8 bit CPU

Logisim을 이용한 기본 컴퓨터 설계

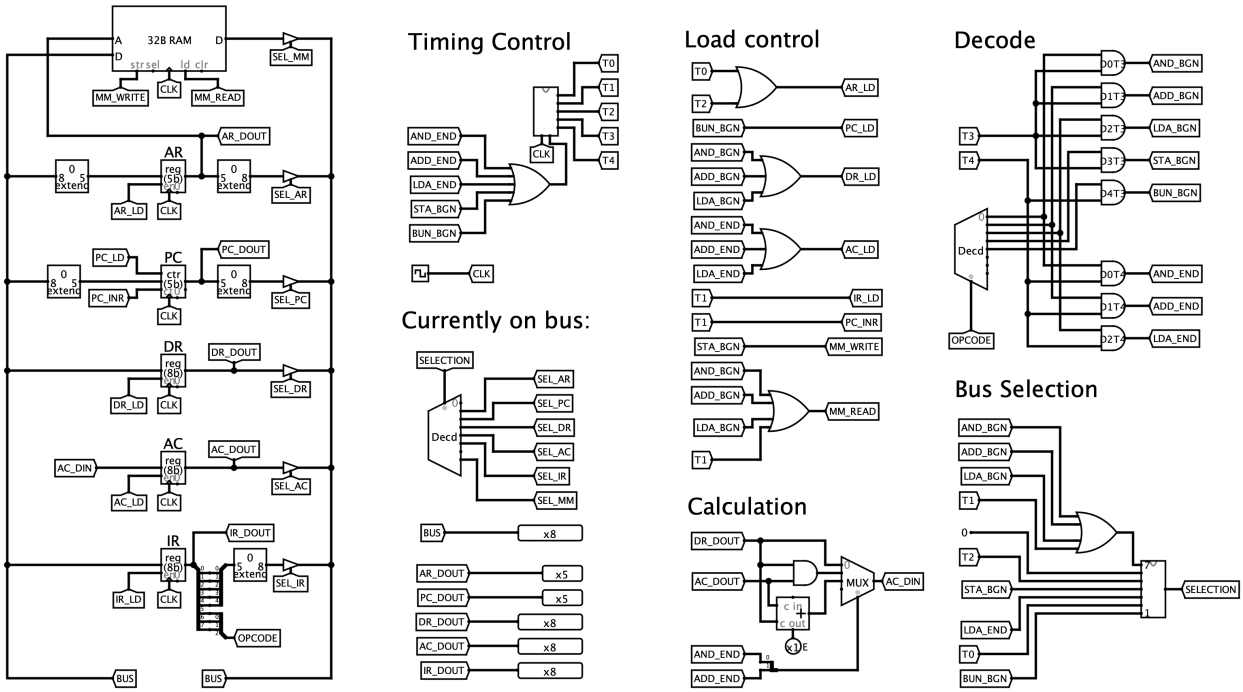
과목명 컴퓨터구조 – 금 야간

참여인원 박정희, 송병준, 이승헌, 정해인

제출일 2018. 12. 5.

목차

1. 전체 회로 개요



완성된 회로는 그림과 같다. 가장 왼쪽에 공통 버스 시스템이 존재하며 각 레지스터는 터널을 통해 컨트롤 게이트와 연결되어 있다. 명령어와 데이터는 모두 메인 메모리(RAM을 사용)에 적재되어 있고 주소 비트는 5, 데이터 비트는 8개를 사용한다. 산술 논리 장치는 AND와 ADD를 지원한다. 사용 가능한 명령은 메모리 참조 명령을 2개 포함하여 총 5개이다.

