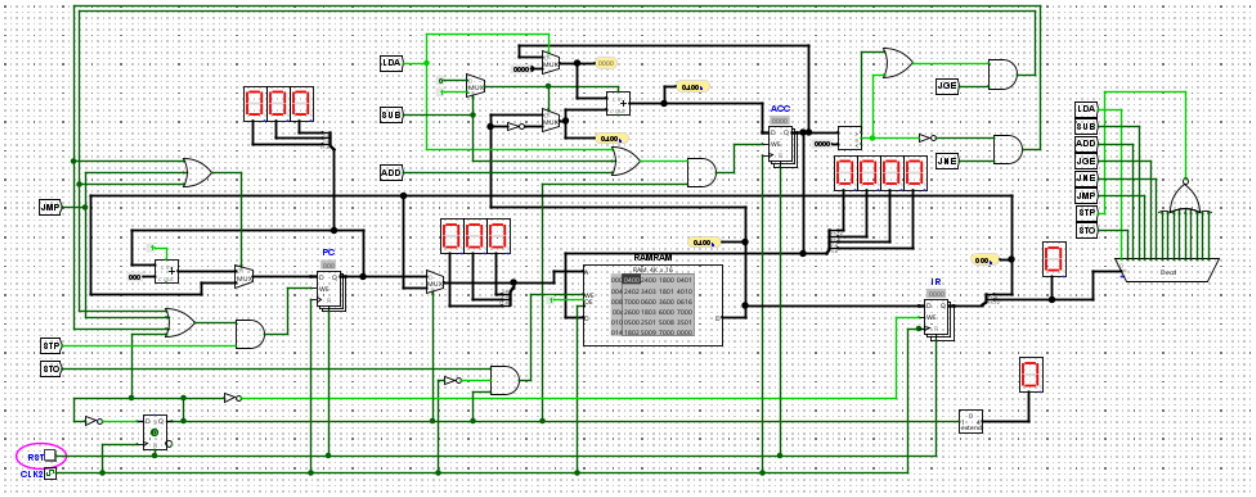
6. Compare Synchronous counter and Asynchronous counter such as a ripple counter.

동기식 카운터는 포든 FF가 동일한 클럭을 공유함에 반해 비동기식 카운터에서는 첫번째를 제외한 다른 FF는 첫번 째 FF의 값을 받아오는 형태로 구현되어 있습니다. 이런 차이가 결과적으로는 출력에 미세한 지연을 유발하고 결과적으로 동기식 카운터가 비동기식 카운터보다 빠르게 됩니다.
회로 설계적으로 보면 비교적 비동기식 카운터가 단순한 형태를 띄지만 정밀한 결과를 요구할때는 오차를 발생할 가능성이 높습니다.

7. What makes that Asynchronous counter is rarely used?

앞서 언급한 단점으로 인한 오차 발생 가능성으로 인해서 현재는 복잡하더라도 동기식 카운터를 더 많이 사용합니다.

2.2 Complete the implementation of MU0 (2 cycle CPU)



다음은 w들에 이름과 이후 디코더를 구현한 모습입니다.

명령어 터널 설명:

STP 프로그램 중지를 위해 설계되었습니다. 해당 신호를 통해서 pc가 클럭에 의해 바꾸지 않게 동작합니다.

LDA: 메모리에 데이터를 ACC로 로드 시키기 위해서 설계되었습니다.

SUB: 메모리의 값을 ACC에서 감하는 동작을 위해 설계되었습니다.

ADD: 메모리의 값을 ACC에 더하는 동작을 위해 설계되었습니다.

JGE: ACC가 0000을 기점으로 이상임을 판단 후 점프하기 위해 설계되었습니다.

JNE: ACC가 0이 아닐 경우 점프하기 위해 설계되었습니다.

JMP: 조건없이 지정된 주소로 점프하기 위해서 설계되었습니다.

STO: ACC값을 메모리에 저장하기 위해 설계되었습니다.

