로고이(가) 표시된 사진

자동 생성된 설명

컴퓨터 구조

Project 2 제

수업 명 : 컴퓨터구조

과제 이름 : project2

담당 교수님 : 이성원 교수님

학 번 : 2019202005

이 름 : 남종식

**문제의 해석 및 해결 방향**

**1. ADDU -> add unsigned**

**ADDU는 레지스터 두 개를 인자로 받아서, 두 레지스터의 값을 더한 결과를 목적 레지스터에 저장합니다. 이는 부호 없는 덧셈을 수행하기 때문에 양수와 음수를 더할 때 오버플로우가 발생하지 않습니다.**

**ADDU $d, $s, $t operation : $d = $s + $t**

**opcode : 0 / function : 100001 / type : R type**

**이는 $s 레지스터와 $t 레지스터의 값을 더한 결과를 $d 레지스터에 저장합니다.**

**IorD :**

**MemRead :**

**MemWrite :**

**DataWidth :**

**IRwrite :**

**RegDst :**

**RegDatSel :**

**RegWrite :**

**ExtMode :**

**ALUsrcA :**

**ALUsrcB :**

**ALUop :**

**ALUctrl :**

**Branch :**

**PCsource :**

**PCwrite :**

**StateSel :**

**2. OR -> or**

**OR는 레지스터 두 개를 인자로 받아서, 두 레지스터의 값을 bitwise OR 연산한 결과를 목적 레지스터에 저장합니다.**

**OR $d, $s, $t**

**operation : $d = $s | $t**

**opcode : 0 / function : 100101 / type : R type**

**이는 $s 레지스터와 $t 레지스터의 값을 bitwise OR 연산한 결과를 $d 레지스터에 저장합니다.**

**IorD :**

**MemRead :**

**MemWrite :**

**DataWidth :**

**IRwrite :**

**RegDst :**

**RegDatSel :**

**RegWrite :**

**ExtMode :**

**ALUsrcA :**

**ALUsrcB :**

**ALUop :**

**ALUctrl :**

**Branch :**

**PCsource :**

**PCwrite :**

**StateSel :**

**3. ADDIU -> add immediate unsigned**

**ADDIU는 레지스터와 상수를 인자로 받아서, 레지스터의 값을 상수만큼 증가시킨 다음 그 결과를 목적 레지스터에 저장합니다. 이는 부호 없는 덧셈을 수행하기 때문에 양수와 음수를 더할 때 오버플로우가 발생하지 않습니다.**

**ADDIU $t, $s, I**

**operation : $t = $s + SE(i) opcode : 001001 / type : I type**

**이는 $s 레지스터의 값을 i만큼 증가시킨 다음, 그 결과를 $t 레지스터에 저장합니다.**

**IorD :**

**MemRead :**

**MemWrite :**

**DataWidth :**

**IRwrite :**

**RegDst :**

**RegDatSel :**

**RegWrite :**

**ExtMode :**

**ALUsrcA :**

**ALUsrcB :**

**ALUop :**

**ALUctrl :**

**Branch :**

**PCsource :**

**PCwrite :**

**StateSel :**

**4. XORI -> exclusive or immediate**

**XORI는 레지스터와 상수를 인자로 받아서, 레지스터의 값과 상수의 비트 별 XOR 연산한 결과를 목적 레지스터에 저장합니다.**

**XORI $t, $s, I**

**operation : $t = $s ^ ZE(i) opcode : 001110 / type : I type**

**이는 $s 레지스터의 값을 i와 비트 별 XOR 연산한 결과를 $t 레지스터에 저장합니다.**

**IorD :**

**MemRead :**

**MemWrite :**

**DataWidth :**

**IRwrite :**

**RegDst :**

**RegDatSel :**

**RegWrite :**

**ExtMode :**

**ALUsrcA :**

**ALUsrcB :**

**ALUop :**

**ALUctrl :**

**Branch :**

**PCsource :**

**PCwrite :**

**StateSel :**

**5. SLL -> shift left logical**

**SLL은 레지스터와 상수를 인자로 받아서, 레지스터의 값을 상수만큼 왼쪽으로 시프트한 결과를 목적 레지스터에 저장합니다.**

**SLL $d, $t, a**

**operation : $d = $t << a**

**opcode : 0 / function : 000000 / type : R type**

**이는 $t 레지스터의 값을 왼쪽으로 a비트 시프트한 결과를 $d 레지스터에 저장합니다. 이는 왼쪽으로 시프트하기 때문에, 시프트된 비트는 0으로 채워집니다.**

**IorD :**

**MemRead :**

**MemWrite :**

**DataWidth :**

**IRwrite :**

**RegDst :**

**RegDatSel :**

**RegWrite :**

**ExtMode :**

**ALUsrcA :**

**ALUsrcB :**

**ALUop :**

**ALUctrl :**

**Branch :**

**PCsource :**

**PCwrite :**

**StateSel :**

**6. SRAV -> shift right arithmetic variable**

**SRAV는 레지스터와 레지스터를 인자로 받아서, 첫 번째 레지스터의 값을 두 번째 레지스터로 지정된 비트 수만큼 오른쪽으로 시프트한 결과를 목적 레지스터에 저장합니다. 시프트된 비트는 첫 번째 레지스터의 최상위 비트와 같은 값으로 채워집니다.**

**SRAV $d, $t, $s**

**operation : $d = $t >>> $s**

**opcode : 0 / function : 000111 / type : R type**

**이는 $t 레지스터의 값을 $s 레지스터로 지정된 비트 수만큼 오른쪽으로 시프트한 결과를 $d 레지스터에 저장합니다.**

**이는 오른쪽으로 시프트하기 때문에, 시프트된 비트는 첫 번째 레지스터의 최상위 비트와 같은 값으로 채워집니다.**

**IorD :**

**MemRead :**

**MemWrite :**

**DataWidth :**

**IRwrite :**

**RegDst :**

**RegDatSel :**

**RegWrite :**

**ExtMode :**

**ALUsrcA :**

**ALUsrcB :**

**ALUop :**

**ALUctrl :**

**Branch :**

**PCsource :**

**PCwrite :**

**StateSel :**

**7. SH -> store half words**

**SH는 레지스터와 상수를 인자로 받아서, 레지스터의 값을 상수만큼 이동한 메모리 위치에 있는 2바이트 크기의 (워드의 반) 데이터에 저장합니다.**

**SH $t, i($s)**

**operation : MEM [$s + i]:2 = LH ($t) opcode : 101001 / type : I type**

**이는 $t 레지스터의 값을 $s 레지스터의 값에 i를 더한 메모리 위치에 있는 2바이트 크기의 데이터에 저장합니다**

**IorD :**

**MemRead :**

**MemWrite :**

**DataWidth :**

**IRwrite :**

**RegDst :**

**RegDatSel :**

**RegWrite :**

**ExtMode :**

**ALUsrcA :**

**ALUsrcB :**

**ALUop :**

**ALUctrl :**

**Branch :**

**PCsource :**

**PCwrite :**

**StateSel :**

**8. LH -> load half words**

**LH는 레지스터와 상수를 인자로 받아서, 레지스터의 값을 상수만큼 이동한 메모리 위치에 있는 2바이트 크기의 데이터를 로드 하여 레지스터에 저장합니다.**

**LH $t, i($s)**

**operation : $t = SE (MEM [$s + i]:2) opcode : 100001 / type : I type**

**이는 $s 레지스터의 값에 i를 더한 메모리 위치에 있는 2바이트 크기의 데이터를**

**$t 레지스터에 로드 합니다.**

**IorD :**

**MemRead :**

**MemWrite :**

**DataWidth :**

**IRwrite :**

**RegDst :**

**RegDatSel :**

**RegWrite :**

**ExtMode :**

**ALUsrcA :**

**ALUsrcB :**

**ALUop :**

**ALUctrl :**

**Branch :**

**PCsource :**

**PCwrite :**

**StateSel :**

**9. BLTZ -> branch on than lsss than zero**

**BLTZ는 레지스터와 상대적인 분기 주소를 인자로 받아서, 첫 번째 레지스터의 값이 0보다 작을 때 상대적인 분기 주소로 분기합니다.**

**BLTZ $s, label**

**operation : if ($s < 0) pc += i << 2**

**opcode : 000001 / RegImm : 00000 / type : I type**

**이는 $s 레지스터의 값이 0보다 작으면 label로 분기합니다.**

**IorD :**

**MemRead :**

**MemWrite :**

**DataWidth :**

**IRwrite :**

**RegDst :**

**RegDatSel :**

**RegWrite :**

**ExtMode :**

**ALUsrcA :**

**ALUsrcB :**

**ALUop :**

**ALUctrl :**

**Branch :**

**PCsource :**

**PCwrite :**

**StateSel :**

**10. JAL -> jump and link**

**JAL은 레이블 또는 상대적인 분기 주소를 인자로 받아서, 다음 명령어의 주소를 레지스터에 저장한 후에 분기합니다.**

**jal label**

**operation : $31 = pc; pc = pc4 | i26 << 2 opcode : 000011 / type : J type**

**IorD :**

**MemRead :**

**MemWrite :**

**DataWidth :**

**IRwrite :**

**RegDst :**

**RegDatSel :**

**RegWrite :**

**ExtMode :**

**ALUsrcA :**

**ALUsrcB :**

**ALUop :**

**ALUctrl :**

**Branch :**

**PCsource :**

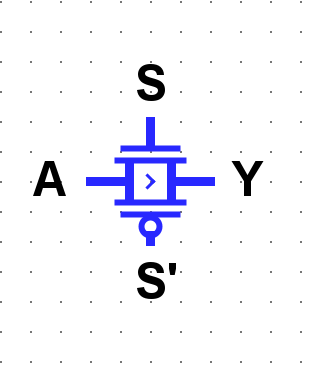
**PCwrite :**

**StateSel :**

Transmission Gate란 쉽게 말해서 회로에서 사용되는 양방향 스위치입니다. Transmission Gate는 NMOS와 PMOS 트랜지스터를 병렬로 연결한 구조로 되어 있으며, 게이트는 서로 연결되어 있습니다. 여기서 NMOS와 PMOS란 MOSFET의 한 종류인데 MOSFET이란 전류를 통제하는 반도체 소자입니다.

각각을 알아보자면 먼저 NMOS는 n-type의 실리콘 기판 위에 Metal Gate와 Oxide Layer, Source와 Drain으로 이루어진 채널이 있습니다. NMOS의 Gate 전압이 HIGH일 때, Gate와 채널 사이에 전하를 끌어당기는 전기장이 발생하여 채널이 약해지고 Source와 Drain 사이에 전류가 흐릅니다. Gate 전압이 LOW일 때, 채널은 강해져 Drain과 Source 사이의 전류가 흐르지 않습니다. 반면에 PMOS는 p-type의 실리콘 기판 위에 Metal Gate와 Oxide Layer, Source와 Drain으로 이루어진 채널이 있습니다. PMOS의 Gate 전압이 LOW일 때, Gate와 채널 사이에 전하를 끌어당기는 전기장이 발생하여 채널이 약해지고 Source와 Drain 사이에 전류가 흐릅니다. Gate 전압이 HIGH일 때, 채널은 강해져 Drain과 Source 사이의 전류가 흐르지 않습니다.

다시 정리를 해보자면 NMOS 트랜지스터의 게이트가 HIGH일 때, 해당 트랜지스터는 전류를 통과시켜 회로를 열게 됩니다. 반면에 PMOS 트랜지스터의 게이트가 LOW일 때, 해당 트랜지스터는 전류를 통과시켜 회로를 반대 방향으로 열게 됩니다. 만약 두 개의 게이트 중 하나는 LOW이고, 다른 하나는 HIGH라면, 해당 트랜지스터는 차단되어 회로가 닫히게 됩니다.