2. 요구사항 분석

요구사항은 다음과 같다:

- 8비트 명령어, 상위 3비트는 opcode, 하위 5비트는 operand에 할당.
- 직접 메모리 참조.
- AND, ADD, LDA, STA, BUN 지원
- 32 * 8bit 메인 메모리
- 다음 레지스터 사용: AR, PC, DR, AC, IR, SC, E
- 8비트 공통 버스 시스템 사용
- 가산, 논리곱 연산 지원

요구사항에는 인터럽트나 입출력이 없다. 또한 연산 또한 두 종류밖에 없다.
염두에 두어야 할 것은 명령어 세부 구현이다. 요구사항의 opcode와 구현은 다음과 같다:

AND :
000 | AC <- AC ^ M[AR], SC <- 0

ADD :
001 | AC <- AC + M[AR], SC <- 0

LDA :
010 | AC <- M[AR], SC <- 0

STA :
011 | M[AR] <- AC, SC <- 0

BUN :
100 | PC <- AR, SC <- 0

5개의 명령어의 opcode가 0부터 4까지 순차적으로 증가한다. 후에 명령 decode 로직을 설계할 때에 참고할 것이다.

3. 설계 원칙

주어진 도구—Logisim—의 목적은 쉽고 편한 simulation이다. 실제 기판에 그릴 회로를 제작하거나 극한의 최적화를 수행하는 것이 목적이 아니므로, 기계적인 요소는 배제하고 회로에서 일어나는 논리적 요소들에 집중하였다. 따라서 이 설계는 다음의 원칙을 따라야 한다:

1. 회로의 동작이 가시적이어야 할 것.
2. 설계를 유동적으로 바꿀 수 있도록 할 것.

논리와 동작이 하드웨어에 embed된 논리 회로는 그 특성상 프로그램 code처럼 쉽게 읽을 수 없다. 따라서 회로의 흐름이나 동작의 원인들을 빠르게 파악할 수 있도록 다음의 구체적인 지침 또한 정하였다:

1. 참조가 잦은 변수 또는 거리가 먼 변수는 Tunnel을 사용할 것.
2. 비슷한 기능을 하는 하드웨어를 모아 모듈로 구성할 것.
3. 각 모듈에는 Label을 두어 기능을 명시할 것.
4. 버스와 각 레지스터 출력은 항상 잘 보이도록 할 것.

4. 설계와 구현

요구사항을 충족하는 시스템에서, 모든 제어는 Sequence Control에 의해 일어난다.
가장 기본적인 제어는 처음부터 일어나는 Fetch와 Decode이다.

Fetch
T0: AR <- PC
T1: IR <- M[AR], PC <- PC + 1

Decode
T2: D0…D7 <- Decode IR(5-7), AR <- IR(0-4)

초기에 명령어를 가져오고 해석하는 과정에서 3 time이 필요하다는 사실을 알 수 있다. 후에 명령어를 수행하는 것은 T3부터 일어난다.
위의 T0..T2는 우측의 동작이 실행될 조건이다. T0일 때에는 무조건 Fetch가 시작되고, T2일 때에는 무조건 Decode가 시작된다.
이처럼 사용자의 명령어들을 실행하기 위한 조건을 정의해야 한다.

예를 들어 AND 명령은 opcode가 000이고, T3일 때에 시작되어야 한다. opcode는 3x8 디코더의 출력을 통해 판별한다.

000 -> 00000001
001 -> 00000010
010 -> 00000100
.
.
.

출력의 각 비트를 좌측부터 D7 … D0이라고 하면,
AND는 000의 opcode를 가지므로 D0일 때 일어나야 한다고 볼 수 있다. 그러면 명령이 시작되는 T3에 D0이면 AND를 수행하면 된다고 볼 수 있다.

그러나 메모리 로드와 연산, AC로의 저장은 한 번에 일어날 수 없으므로 다음과 같이 나눈다.

AND :
D0 T3: DR <- M[AR]
D0 T4: AC <- AC ^ DR, SC <- 0

이와 같이 명령어의 실행 조건을 정리하면 다음과 같다:

AND:
D0 T3: DR <- M[AR]
D0 T4: AC <- AC ^ DR, SC <- 0

ADD:
D1 T3: DR <- M[AR]
D1 T4: AC <- AC + DR, E <- Cout, SC <- 0

LDA:
D2 T4: AC <- DR, SC -> 0

STA:
D3 T3: M[AR] <- AC, SC <- 0

BUN:
D4 T3: PC <- AR, SC <- 0

명령어가 끝나는 시점에는 반드시 Sequence Counter를 0으로 초기화한다.
모든 명령어가 최대 2 time 동안 수행된다.