해당 터널들은 디코더를 통해 묶여있습니다. 이때 IR를 통해서 몇번째 터널을 사용할지 선택하게 됩니다.

회로도 설명**:**

CKL앞에 있는 DFF를 통해서 먼저 데이터를 임시 저장합니다. 사이클 구조에서 FETCH단계에서 해당 데이터를 전송하는 일을 방지하기 위해서 FETCH일때는 write enable를 닫고 EXECUTE단계에서만 열게 설계합니다. 이때 DFF를 이용해 CLK가 올릴때 조건하에 D값을 보내게 유도합니다.

PC

3개로 이루어진 디지털 다이얼 중 왼쪽 위는 pc값, 현재 수행하는 명령 순서를 항상 출력하지만 중앙에 있는 3개의 디지털 다이얼은은 선택적으로 PC값과 RAM에서의 주소값을 반환합니다. 이는 JUMP할 주소를 반환하는 것입니다. 해당 데이터는 MUX를 이용하여 선택하게 되는데, 이전의 DFF에 따라 클럭 타이밍에 맞춰 선택하는 모습입니다. 이를 통해서 이전에 언급한 PC값이 한클럭 늦게 설계됩니다. 가산기를 거쳐 만들어진 PC+1 에대한 명령어를 바로 적용하지 않고 MUX.DFF.CLK 순서대를 통해 흐르게 됩니다.

ACC

대표적으로 연산 결과를 저장하기 전 저장소로 사용됩니다. 이에 따라 ADD , SUB 터널과 연결되어있는 모습을 확인할 수 있습니다. 각각의 MUX에 상태에 따라서 수행합니다. 또한 LDA STO를 통해 불러오고 저장하는 기능 또한 수행합니다. 이에 따라 STO 입력과 출력에 각각 RAM의 D OUT / D IN이 연결되어있는 모습을 확인할 수 있습니다.

IR

RAM에서 가져온 다음 명령어를 임시저장합니다. 해당 데이터에서 상위 비트 4개를 기준으로 분기점을 나누어 앞에 4개비트는 디코더르 뒤에 12비트는 PC로 들어가게 됩니다.

MU0 프로그램은 메모리에 저장된 명령어를 순서대로 실행하면서 산술 연산을 수행하고, 조건에 따라 분기하거나 값을 저장하는 과정을 거칩니다. 이 프로그램의 전체적인 흐름을 설명하면 다음과 같습니다.

동작 검증을 위한 시나리오 제시

먼저 과제 자료로 받은 MU0의 명령어를 텍스트 파일로 가져와보았습니다.

```
v2.0 raw
0400 2400 1800 0401 2402 3403 1801 4010 7000 0600 3600 0616 2600 1803 6000 7000
0500 2501 5008 3501 1802 5009 7000

0020 5555 00aa 0055

7fff 0001

1234
```

이를 통해서 우리는 해당 명령어가 있음을 알 수 있습니다. 이제 각각의 명령어가 어떤 수행을 하는지 해석해보면 다음과 같습니다.

주소	내용	해석
0X0000	0400	LDA 400

0X0001	2400	ADD 400
0X0002	1800	STO 800
0X0003	0401	LDA 401
0X0004	2402	ADD 402
0X0005	3403	SUB 403
0X0006	1801	STO 801
0X0007	4010	JMP 0010
0X0008	7000	STP
0X0009	0600	LDA 600
0X000A	3600	SUB 600
0X000B	0616	LDA 616
0X000C	2600	ADD 600
0X000D	1803	STO 803
0X000E	6000	JNE 0000
0X000F	7000	STP
0X0010	0500	LDA 500
0X0011	2501	ADD 501
0X0012	5008	JGE 0008
0X0013	3501	SUB 501
0X0014	1802	STO 802
0X0015	5009	JGE 0009
0X0016	7000	STP

그리고 이후에 데이터가 주소와 값으로 존재합니다. 이외에 주소 데이터는 0000입니다.