Transmission gate는 위 사진처럼 나타낼 수 있고 S쪽이 nmos, S’쪽이 pmos라고 할 수 있습니다. Truth table은 다음과 같습니다.

|  |  |  |
| --- | --- | --- |
| A | S | Y |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | X | Z |

**2.1.2 Implement a MUX using Transmission gates**

도표, 개략도이(가) 표시된 사진

자동 생성된 설명

Transmission gates를 이용하여 MUX를 구현했습니다.

|  |  |  |  |
| --- | --- | --- | --- |
| EN | A | B | Output |
| 0 | X | 0 | 0 |
| 0 | X | 1 | 1 |
| 1 | 0 | X | 0 |
| 1 | 1 | X | 1 |

Enable이 0일때는 B의 값을 출력하고 enable이 1일때는 A의 값을 출력하는 모습을 확인할 수 있습니다. 이는 두개의 값 중 하나를 선택하는 MUX의 기능과 똑같습니다.

도표이(가) 표시된 사진

자동 생성된 설명

A부분을 봤을 때 A transmission gate는 nmos에 LOW, pmos에 HIGH가 걸려있는 모습입니다. 그래서 전류가 흐르지 않고 있는 모습을 확인할 수 있습니다. 반면에 B transmission gate에는 nmos에 HIGH, pmos에 LOW가 걸려있어 전류가 흐르므로 B의 input값인 1이 output값으로 나온 모습을 확인할 수 있습니다.

도표이(가) 표시된 사진

자동 생성된 설명

B transmission gate의 nmos에 LOW, pmos가 HIGH일 때 전류가 흐르지 않아 B의 input값이 output값으로 출력되지 않는 모습을 확인할 수 있습니다.

도표이(가) 표시된 사진

자동 생성된 설명

A transmission gate의 nmos에 HIGH, pmos에 LOW일 때 전류가 흘러 A의 input값이 output값으로 출력된 모습을 확인할 수 있습니다.

이렇게 각각의 transmission gate에 nmos와 pmos에 다른 전압을 걸어 줌으로써 둘 중 하나의 값을 결정해주는 MUX를 구현할 수 있습니다.

**2.1.3 Implement a Latch using MUXs.**

Latch는 일종의 메모리로서, 입력 신호의 상태에 따라 출력 신호를 유지하거나 변경하는 역할을 합니다. D latch에서 enable이 HIGH일 때, 데이터 입력 값인 D의 값을 Latch하여 출력으로 내보냅니다. enable이 LOW일 때는 이전 상태의 출력 값을 유지합니다. 2 to 1 MUX를 사용하여 D latch를 구현할 때는 enable에 따라서 D입력을 출력할 지 이전 상태의 출력을 출력할 지 선택합니다.

달력이(가) 표시된 사진

자동 생성된 설명

enable이 0일 때 이전 상태의 값을 출력하고 enable이 1일 때 D입력 값을 출력하도록 latch를 구현했습니다. Q를 mux의 출력 값이라고 하겠습니다.

|  |  |  |
| --- | --- | --- |
| EN | D | Qn+1 |
| 0 | Q | Q(유지) |
| 1 | 0 | 0(reset) |
| 1 | 1 | 1(set) |

도표이(가) 표시된 사진

자동 생성된 설명

Enable이 0일 때 D의 input값이 출력되지 않는 모습을 확인할 수 있습니다.

도표이(가) 표시된 사진

자동 생성된 설명

Enable이 1일 때는 output값이 D의 input값이 나온 모습을 확인할 수 있고 output값이 D input값 이외의 다른 input값으로 들어간 것을 확인할 수 있습니다.

도표이(가) 표시된 사진

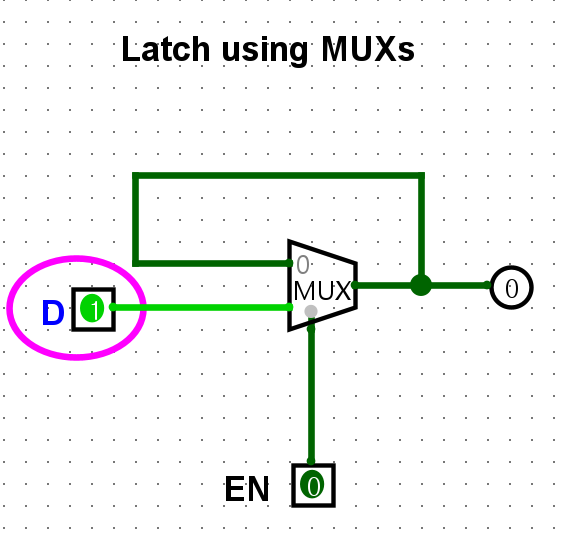
자동 생성된 설명

D이외의 input값으로 이전 상태의 출력 값이 들어갔기 때문에 enable이 0이고 D의 input도 0일 때 이전 출력 값이 그대로 출력되는 모습을 확인할 수 있습니다.

도표이(가) 표시된 사진

자동 생성된 설명

다음으로 enable이 1이 되었을 때 D input값을 다시 출력하고 다른 input값으로 output값인 0이 입력되는 모습을 확인할 수 있습니다.



Enable이 0일때 D 이외의 input값이 이전 상태의 출력 값인 0이 들어와 있으므로 D의 input값인 1이 아니라 0을 출력하는 모습을 확인할 수 있습니다.

이처럼 MUX를 이용하여 입력신호의 상태에 따라 출력 신호를 유지하거나 변경하는 latch를 구현하였습니다.

**2.1.4 Implement D-FF using Latches**

도표이(가) 표시된 사진

자동 생성된 설명

다음으로 latch를 이용한 D flip flop입니다. 이때 latch는 mux를 이용해 구현한 latch를 사용했습니다.

Latch를 사용하여 D flip flop를 구현할 때 master-slave D flip flop을 이용하여 구현했습니다. 이때 clk이 mux에서의 enable역할을 합니다. 먼저 첫번째 latch를 master latch 그리고 두번째 latch를 slave latch라 하여 master latch의 output값을 slave latch의 input값으로 넣어 주었고 서로 같은 clk을 연결해 master latch는 clk이 1일 때 작동하고 not gate를 이용하여 slave latch는 clk이 0일 때 작동하게 구현하였습니다. Master latch의 output값이 Q(M)이고 slave latch의 output값을 Q(S)이라고 하겠습니다.

|  |  |  |  |
| --- | --- | --- | --- |
| Clk | D | Q(M) | Q(S) |
| Rising edge | 0 | 0 | Q(S) |
| Rising edge | 1 | 1 | Q(S) |
| Falling edge | x | Q(M) | Q(M) |

도표이(가) 표시된 사진

자동 생성된 설명

Clk이 1이고 D가 1일 때 master latch에는 MUX를 통해 D의 input값인 1을 출력하고 출력 값이 slave master의 input값으로 들어갑니다. 하지만 slave latch에서는 clk이 0이라서 작동하지 않습니다.

도표이(가) 표시된 사진

자동 생성된 설명

Clk이 0일 때 master latch는 동작하지 않고 slave latch에서는 clk이 1이 들어와 MUX를 통해 master latch에서 들어온 input값인 1을 출력하게 됩니다.

도표이(가) 표시된 사진

자동 생성된 설명

Clk이 1이고 D의 input값으로 0이 들어왔을 때 master latch에서는 D의 input값을 출력해 slave latch의 input값으로 들어가지만 slave latch에서는 clk이 0이므로 이전 상태의 출력 값이 1을 출력하는 모습을 볼 수 있습니다.

도표이(가) 표시된 사진

자동 생성된 설명

그 이후 clk이 일 때 master latch에서는 이전 상태의 출력 값인 0을 출력하게 되고 slave latch에서는 clk이 이므로 master latch에서 넘어 온 출력 값이 입력되어 이 값을 출력하게 됩니다. 따라서 0이 출력되는 모습을 확인할 수 있습니다.

**2.1.5 Implement Synchronous Up/Down Counter using D-FFs and adders**

**도표이(가) 표시된 사진

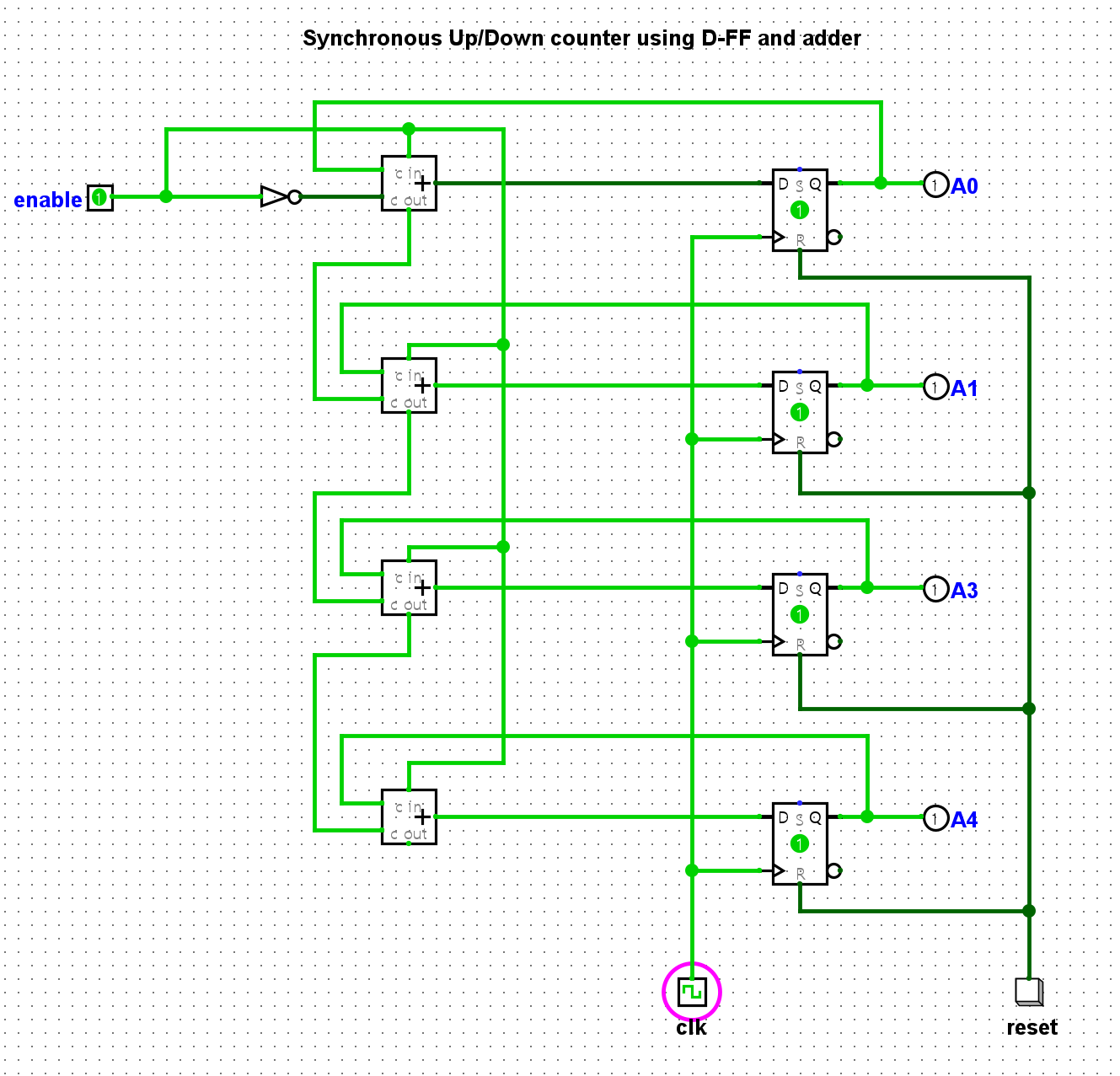
자동 생성된 설명**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| clk | Enable(up) | output | clk | Enable(down) | output |
| ↑ | 0 | 0001 | ↑ | 1 | 1111 |
| ↑ | 0 | 0010 | ↑ | 1 | 1110 |
| ↑ | 0 | 0011 | ↑ | 1 | 1101 |
| ↑ | 0 | 0100 | ↑ | 1 | 1100 |
| ↑ | 0 | 0101 | ↑ | 1 | 1011 |
| ↑ | 0 | 0110 | ↑ | 1 | 1010 |
| ↑ | 0 | 0111 | ↑ | 1 | 1001 |
| ↑ | 0 | 1000 | ↑ | 1 | 1000 |
| ↑ | 0 | 1001 | ↑ | 1 | 0111 |
| ↑ | 0 | 1010 | ↑ | 1 | 0110 |
| ↑ | 0 | 1011 | ↑ | 1 | 0101 |
| ↑ | 0 | 1100 | ↑ | 1 | 0100 |
| ↑ | 0 | 1101 | ↑ | 1 | 0011 |
| ↑ | 0 | 1110 | ↑ | 1 | 0010 |
| ↑ | 0 | 1111 | ↑ | 1 | 0001 |
| ↑ | 0 | 0000 | ↑ | 1 | 0000 |