4 - 1. Common Bus

버스의 데이터 폭은 8비트이다. 이 설계에서는 bus selection을 위해 Multiplexor 대신 controlled buffer를 사용한다. 선택 가능한 출력은 6개—AR, PC, DR, AC, IR, Main Memory—존재하므로, 6개의 controlled buffer를 사용한다. 각 buffer의 입력은 3x8 decoder를 거쳐 나온 8개의 출력 중 1, 2, 3, 4, 5, 7번 만을 사용한다. 저 각 출력들은 다음을 뜻한다:

1: Select AR
2: Select PC
3: Select DR
4: Select AC
5: Select IR
7: Select MM

4 - 2. Address Register

다음은 주소 레지스터이다. 주소 레지스터는 데이터 입력과 출력을 모두 버스에 연결한다. 다만 데이터 폭과 버스 폭은 모두 8비트인데 반해 주소 폭은 5비트이므로 레지스터의 입출력단에 bit extender를 두어 상위 패딩 000을 추가하거나 상위 3자리를 잘라야 한다.
요구사항을 수행하는 데 있어 AR의 INR 제어는 필요 없으므로 일반 레지스터를 사용하여 구현한다.

주소 레지스터는 버스를 거치지 않고 메모리의 주소 입력으로 바로 진입하는 통로도 가지고 있어야 한다. bit extender를 거치기 전 5비트의 레지스터 출력을 메인 메모리의 주소 입력에 연결하여 준다.
주소 레지스터의 로드는 주소 레지스터에 값을 쓸 때에 일어나야 한다. 명령 중 'AR <-'을 포함하는 명령은 Fetch와 Decode로, 각각 T0, T2일 때 일어난다. 따라서 주소 레지스터의 로드도 그 때에 일어나야 한다.

AC_LD = T0 + T2

4 - 3. Program Counter

프로그램 카운터는 실행할 명령의 주소를 저장한다.
프로그램 카운터 또한 입력과 출력 모두 버스에 연결한다. 주소를 담고 있어 폭이 5비트이므로, 양측에 bit extender를 달아준다. 프로그램 카운터는 다른 레지스터와 달리 INR을 사용해야 할 일이 많으므로 Counter로 구현한다.
T1에서 Fetch가 끝나면 카운터가 1 증가하고, BUN 명령이 시작되는 D4T3에 로드된다.

PC_INR = T1
PC_LD = D4T3

4 - 4. Data Register

데이터 레지스터에는 주로 메모리에서 가져온 정보가 담긴다.
입출력 모두 버스에 연결하며, 메모리 참조를 포함하는 명령 모두에서 로드된다.
로드 조건은 (D0 + D1 + D2)T3이다.

DR_LD = D0T3 + D1T3 + D2T3

4 - 5. Accumulator

AC에는 연산 결과가 담긴다. 특이하게도, AC는 버스로부터 직접 입력을 받지 않는다.
출력만 버스에 연결하고 입력은 별도의 회로로 구성한다.
AC의 입력은 (1)DR의 출력, (2)AC와 DR의 논리곱, (3)AC와 DR의 합 세 가지 중 하나로 선택한다.
AND 명령을 분석하여 보면, 명령이 끝나는 D0T4에 AC에 논리곱 결과가 들어가야 함을 알 수 있다. ADD도 마찬가지로, D1T4에 합 결과가 들어가야 한다. 따라서 AC 입력 회로는 MUX를 이용해 다음과 같이 구성한다:

```
// C

int AND_END = D0 & T4;
int ADD_END = D1 & T4;

if (! AND_END && ! ADD_END) { AC_DIN = DR_OUT; }
else if (AND_END && ! ADD_END) { AC_DIN = DR_OUT & AC_DOUT; }
else if (! AND_END && ADD_END) { AC_DIN = DR_OUT + AC_DOUT; }
else { } // floating
```

AC의 로드는 연산이 수행되는 AND_END + ADD_END와 LDA_BGN에 일어난다.