0x0400	0020
0x0401	5555
0x0402	00AA
0x0403	0055
0x0500	7FFF
0x0501	0001
0x0600	1234

순서대로 구현해보면 다음과 같습니다.

0000 LDA 400 ACC ← 0x0400 = 20

0001 ADD 400 ACC ← ACC + 0x0400 = 20 + 20 = 40

0002 STO 800 0x0800 ← ACC = 40

0003 LDA 401 ACC ← 0x0401 = 5555

0004 ADD 402 ACC ← 5555 + 55FF = 55FF

0005 SUB 403 ACC ← 55FF - 0055 = 55AA

0006 STO 801 0x0801 ← 55AA

0007 JMP 10 PC ← 0010

0010 LDA 500 ACC ← 0x0500 = 7FFF

0011 ADD 501 ACC ← 7FFF + 8000 = 8000

0012 JGE 8 ACC = 8000 < 0 , 점프 없이 0013

0013 SUB 501 ACC ← 8000 - 8000 = 0

0014 STO 802 0x0802 ← 0

0015 JGE 9 ACC = 0 ≥ 0 , 0009로 점프

0009 LDA 600 ACC ← = 1234

000A SUB 600 ACC ← 1234 - 1234 = 0

000B LDA 616 ACC ← 0x0616 = 0

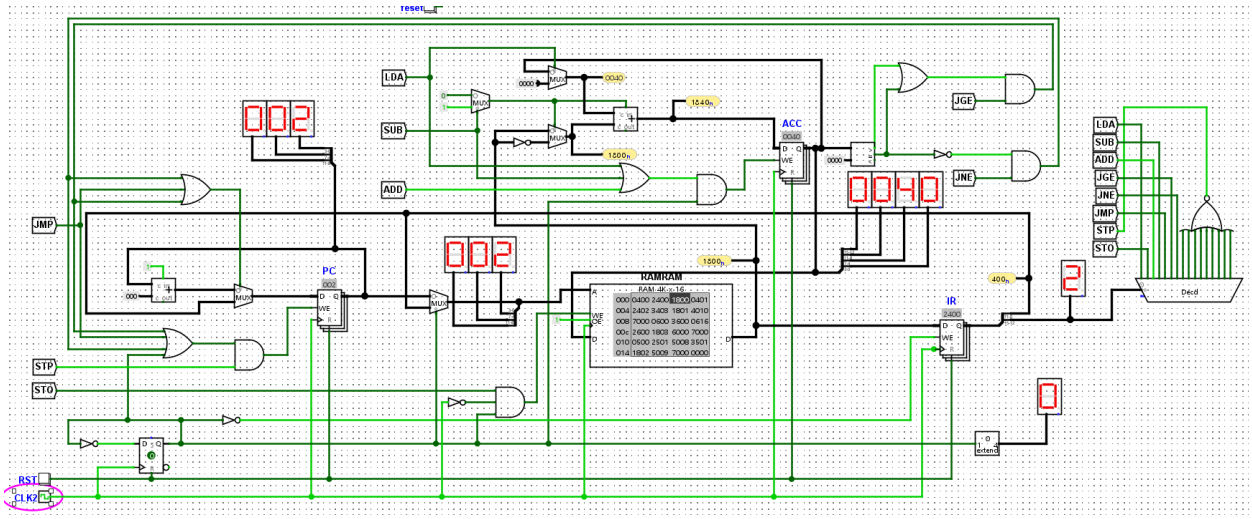
000C ADD 600 ACC ← 0 + 1234 = 1234

000D STO 803 0x0803 ← 1234

000E JNE 0 ACC(1234) != 0, 0000으로 점프

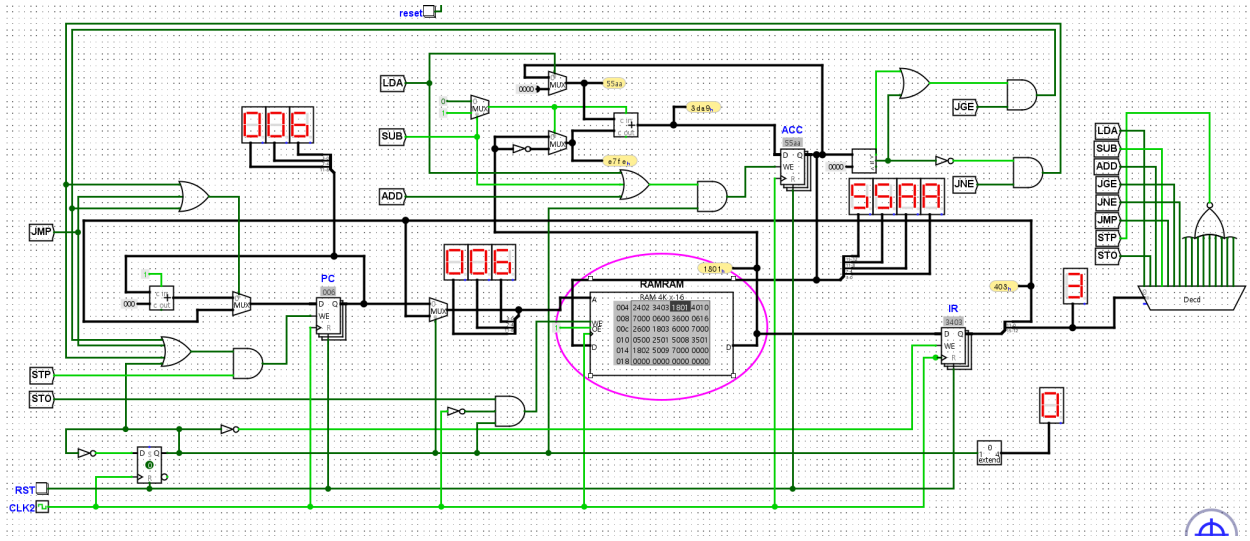
결과 분석

0000주소에서 시작한 프로그램은 400번째 명령어를 통해서 20값을 LDA해옵니다. 이후 ADD명령어를 통해서 20+20을 통해 40을 ACC에서 출력합니다. 다음은 40이 ACC에서 나온 모습입니다. 이후 결과는 STO 800을 통해 저장됩니다.