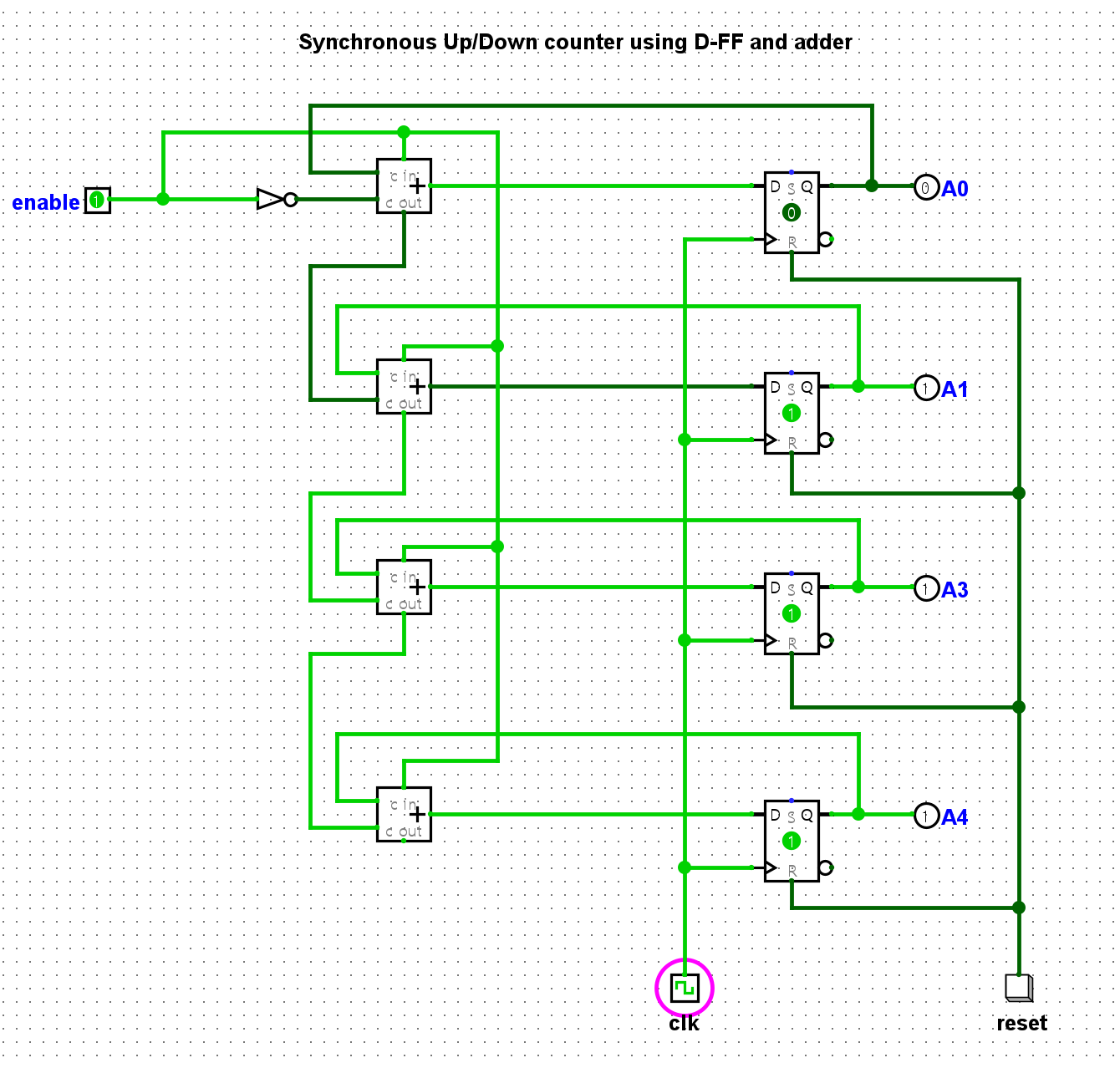
Synchronous Up/Down Counter는 카운팅 방향을 지정하여 증가 또는 감소할 수 있으며, 이 회로는 모든 flip flop이 같은 clk에 연결되어 있어 동기 회로입니다.

D flip flop은 이전 상태의 출력을 저장하고, 다음 상태의 출력을 결정하는 회로이며 이전 상태의 출력과 현재 입력을 조합하여 다음 상태의 출력을 결정합니다.

Enable이 0일 때 count가 1씩 증가하며 Enable이 1일 때 count가 1씩 감소합니다. Up counter로써 작동하는 경우에는 carry out의 값이 이전 단계에서 발생한다면 현재 단계에서 가지고 있는 값과 더합니다. 이를 통해 enable이 0이고 clk이 들어올 때 마다 1씩 증가하는 모습을 확인할 수 있습니다. Down counter로써 동작하는 경우에는 enable에 1을 넣어주고 모든 bit에 1을 들어온 다음 clk이 들어올 때마다 1씩 감소하는 모습을 확인할 수 있습니다.

****

Enable이 1이고 clk이 들어오면 모든 비트에 1이 들어온 것을 확인할 수 있습니다.

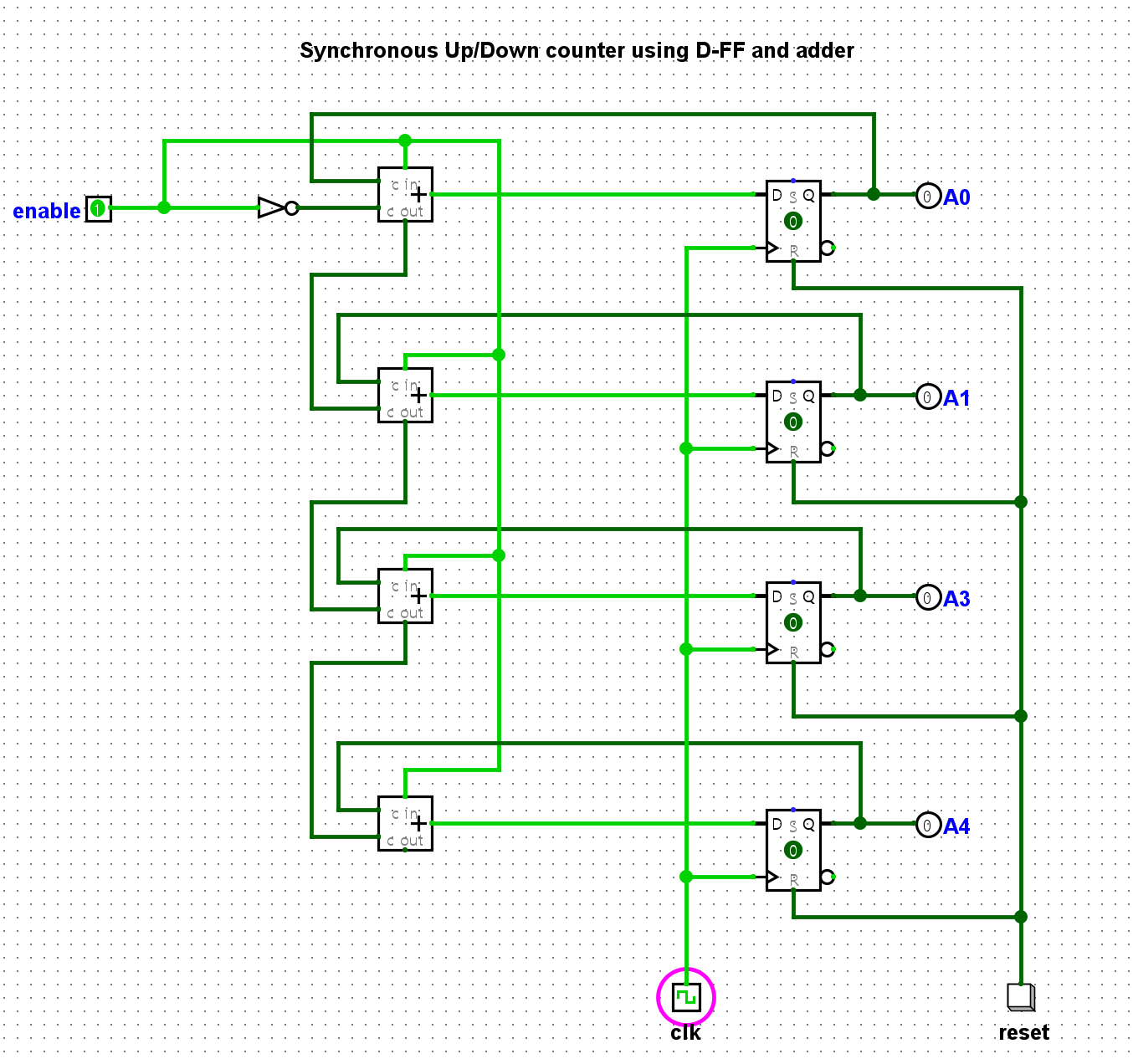


Clk이 한번 더 들어왔을 때 1이 감소되어 출력되어 나오는 모습을 확인할 수 있습니다.