AC_LD = D0T4 + D1T4 + D2T4

4 - 6. Instruction Register

명령 레지스터는 메모리에서 Fetch해온 명령어를 보관한다. 입력과 출력 모두 버스에 연결되어 있는데, 출력이 버스에 바로 연결되지는 않는다. 상위 3비트는 opcode로 분류하여 별도의 처리를 거치는 조합회로에 연결되며, 하위 5비트가 bit extender를 거쳐 버퍼로 전송된다.
명령 레지스터는 T1에 로드된다.

IR_LD = T1

4 - 7. Main Memory

8비트 데이터에 5비트 주소로 32*8비트의 용량을 가진 메인 메모리는 읽기와 쓰기를 모두 사용해야 하므로 RAM을 사용한다. 메모리의 입출력 모두 버스에 연결하며, AR로부터 주소 입력을 직접 받는다. 메모리를 쓰는 조건과 읽는 조건은 따로 존재하는데, 다음과 같다:

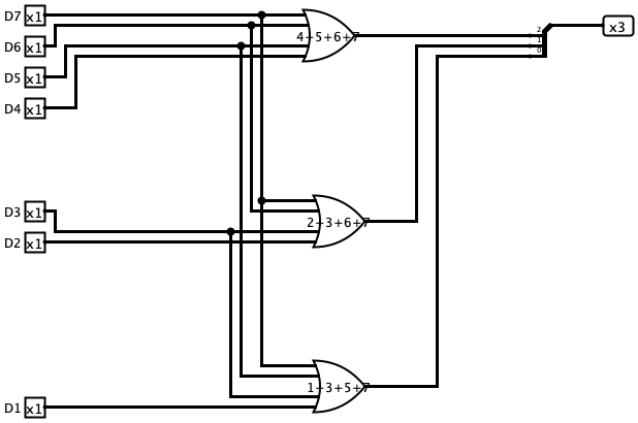
MM_WRITE = D3 T3
MM_READ = T1 + D0 T3 + D1 T3 + D2 T3

쓰기는 STA가 시작될 때, 읽기는 Fetch가 끝날 때와 메모리 참조 연산이 시작될 때에 일어난다.

4 - 8. Bus Selection

버스에 어떤 데이터를 올릴 지 결정하는 것은 중요한 문제이다. 주로 다른 레지스터로 가는 소스가 되어야 하는 레지스터가 버스에 올라가 있어야 한다.
버스 선택 데이터는 먼저 7개의 다른 입력—x1 … x7—으로부터 인코더를 거쳐 3bit로 변환된 뒤, 버스의 디코더를 거쳐 다시 8개의 입력으로 나뉘어져 controlled buffer를 제어한다.
정리하면 다음과 같다:
8bit width control gates outputs =〉
encoder =〉 3bit width selection signal =〉 decoder =〉
8 bit width bus selection signal =〉 each controlled buffers

	Decoded	Encoded
0	0000 0001	000
1	0000 0010	001
2	0000 0100	010
3	0000 1000	011
4	0001 0000	100
5	0010 0000	101
6	0100 0000	110
7	1000 0000	111



인코더에서 0번째 입력—x0—은 0일 때와 1일 때 모두 결과가 000이므로 의미가 없다. 따라서 7개의 입력만 받는 인코더를 설계한다.

인코더의 각 입력 x1, x2, x3, x4, x5, x6, x7은 각각 다음 레지스터 또는 메모리를 의미한다.

x0: none x1: AR x2: PC x3: DR
x4: AC x5: IR x6: none x7: MM

각각이 출력들이 버스로 올라가야 하는 조건들은, 위의 명령어 정의에 따르면 다음과 같다:

x0 = 0
x1 = D4 T3
x2 = T0
x3 = D2 T4
x4 = D3 T3
x5 = T2
x6 = 0
x7 = T1 + D0 T3 + D1 T3 + D2 T3

0번 연결과 6번 연결은 존재하지 않아 상수 0으로 대체하였다.