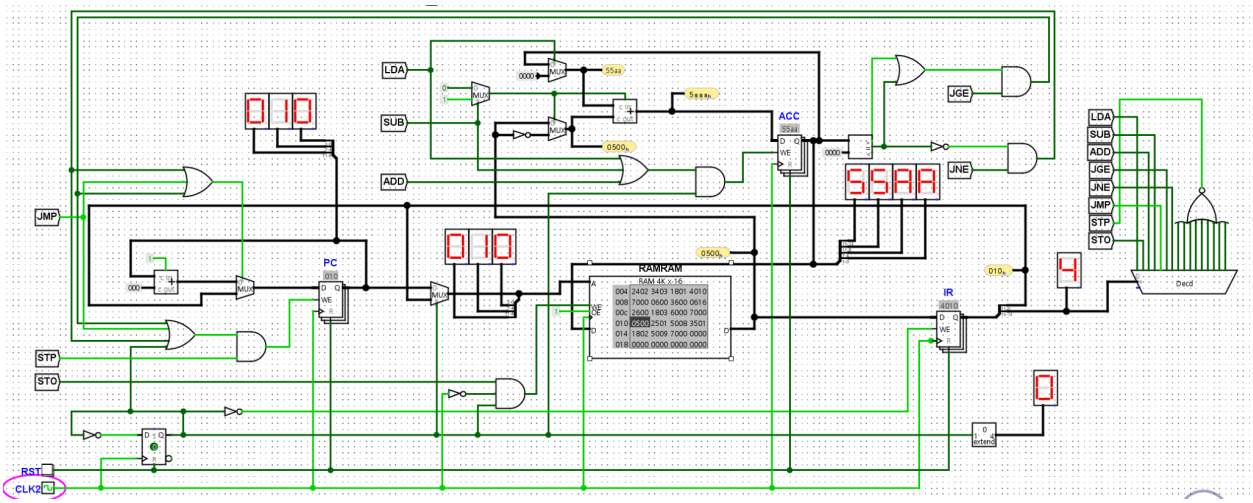


4번째 LDA 명령어를 수행하여 5555값을 ACC로 저장해둡니다. 이후 ADD와 SUB를 통해서 5555 > 55FF > 55AA로 값이 변화합니다. 해당 결과는 STO 801을 통해서 저장됩니다.

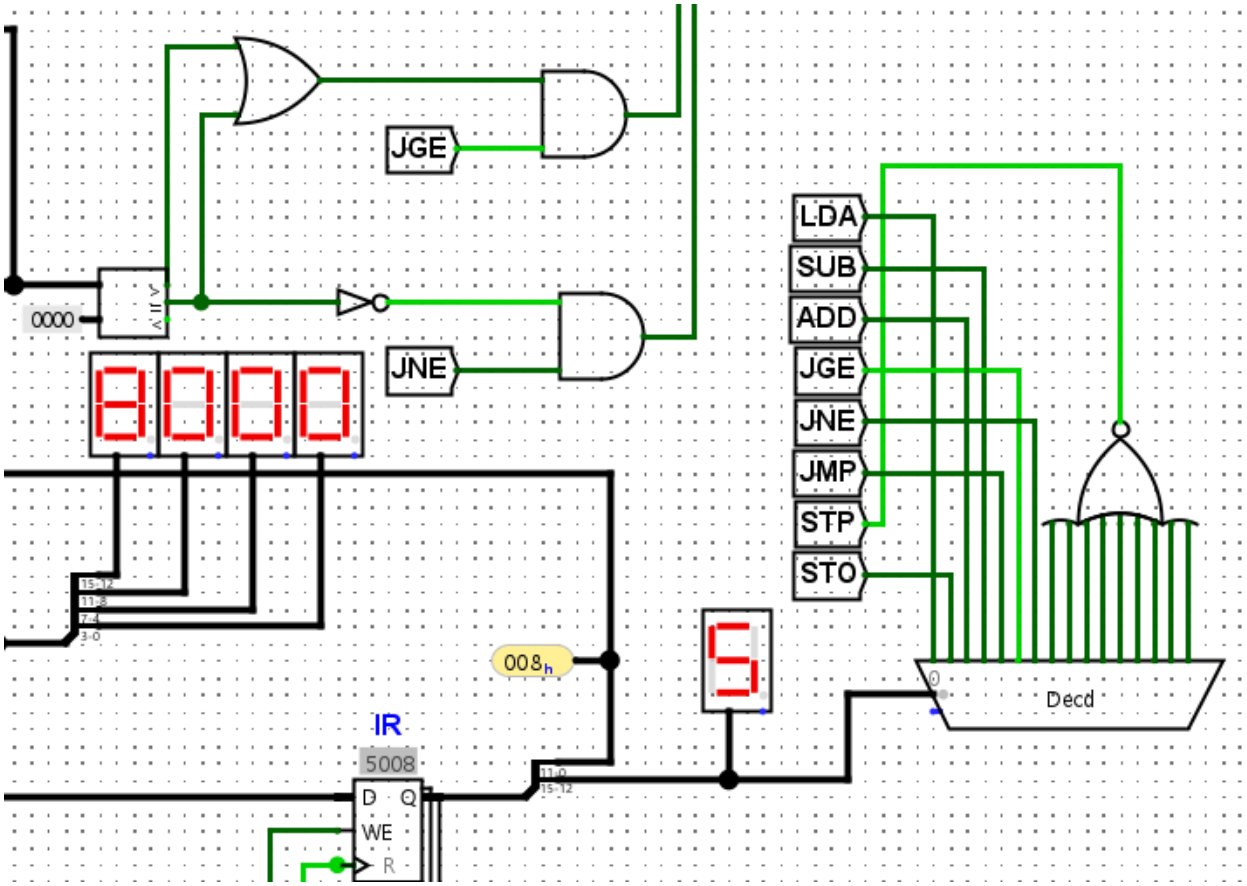
다음은 55AA의 모습입니다.