테이블이(가) 표시된 사진

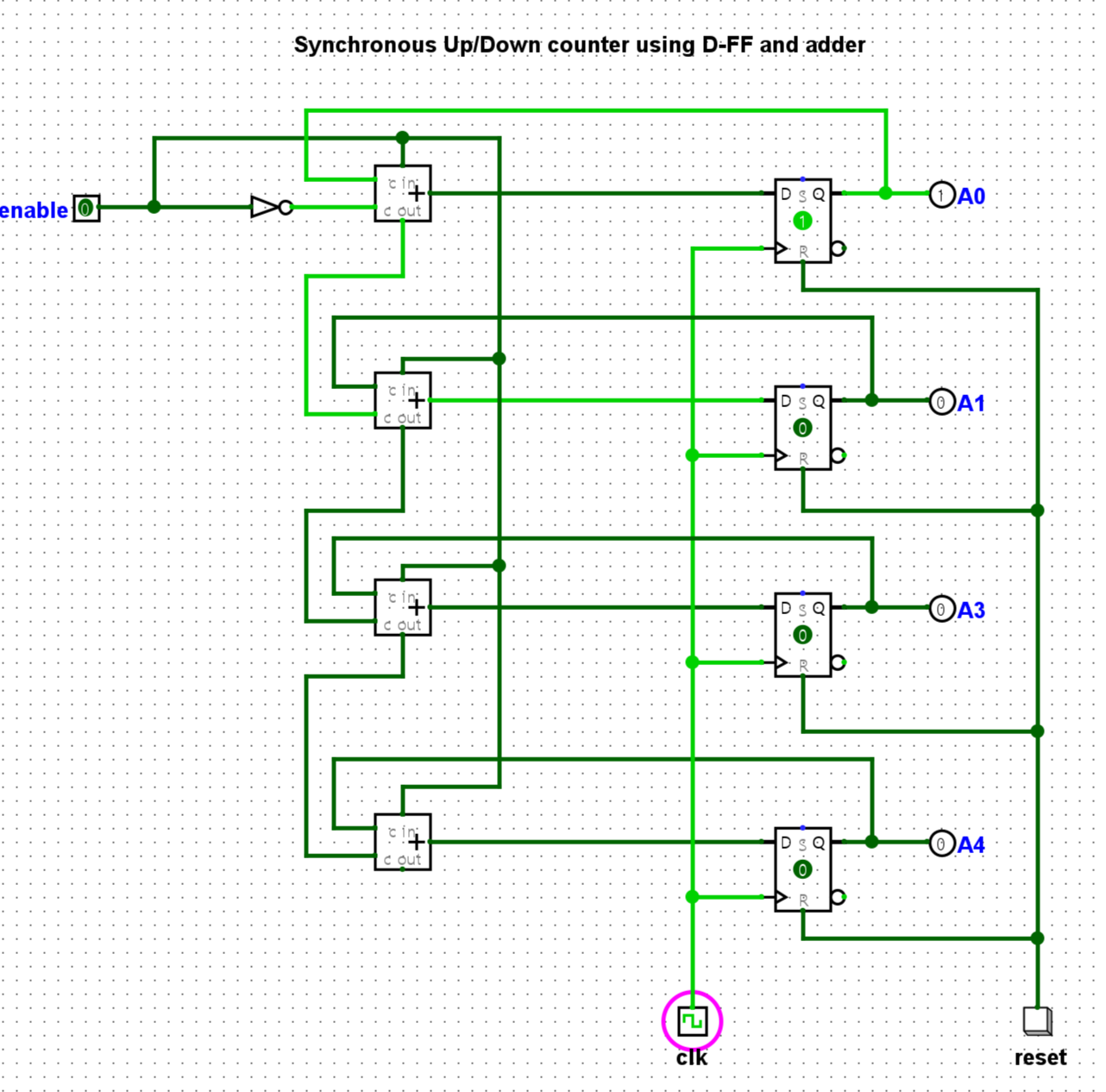
자동 생성된 설명

이후 clk이 한번 더 들어오면 1이 또 감소되는 모습을 확인할 수 있습니다.

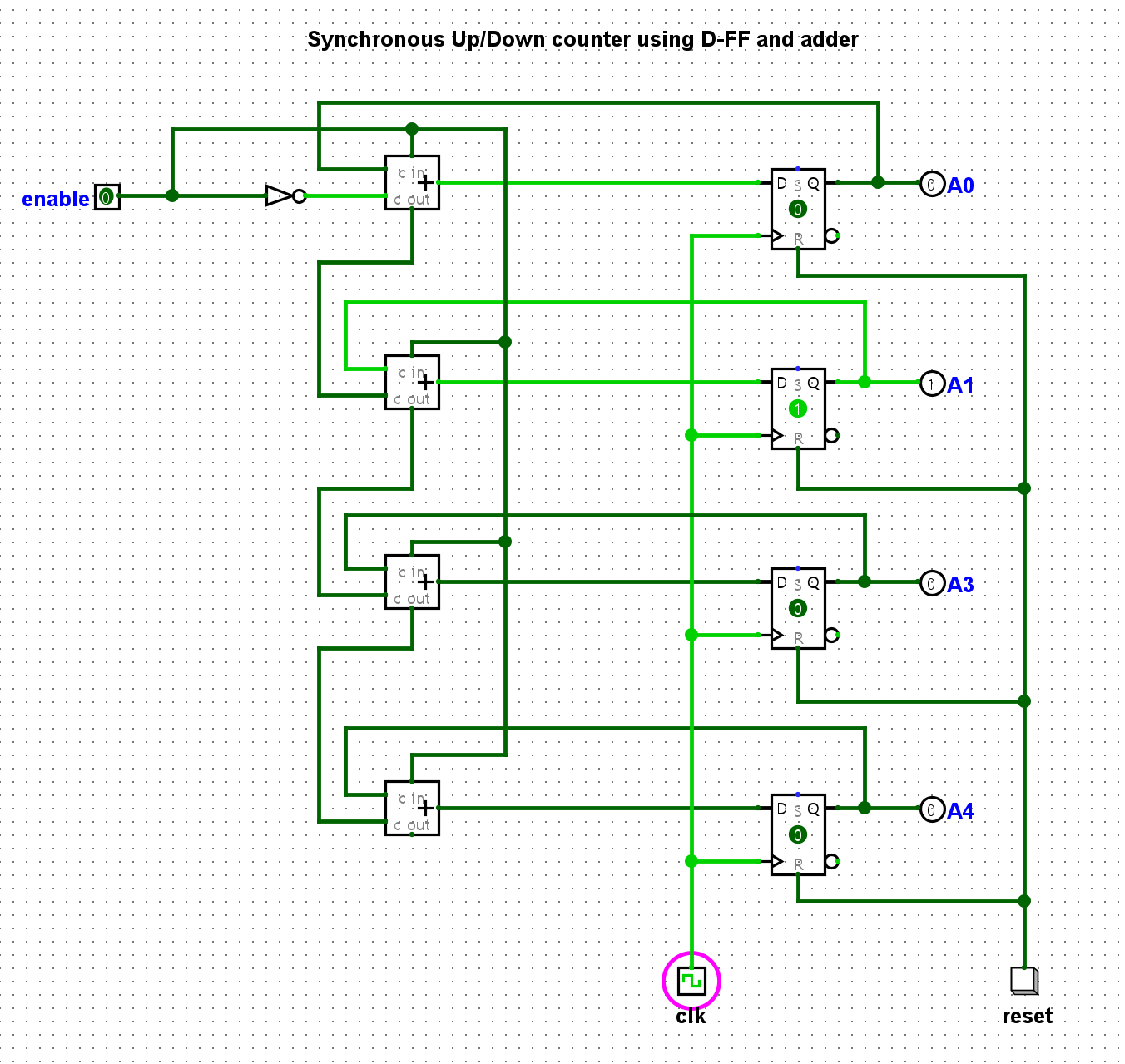


Clk이 계속 들어와 1씩 감소해 모든 비트의 출력 값이 0이 된 모습을 확인할 수 있습니다. 이렇게 clk이 들어올 때마다 1씩 출력 값이 1씩 감소되는 Synchronous down counter를 구현했습니다.

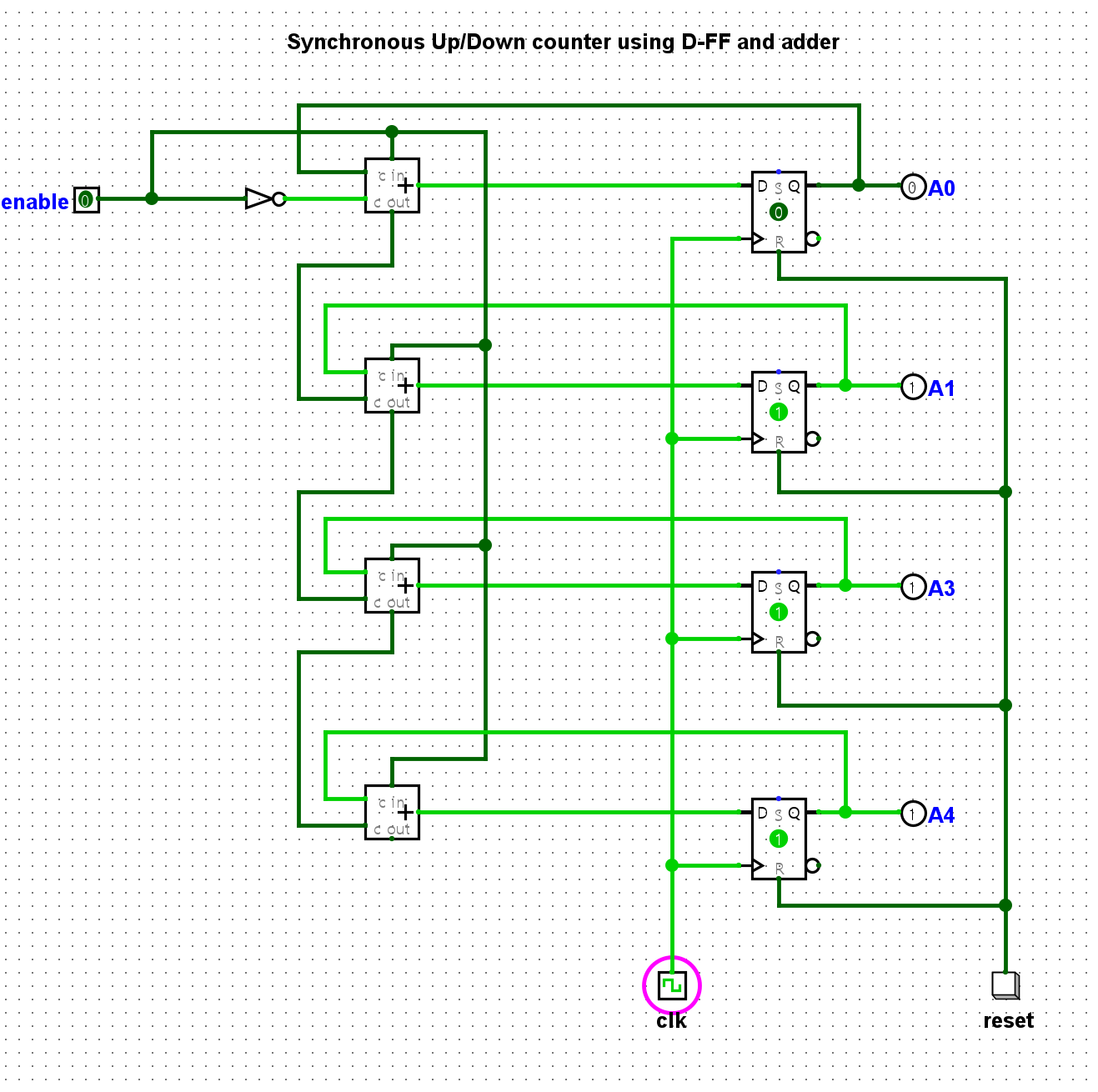
다음으로 Synchronous up counter를 알아보겠습니다.