4 - 9. Decoder

명령을 해독하는 로직은 IR에 연결되어 있는 조합 회로로 구성되어 있다. 위에서 여러 차례 사용한 AND_BGN, ADD_BGN 등이 이 곳에서 만들어진다. opcode와 timing 변수들을 이용해서 다음과 같은 출력을 내보낸다:

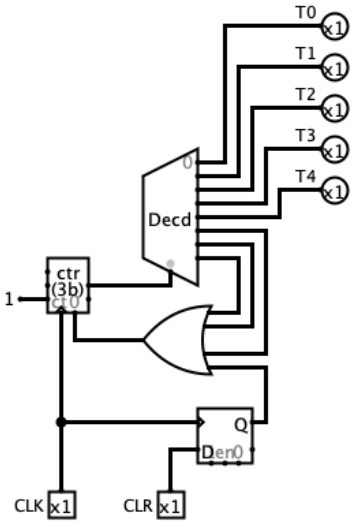
AND_BGN = D0T3
ADD_BGN = D1T3
LDA_BGN = D2T3
STA_BGN = D3T3
BUN_BGN = D4T3

명령어가 한 time에 끝나지 않는 경우에 쓰이는 다음 변수들도 있다.

AND_END = D0T4
ADD_END = D1T4
LDA_END = D2T4

4 - 10. Timing Sequencer

이 시스템에서 매우 중요한 timing 신호는 이 곳에서 만들어진다. 사용하는 신호는 T0부터 T4까지 5개인데, 5개의 상태를 구현하기 위해서 3x8 decoder와 counter를 사용한다. T5~T7이면 강제로 counter를 clear한다. 외부에서 clear 신호를 받기 위해 OR gate로 CLR 신호를 받는데, 이때 새로운 시퀀스인 T4가 활성화되자마자 비동기적으로 clear가 되면 작동에 문제가 생기므로, D flip-flop을 두어 CLR 신호가 한 클럭 늦게 들어오도록 처리한다.



항상 증가해야 하기 때문에 count 입력은 상수 1이며 출력은 시스템에서 사용하는 T0에서 T4까지의 각각 1bit 데이터이다.

5. 동작

5 - 1. Test Program 작성

메모리에 저장된 두 개의 값을 더하여 임의의 주소에 저장하고자 한다. 먼저 메모리에서 값을 읽어 AC에 저장해놓는다. 그런 다음 메모리에서 다른 값을 읽어 AC에 있는 것과 더하여 AC에 저장한다. 그리고 메모리 임의의 주소에 그 결과를 저장한다.
위 동작을 명령어로 치환하면 다음과 같다:

```
LDA   10000
ADD  10001
STA    10010
```

이를 응용하여, 특정 주소에 있는 값을 1씩 증가시키는 프로그램을 만들어 볼 것이다.
16번지에 있는 값을 가져온 뒤 17번지에 있는 값과 더해서 다시 16번지에 저장한다.

이렇게도 표현할 수 있다:

```
uint8_t target       = 0x00;
uint8_t amount     = 0x01;

while (1) {
        target += amount;
}
```

이를 시스템에 맞게 다듬어 메모리에 옮기면 다음과 같다:

AD	BIN	HEX
00	0101 0000	50
01	0011 0001	31
02	0111 0010	70
03	1000 0000	80
…		
10	0000 0000	00
11	0000 0001	01

이 프로그램의 이름은 addLoop이며, 다음은 소스 전문이다.