점프 명령어를 통해서 이후 10번 주소로 이동하는 모습입니다.

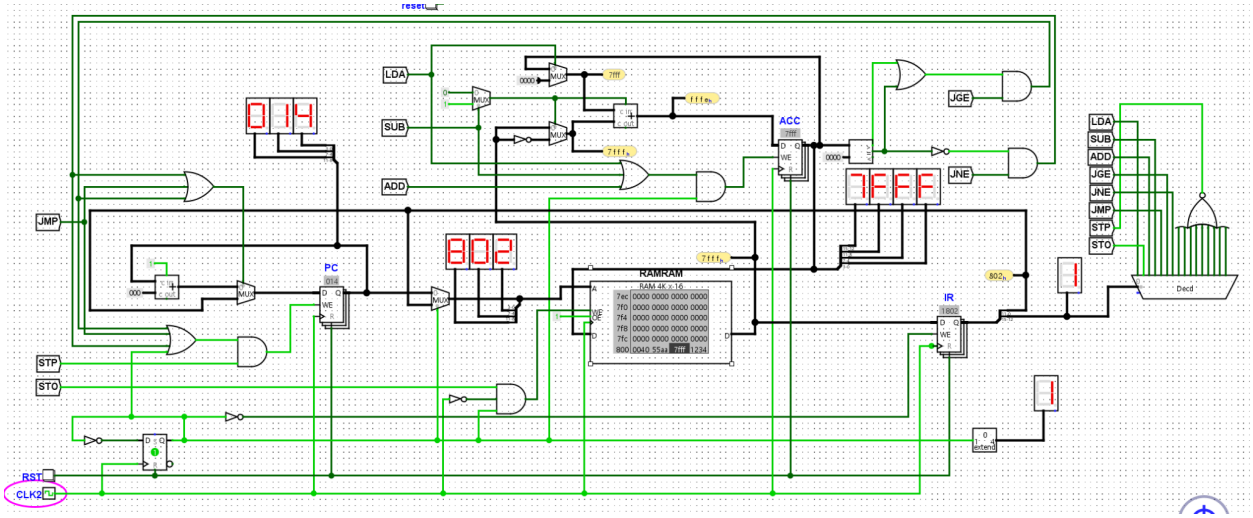


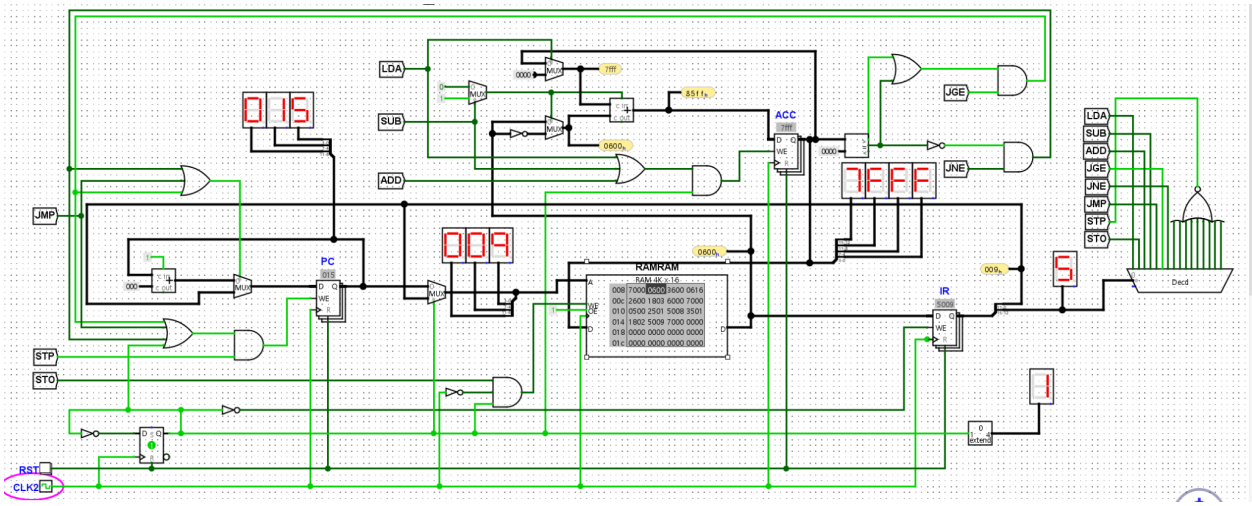
해당 주소로 이동 후 7FFF를 ADD로 더한결과 8000이 나옵니다. 하지만 이때 8000은 2진수로 1000 0000 0000 0000으로 부호비트를 통해서 음수처리가 되고 결과에서 보다싶이 음수로 판별됩니다.