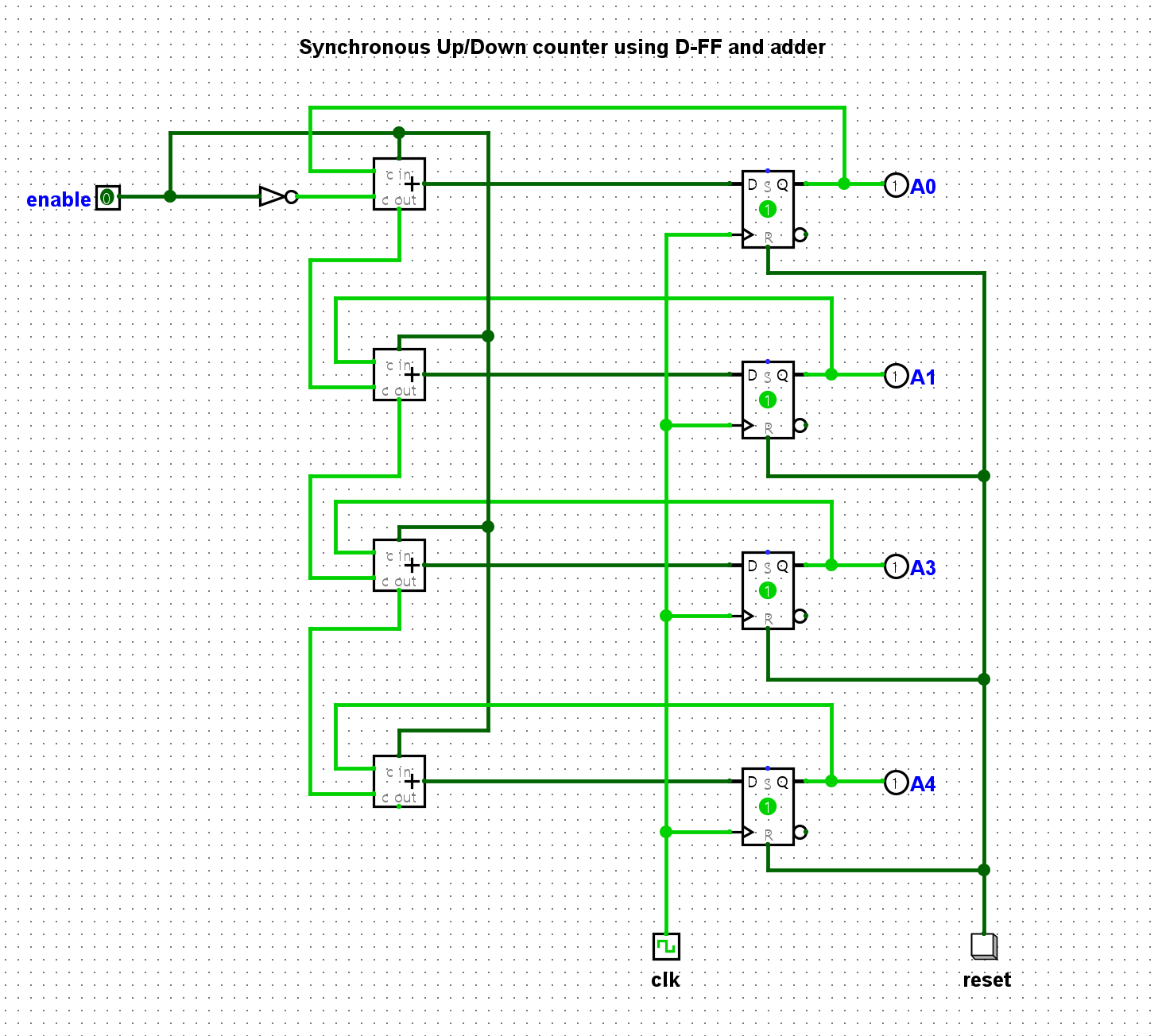


Enable이 0이고 clk이 한번 들어오면 A의 0번째 bit의 값이 1이 증가한 것을 확인할 수 있습니다.



이전 단계에서 clk이 한 번 더 들어와 adder에서 올림수가 발생해 다음 A1비트로 넘어가 1이 출력되고 A0비트는 0이 출력된 모습을 확인할 수 있습니다.



****

Clk이 계속 들어와 출력 이 1110이 된 상태에서 clk이 한 번 더 들어와 1이 더해져 1111이 출력된 모습을 확인할 수 있습니다. 이렇게 clk이 한번 들어올 때마다 출력 값이 1씩 증가하는 Synchronous Up Counter를 만들었습니다.

**2.1.6 Compare Synchronous counter and Asynchronous counter such as a ripple counter**

Synchronous counter와 Asynchronous counter이 두 counter는 모두 flip flop에 clk신호가 들어옵니다. Synchronous counter는 모든 flip flop에 하나의 clk신호가 들어갑니다. 하지만 Asynchronous counter는 첫 번째 flip flop에만 clk신호가 들어오고 다음 flip flop부터는 이전 flip flop의 출력 값이 신호로 들어간다는 점에서 Synchronous counter와 차이점이 있습니다. 이는 회로를 간단하게 구현할 수 있지만 앞 flip flop에서의 출력 값이 전달되기 때문에 이때 전달 지연 시간이 발생할 수 있습니다.

**2.1.7 What makes that Asynchronous counter is rarely used?**

앞에서 말했듯이 비동기식 카운터는 첫번째 flip flop에만 clk이 들어오고 다음 flip flop부터는 각 flip flop의 출력을 clk값으로 받습니다. 이는 카운터의 동작이 불안정해지고 오류가 발생할 가능성이 높기 때문에 설계 또한 어렵습니다.

다시 말하자면 비동기식 카운터는 입력 신호에 따라 출력 신호가 변하는 비동기의 원리를 따르기 때문에 카운터의 크기가 커질수록 설계가 어려워지며 오류가 발생할 가능성이 높아집니다. 그리고 입력 신호가 동시에 변화할 때 카운터의 동작이 불안정해지는 경우도 많습니다. 동작이 불안정해지고 오류가 발생할 가능성이 높아지면 설계자 또한 예측하지 못한 오류가 생겨 디버깅도 어렵고 시간도 많이 필요합니다. 이러한 이유 때문에 비동기식 카운터 보다는 입력신호가 동기화되어 설계가 쉽고 동작이 안정적인 동기식 카운터를 대부분의 경우 사용합니다.

**2.2 Complete the implementation of MU0 (2 cycle CPU)**

다음 표는 MU0\_ram.txt 파일을 보고 어떻게 동작할 지에 대해 예상한 결과입니다.

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction | Assembly-language | PC | result |
| 0400 | LDA 400 | 00 | ACC=MEM[400]=20 |
| 2400 | ADD 400 | 01 | ACC=ACC+MEM[400]=40 |
| 1800 | STO 800 | 02 | MEM[800]=40 |
| 0401 | LDA 401 | 03 | ACC=MEM[401]=5555 |
| 2402 | ADD 402 | 04 | ACC=ACC+MEM[402]=5555+00aa=55ff |
| 3403 | SUB 403 | 05 | ACC=ACC-MEM[403]=55ff-0055=55aa |
| 1801 | STO 801 | 06 | MEM[801]=55aa |
| 4010 | JMP 010 | 07 | JUMP to PC=10 |
| 7000 | STP | 08 | Stop |
| 0600 | LDA 600 | 09 | ACC=MEM[600]=1234 |
| 3600 | SUB 600 | 0a | ACC=ACC-MEM[600]=0000 |
| 0616 | LDA 616 | 0b | ACC=MEM[616]=0000 |
| 2600 | ADD 600 | 0c | ACC=ACC+MEM[600]=0000+1234 |
| 1803 | STO 803 | 0d | MEM[803]=1234 |
| 6000 | JNE 000 | 0e | ACC!=0, JUMP to PC=00 |
| 7000 | STP | 0f | Stop |
| 0500 | LDA 500 | 10 | ACC=MEM[500]=7fff |
| 2501 | ADD 501 | 11 | ACC=ACC+MEM[501]=7ffff+0001=8000 |
| 5008 | JGE 008 | 12 | ACC>=0, but 8000<0 |
| 3501 | SUB 501 | 13 | ACC=ACC-MEM[501]=8000-0001=7fff |
| 1802 | STO 802 | 14 | MEM[802]=7fff |
| 5009 | JGE 009 | 15 | ACC>=0, JUMP to PC=09 |
| 7000 | STP | 16 | Stop |

**도표이(가) 표시된 사진

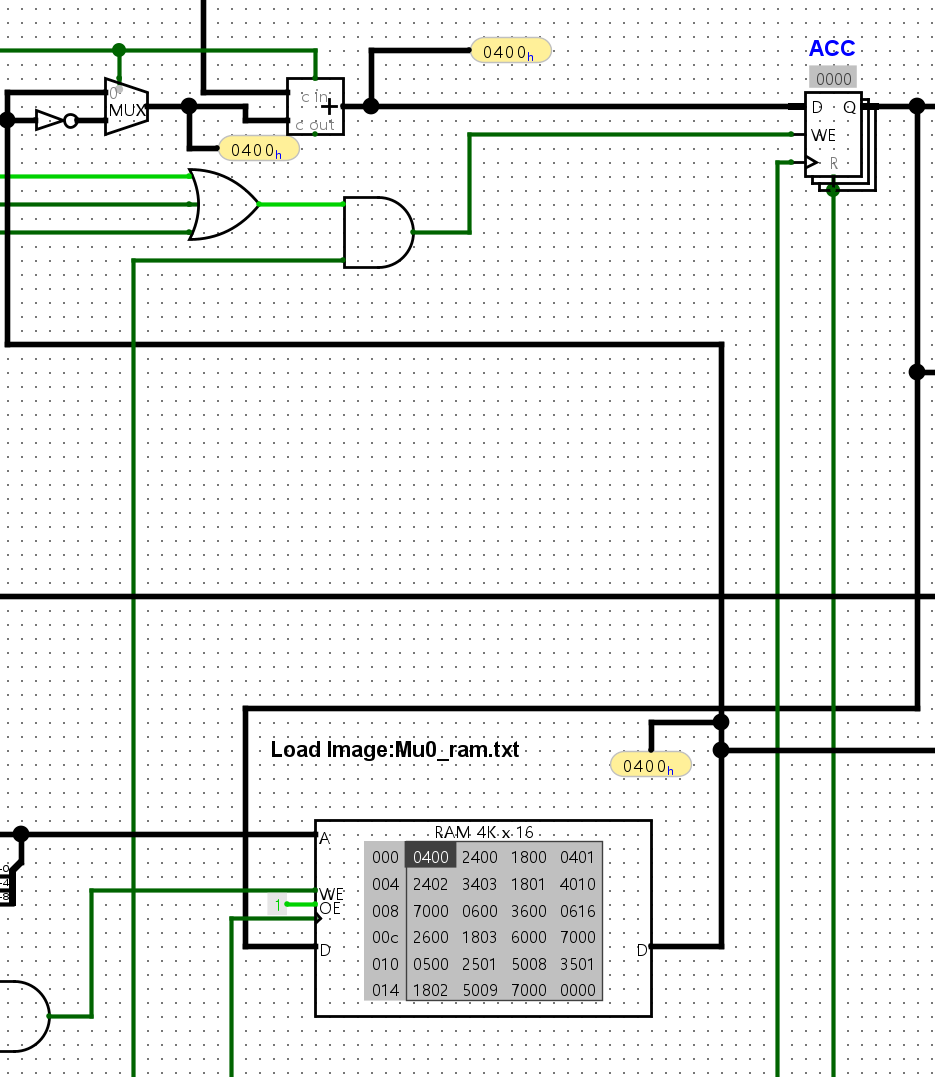
자동 생성된 설명**

먼저 설계한 회로도는 위 사진과 같습니다. 오른쪽에 있는 decoder에 wire들을 알맞게 연결하였습니다. decoder에서 모든 wire를 tunnel을 이용해서 동일한 이름을 이용해 입력과 출력을 연결해주었습니다. 그리고 0번째 비트부터 6번까지 비트는 opcode의 0번부터 6번까지의 해당하는 wire를 연결시켜주었고 7번째 비트부터 15번째 비트까지는 nor게이트를 통해 opcode의 7번에 해당하는 wire를 연결시켜주었습니다.

그럼 각 wire에 해당하는 명령어를 설명하겠습니다.