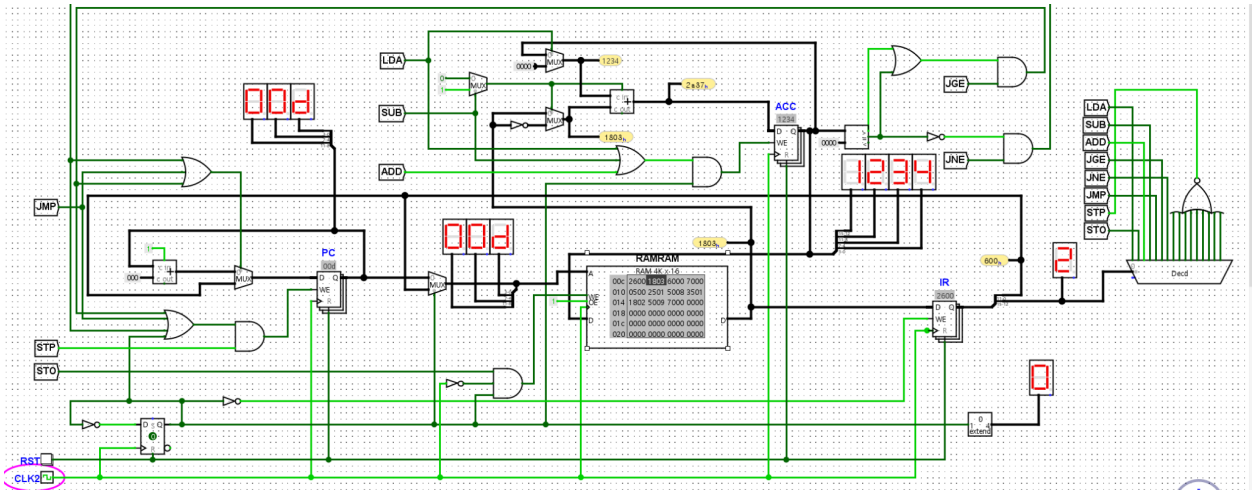
해당사진에서 JGE가 발생했지만 조건에 따라 점프하지 않는 모습입니다.

이후 501 명령어로 8000을 빼고 802주소로 저장된 모습입니다.



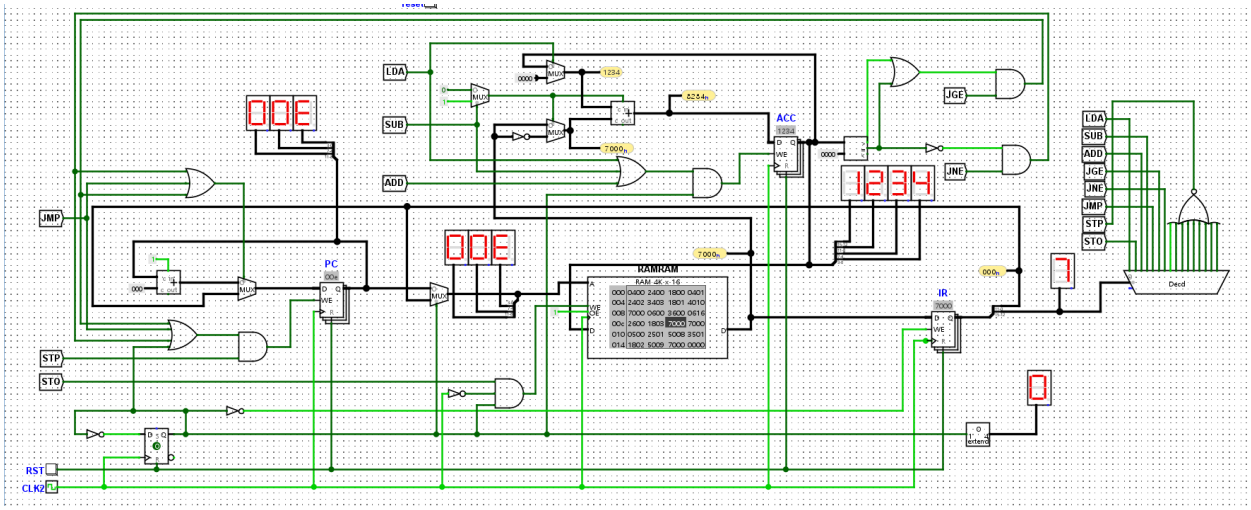


다시한번 JGE 명령어가 실행되는데, 이번에는 ACC가 0이상 이므로 조건에 따라 주소 0009로 점프합니다.



0009에서 LDA 600 명령어를 통해 ACC에 1234를 저장합니다. 이후 SUB를 통해서 해당 값을 0으로 바꿉니다. 이후 LDA 616은 0을 로드하고, ADD로 1234를 더해 다시 ACC는 다시 한번 1234가 됩니다. 결과를 STO 803을 통해 저장합니다.

JNE명령어를 통해 조건을 달성해, 처음 주소 0000으로 점프하여 위의 루프의 모습을 수행합니다.



해당 코드는 STP를 추가 구현한 모습입니다. 기존의 시나리오에서 마지막 점프하는 것이 아닌 0000이 아닌 7000으로 대체하여 정지를 구현하였고, pc값이 증가하지 않는 모습과 ACC에서도 출력이 변화되지 않는 모습을 통해 정상 작동하는 모습을 보았다.

v2.0 raw

0400 2400 1800 0401 2402 3403 1801 4010 7000 0600 3600 0616 2600 1803 7000 7000

0500 2501 5008 3501 1802 5009 7000 —-(이후 명령어 없음)

위와같이 변경시켜 구현하였다.

고찰

이번 과제를 통해서 명령어와 데이터를 읽으며 회로를 구현하기 전부터 결과를 예측해볼 수 있었습니다. 또한 해당 메모리를 사용하면서 16진수를 많이 접하여 익숙해질 수 있었습니다.

회로를 구현할 때 터널링 비트 스플릿 구간에서 앞에 4개를 가져오는 부분을 이해하지 못하고 회로를 구현하여 선에서 빨강 혹은 주황색을 경험했습니다. 스플릿이 무조건적으로 같은 비율로 나뉘는게 아닌, 조건에 따라서 앞에 4개만 가져오는 등의 분할이 생길 수 있음을 배웠습니다. 또한 이런 설계가 결과적으로 무한 루프를 연상시켜 이후 조건문이 이미 존재하니 다양한 코딩을 구현할 수 있을것이라고 느꼈습니다.

마지막으로 로지심 하나에 모든 회로를 구현했는데, 이러다보니 시뮬레이터 할 때 클럭이 모두 발생하는 모습에 파일 분할이 필요하다 생각했습니다.

참조 : 컴퓨터구조 , 컴퓨터구조실험 자료