1. W1, W2, W3

Opcode 0000에 해당하는 LDA 명령어를 알아보겠습니다. LDA명령어는 메모리에서 S번지에 저장된 값을 읽어서 ACC레지스터에 저장하는 역할을 합니다. 먼저 ACC레지스터에 입력 값으로 들어가는 선 중 가장 관련이 있는 w1, w2, w3을 살펴보겠습니다. 일단은 메모리에서 읽어온 값을 ACC레지스터에 저장해야 하기 때문에 메모리에서 나온 값이 mux로 들어가는 부분을 살펴보겠습니다.

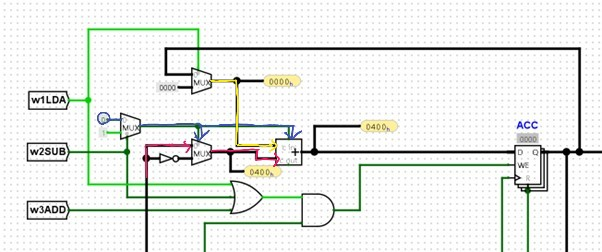


메모리에서 나온 값이 mux를 들어갈 때 하나의 값은 select가 0일 때 출력되고 다른 하나의 값은 select가 1일 때 not gate를 출력됩니다. 이때 메모리에서 가져온 값을 not gate를 통해 출력한다면 LDA의 기능을 하지 못할 테니 mux의 input 0 부분을 통해 출력되어야 한다는 점을 알 수 있습니다.

도표이(가) 표시된 사진

자동 생성된 설명

이제 빨간색 화살표 부분으로 메모리가 들어오기 위해서는 파란색 화살표가 나오는 mux에서 select가 0이고 그럼 0이 출력되어 빨간색 화살표 부분의 mux에서 select가 0이 되어야 한다는 점을 알 수 있습니다. 이를 통해 w2의 값은 0이어야 하기 때문에 w2는 LDA가 아니라는 점을 알 수 있습니다.



이제 메모리에서 나온 값이 Adder를 만나게 되는데 이는 노란색 화살표에서 나온 값과 더해서 ACC레지스터에 저장되게 됩니다. 메모리에서 읽어온 값이 레지스터에 그대로 저장되기 위해서는 노란색 화살표가 나오는 mux에서 0000값이 출력되어 이 값과 Adder에서 더해져 레지스터에 저장되어야 합니다. 아니면 mux부분에서 ACC에서 출력된 값과 더해져서 들어가게 되는데 이 값은 메모리에서 읽어온 값이 변형될 수 있습니다. 이를 통해 노란색 화살표가 나오는 mux부분에서 select가 1이 되어야 하므로 w1이 1이 되어야 합니다. 이를 통해 w1이 LDA명령어라는 점을 알 수 있습니다. 그리고 만약 w1이 0이라면 노란색 화살표가 나오는 부분의 mux에서 select가 0이므로 ACC에서 출력된 값과 메모리에서 출력된 값이 Adder를 통해 출력되니 여기서 SUB와 ADD명령어를 유추할 수 있습니다. 만약에 w2가 1이라면 w1이 0이므로 노란색 화살표가 나오는 mux부분에서 ACC레지스터에서 나온 값이 출력되어 adder로 들어가고 파란색 화살표가 나오는 mux부분에서 select가 1이되어 빨간색 화살표가 나오는 mux에서 메모리에서 나온 값이 not gate를 통해 1의 보수가 되어 adder로 들어가고 c\_in의 값으로 1과 더해져 2의 보수로 출력되는 것을 알 수 있습니다. 2의 보수의 값으로 ACC레지스터에 들어가므로 w2는 SUB명령어라는 점을 알 수 있습니다. 마지막으로 w3이면 w1과 w2가 0이므로 ACC레지스터에서 출력된 값과 메모리에서 읽어온 값이 Adder에서 더해져 ACC레지스터에 들어가는 것을 알 수 있습니다. 이를 통해 w3은 ADD명령어라는 점을 알 수 있습니다.

2. W4, W5

도표이(가) 표시된 사진

자동 생성된 설명

다음으로 이 부분을 살펴보겠습니다. ACC레지스터에서 연산 되어 나온 값이 Comparator를 만나게 됩니다. 무언가를 비교하여 값을 전달하겠다는 말은 조건을 통해 값을 전달하겠다는 말이므로 이 부분은 조건 분기와 관련 있을 것이라고 생각할 수 있습니다. JGE 명령어는 ACC레지스터에 저장된 값이 0보다 크거나 같은 경우에 지정된 메모리 주소로 점프하는 조건 분기 명령어입니다. Comparator에서 0보다 크거나 같은 값이 전달되기 위해서는 w4가 무조건 1이여야 하므로 w4는 JGE명령어라는 점을 알 수 있습니다.

JNE 명령어는 ACC레지스터에 저장된 값이 0이 아닌 경우에 지정된 메모리 주소로 점프하는 조건 분기 명령어입니다. 회로를 보면 comparator에서 0과 비교해서 0과 같을 때 NOT gate를 거쳐 전달됩니다. 이 0이 아닌 값이 전달되기 위해서는 w5가 무조건 1이어야 하므로 w5는 JNE명령어라는 점을 알 수 있습니다.

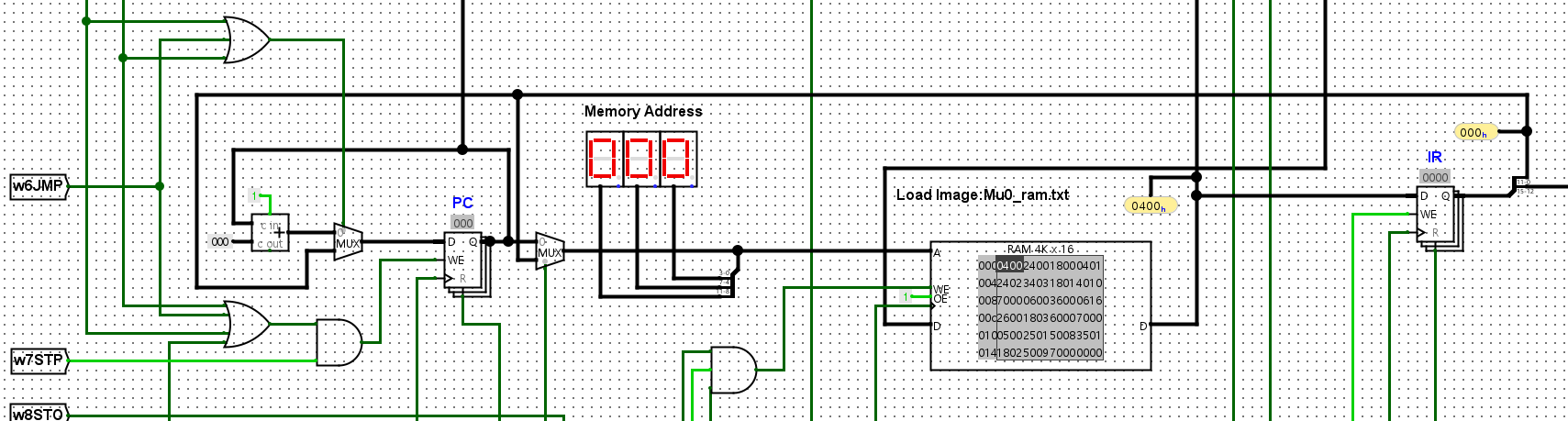
3. W8

도표, 개략도이(가) 표시된 사진

자동 생성된 설명

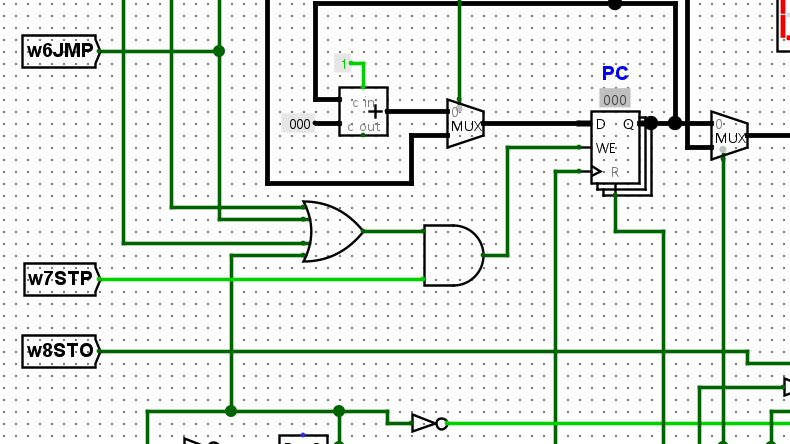
다음으로 이 부분을 살펴보겠습니다. 만약에 w8이 0이라면 AND gate는 무조건 0이므로 메모리에서 WE가 1이 될 수 없습니다. WE가 0이라면 메모리에 값을 저장할 수 없으니 w8은 STO명령어라는 점을 알 수 있습니다.

4. W6

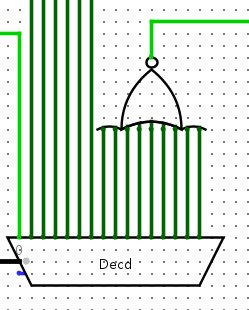


다음으로 w6부분을 살펴보겠습니다. 위 사진을 보면 IR레지스터에서 나온 값이 mux로 들어가는 것을 확인할 수 있습니다. mux에서 select가 0이라면 PC레지스터에서 나온 값에 1을 더한 값이 PC레지스터에 저장되게 됩니다. 만약 select가 1이라면 IR레지스터에서 나온 값이 그대로 mux를 통해 PC레지스터에 저장되는 것을 알 수 있습니다. 만약 w6이 1이라면 이 값이 출력될 테니 w6은 JMP명령어라는 점을 알 수 있습니다. IR레지스터에서 명령어의 주소가 출력되고 이 주소가 바로 PC레지스터에 들어가는 모습은 JMP명령어의 기능과 같기 때문입니다.

5. W7



만약 w7이 0이라면 PC의 write enable이 0이 되어 PC에 저장할 수 없게 됩니다. 그래서 회로가 작동하기 위해서는 w7이 무조건 1이 되어야 하므로 w7은 STP명령어라는 점을 알 수 있습니다.



이 decoder를 봤을 때 NOR gate로 연결되어 있는 부분이 있는데 NOR gate로 묶여 있는 wire를 제외한 wire중에서 하나가 1이 되었을 때 묶여 있는 wire는 모두 0이되어 NOR gate를 거치면 항상 1이 되므로 이는 STP명령어라는 점을 알 수 있습니다.

wire들을 명령어들과 모두 연결한 결과는 다음과 같습니다.

W1 : LDA

W2 : SUB

W3 : ADD

W4 : JGE

W5 : JNE

W6 : JMP

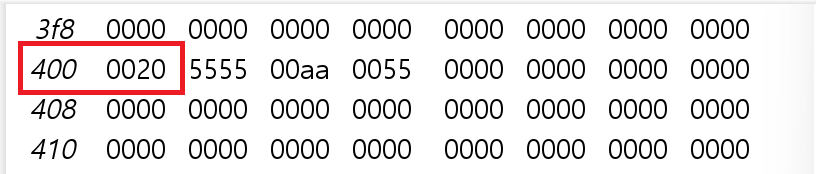
W7 : STP

W8 : STO

다음은 시뮬레이션 결과와 예상 결과의 비교 분석입니다.

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction | Assembly-language | 연산 후 PC | Accumulator value |
| 0400 | LDA 400 | 01 | ACC=MEM[400]=20 |
| 2400 | ADD 400 | 02 | ACC=ACC+MEM[400]=40 |
| 1800 | STO 800 | 03 | MEM[800]=40 |
| 0401 | LDA 401 | 04 | ACC=MEM[401]=5555 |
| 2402 | ADD 402 | 05 | ACC=ACC+MEM[402]=5555+00aa=55ff |
| 3403 | SUB 403 | 06 | ACC=ACC-MEM[403]=55ff-0055=55aa |
| 1801 | STO 801 | 07 | MEM[801]=55aa |
| 4010 | JMP 010 | 10 | 55aa |
| 0500 | LDA 500 | 11 | ACC=MEM[500]=7fff |
| 2501 | ADD 501 | 12 | ACC=ACC+MEM[501]=7ffff+0001=8000 |
| 5008 | JGE 008 | 13 | 8000 |
| 3501 | SUB 501 | 14 | ACC=ACC-MEM[501]=8000-0001=7fff |
| 1802 | STO 802 | 15 | MEM[802]=7fff |
| 5009 | JGE 009 | 09 | 7fff |
| 0600 | LDA 600 | 0a | ACC=MEM[600]=1234 |
| 3600 | SUB 600 | 0b | ACC=ACC-MEM[600]=0000 |
| 0616 | LDA 616 | 0c | ACC=MEM[616]=0000 |
| 2600 | ADD 600 | 0d | ACC=ACC+MEM[600]=0000+1234 |
| 1803 | STO 803 | 0e | MEM[803]=1234 |
| 6000 | JNE 000 | 00 | 1234 |

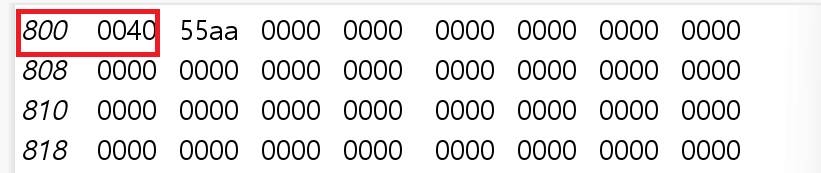
시뮬레이션 결과를 확인하니 연산 후 PC의 결과가 07->10으로 갑자기 바뀌고 15->09로 갑자기 바뀌는 구간 등이 몇몇 존재했습니다. 자세히 살펴보니 이런 구간들이 모두 jump와 관련된 명령어들이었습니다. 제가 처음에 예상했을 때는 jump명령어를 제대로 생각하지 않고 PC의 값을 계속 1씩 증가시켜 출력될 것이라고 예상했는데 이 점에서 약간의 오류가 있었던 것 같습니다. 처음에 조금 당황은 했지만 그래도 생각보다 빨리 찾아서 다행이라 생각했고 이러한 부분 외에는 예상했던 결과와 같게 출력된 것 같습니다.



도표, 개략도이(가) 표시된 사진

자동 생성된 설명

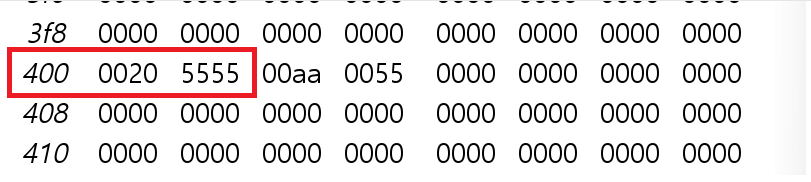
처음 LDA 400을 통해 400번지에 저장되어 있는 값 20을 ACC에 load한 것을 확인할 수 있습니다. 다음 단계에서 ADD 400을 통해 ACC에 저장되어 있는 값과 400번지에 저장되어 있는 값을 더해 ACC에 저장합니다. ACC에는 20+20의 결과인 40이 저장됩니다. 그 후 STO 800을 통해 800번지에 ACC에 저장된 값 40을 저장합니다.



800번지에 40의 값이 저장된 것을 확인할 수 있습니다.

도표, 개략도이(가) 표시된 사진

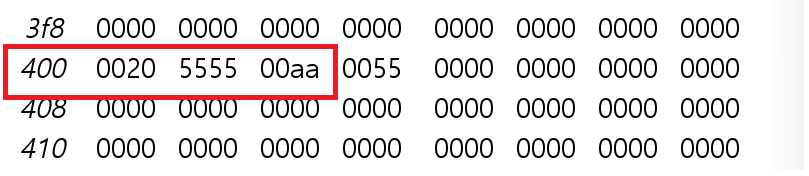
자동 생성된 설명



다음 LDA 401을 통해 401번지에 저장된 값인 5555를 load해 ACC에 저장합니다.

도표, 개략도이(가) 표시된 사진

자동 생성된 설명



ADD 402를 통해 ACC에 저장된 값이 5555와 402번지에 저장되어 있는 값인 00aa를 더합니다. 이는 55ff가 되어 ACC에 저장합니다.

도표, 개략도이(가) 표시된 사진

자동 생성된 설명

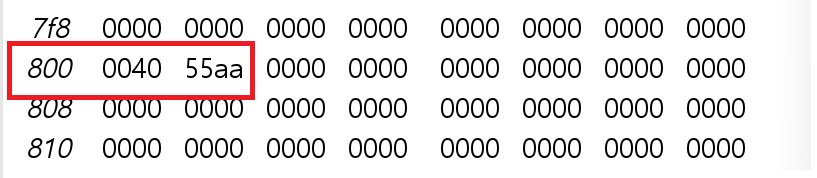
도표이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

다음 명령어인 SUB 403을 통해 ACC에 저장된 값인 55ff에서 403번지에 저장된 값인 0055를 빼서 ACC에 저장합니다. ACC에 55aa값이 저장됩니다.

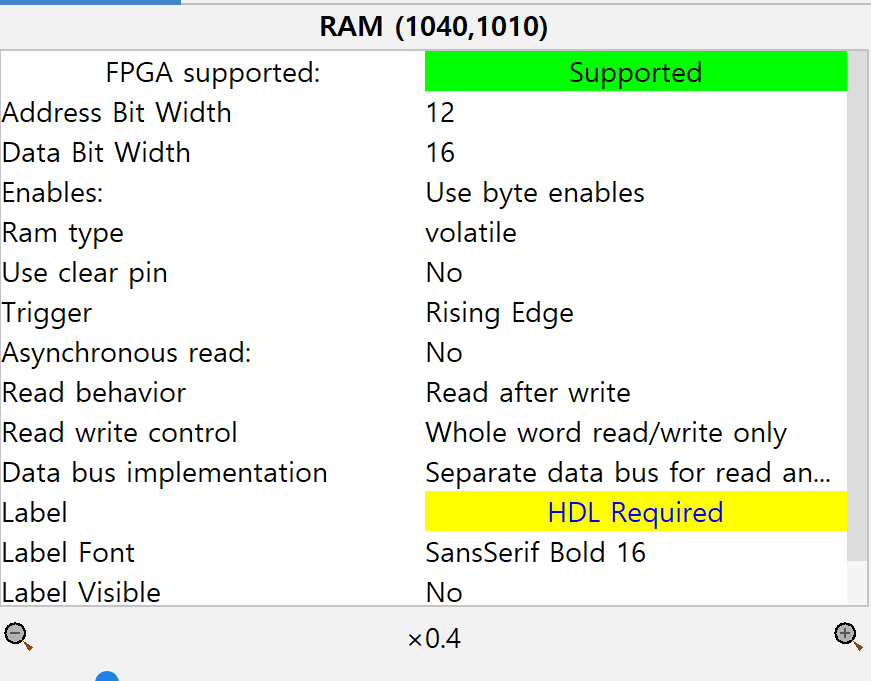


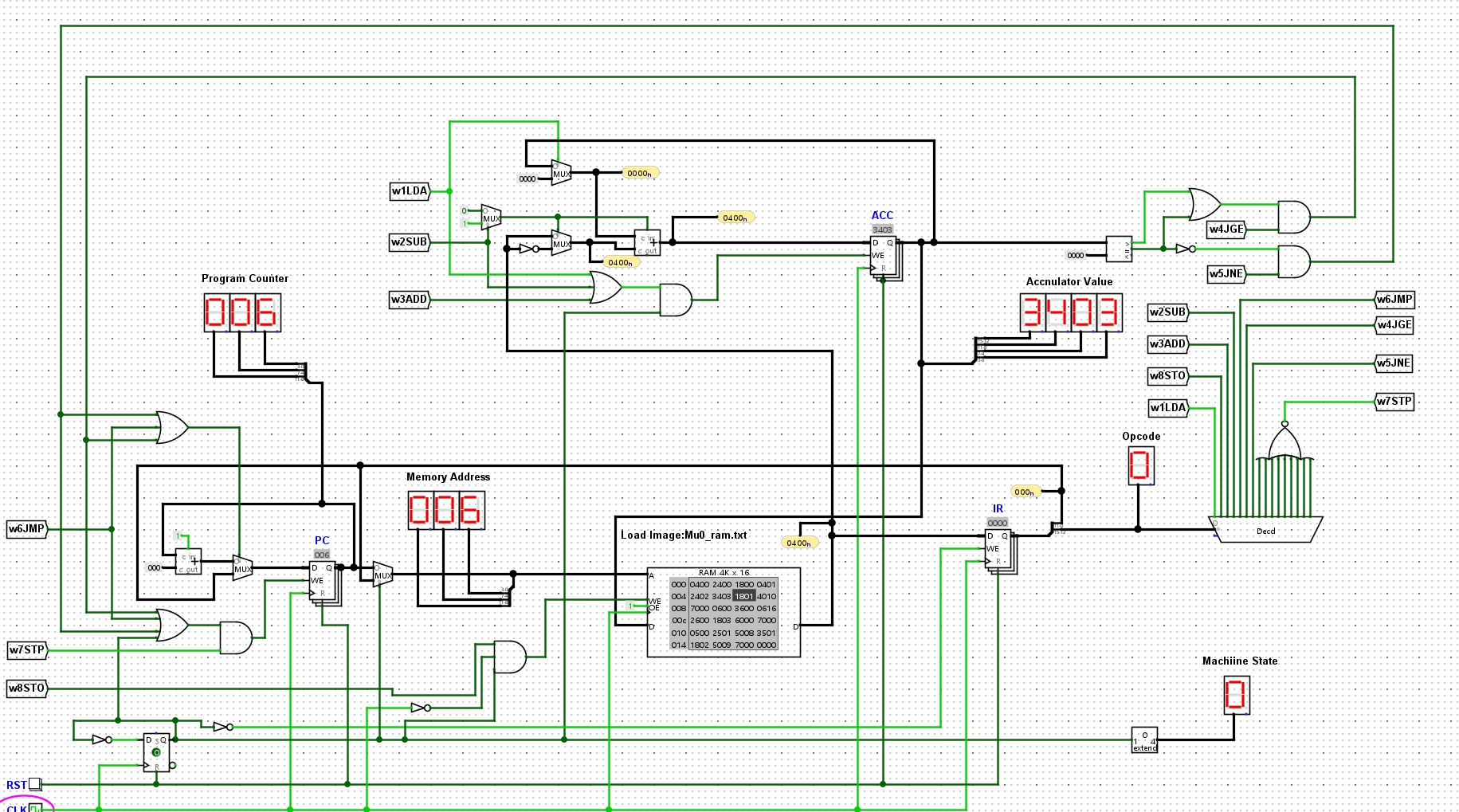
STO 801을 통해 801번지에 ACC에 저장된 값인 55aa값이 저장됩니다.

이후에도 이러한 과정이 계속 진행되며 메모리에 저장된 값을 logisim에서도 잘 출력하는 모습을 확인할 수 있습니다.

**문제점 및 개선점 기술**

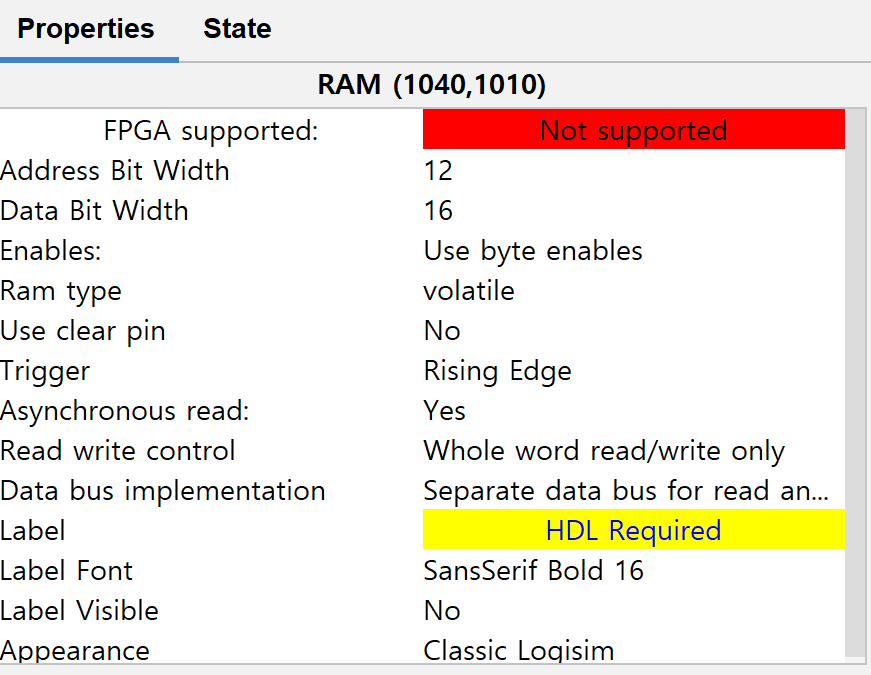
메모리 설계의 문제점을 개선하기 전입니다.

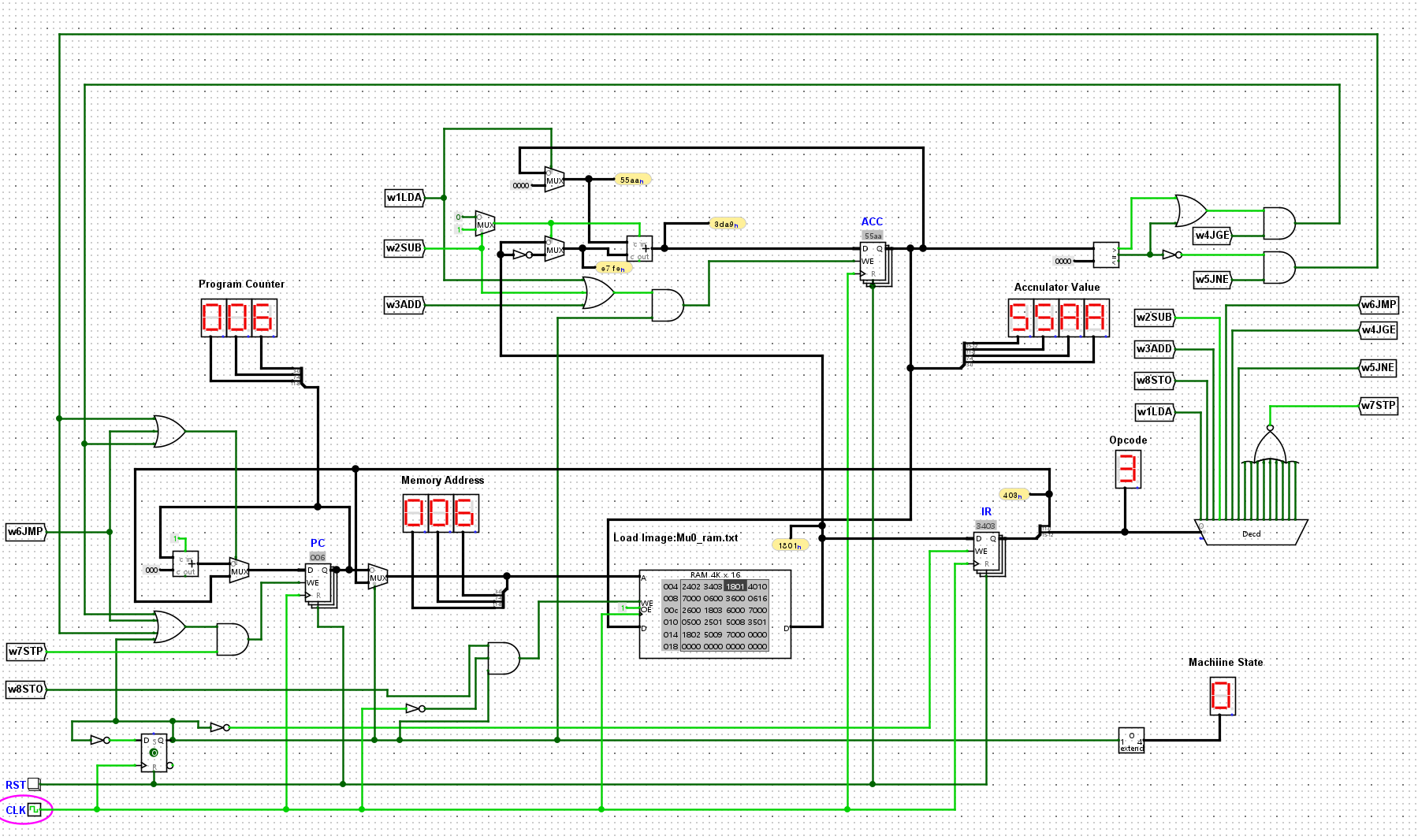




회로를 모두 구현하고 clk을 넣어보니 값이 예상한 결과와 달리 이상하게 나오고 opcode부분이 계속 변하지 않는 모습을 확인할 수 있었습니다. 왜 오류가 나오는 지 확인하던 중 RAM이 동기식으로 되어있는 부분을 확인할 수 있었습니다. MU0는 fetch와 execute를 2cycle동안 반복하는데 동기식으로 진행하다 보니 fetch를 2cycle를 하고 1cycle동안 execute를 진행한다는 점을 알았습니다. 이를 해결하기 위해 메모리의 설계를 비동기식으로 바꿔주었습니다. 바꿔주니 값이 제대로 출력되는 값을 확인할 수 있었습니다.

메모리의 문제점을 개선하고 난 후입니다.





그리고 MU0 processor는 fetch때 pc값이 증가하는데 주어진 회로는 execute때 pc값이 증가하는 모습을 확인할 수 있습니다.

도표, 개략도이(가) 표시된 사진

자동 생성된 설명

그래서 위 사진처럼 저 부분에 not gate를 달아주면 fetch때 pc값이 증가하는 MU0를 구현할 수 있습니다. 저는 이 부분들이 문제점이라고 생각이 들었습니다.

그리고 다음은 개선점 부분입니다. 주어진 MU0\_ram.txt파일을 메모리에서 load해 구현한 회로를 동작하면 STP명령어를 제대로 수행하지 않고 무한 반복하는 상황을 확인할 수 있습니다. 이런 상황에서는 STP의 기능을 제대로 확인할 수 없기 때문에 MU0\_ram.txt파일을 약간 수정해야 한다고 생각했습니다.

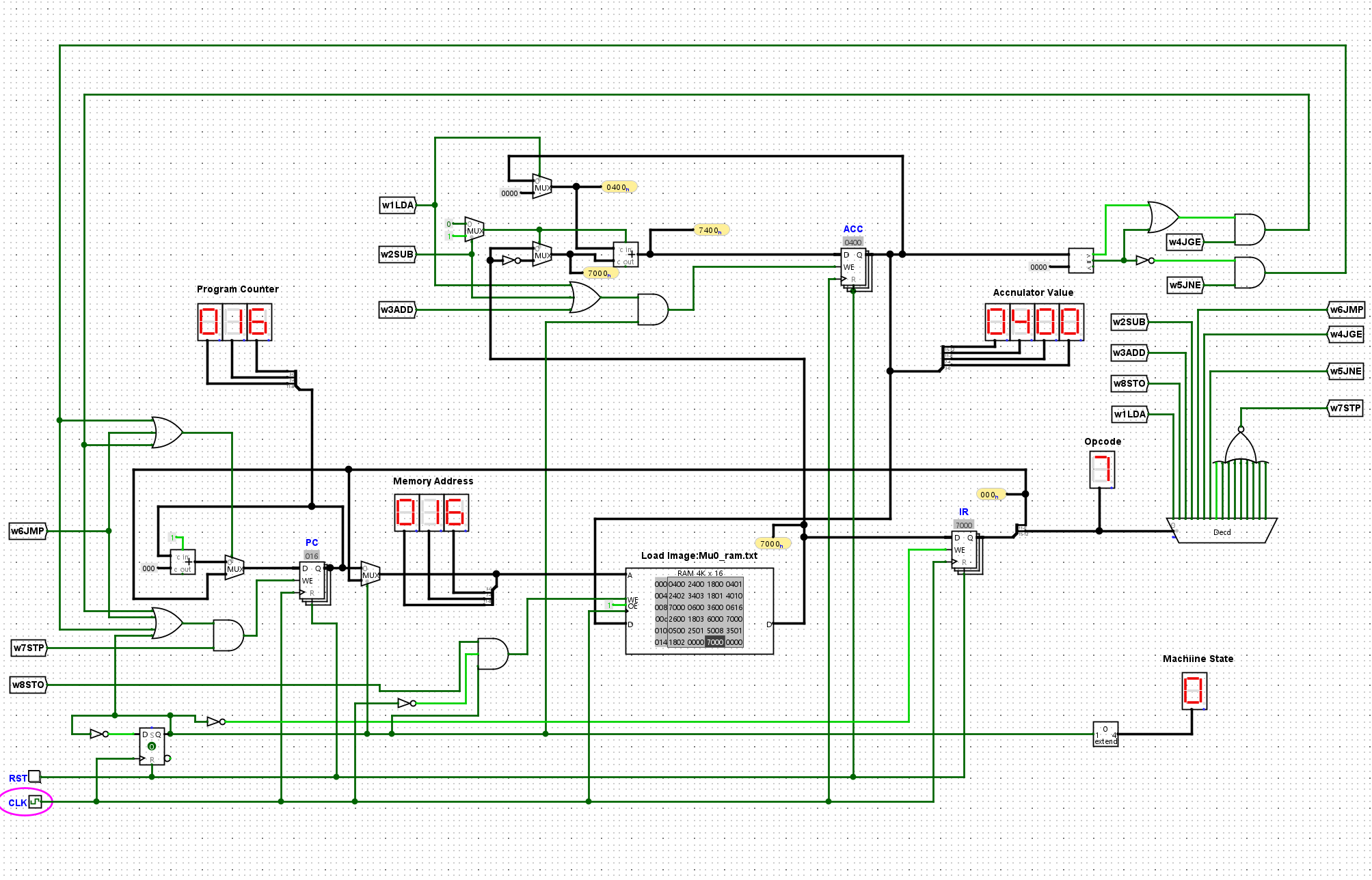
텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

기존의 파일에서 빨간색 부분을 수정해주었습니다. 수정하기 전에는 조건에 따라 JUMP를 해서 무한반복을 진행하였을 것이었습니다. 수정한 후에는 조건에 해당되지 않아 JUMP를 진행하지 않고 다음 명령어인 STP를 수행하였습니다. ACC에 저장된 값을 확인하니 값이 변하지 않는 것을 확인할 수 있었고 PC의 write enable이 꺼져 있어 PC에 값을 저장할 수 없는 것을 확인했습니다. 이를 통해 STP명령어가 제대로 수행된 것을 확인했고 Opcode에도 7이 출력된 것을 확인할 수 있었습니다.



**Reference**

Computer Architecture – Project #0 제안서

Computer Architecture Lab 4주차 강의자료