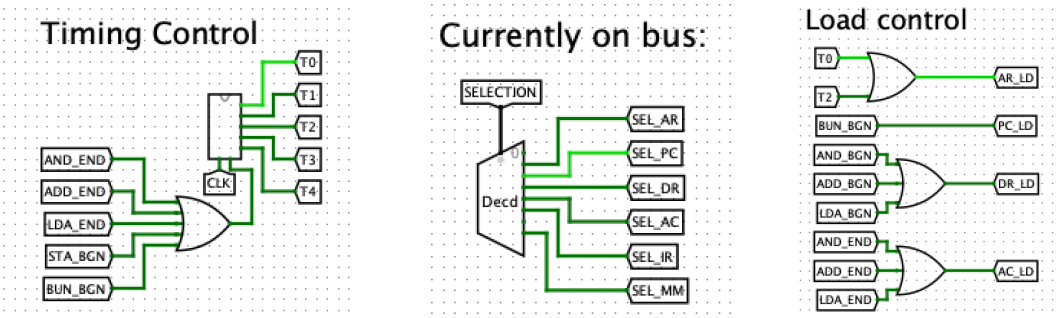
```
v2.0 raw
50 31 70 80 13*0 1
```

위의 프로그램을 구동하여 동작을 분석할 것이다.

5 - 2. Timing Sequence에 따른 동작

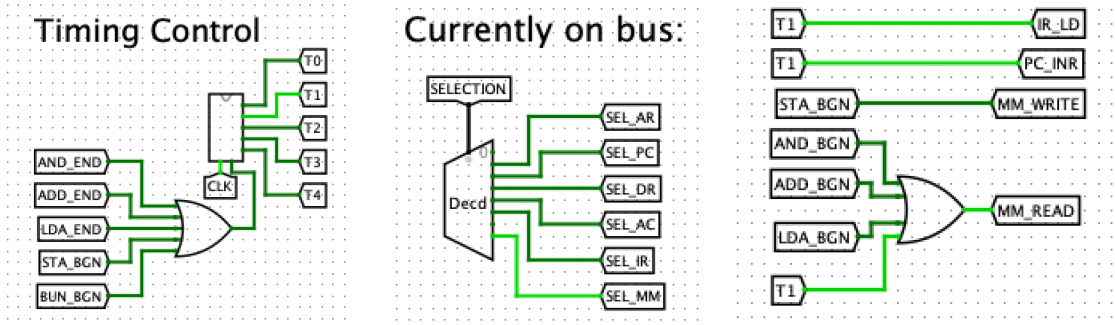
FETCH & DECODE

T0:



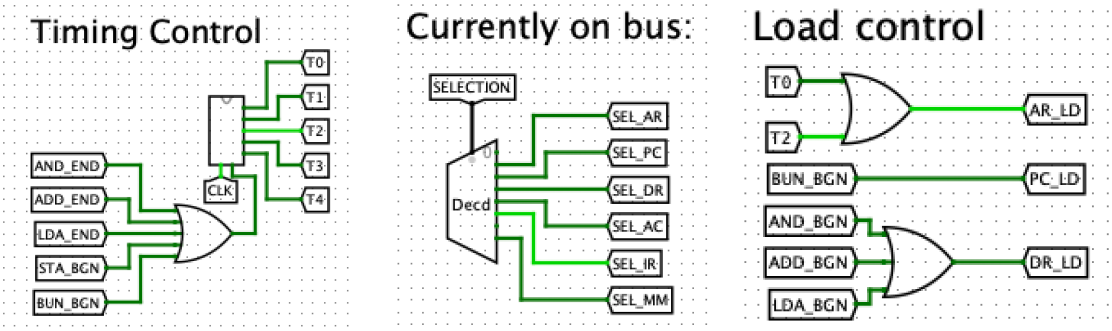
T0일 때에는 Fetch가 실행된다. 버스에는 PC가 올라가 있으며, AR의 로드가 활성화되어있다. PC에서 AR로 값이 전달된다.

T1:



T1에는 메인 메모리가 버스에 올라가 있고 메인 메모리의 읽기 입력이 활성화되어있다. 또한 IR의 로드가 활성화되어있고 PC가 1 증가된다.

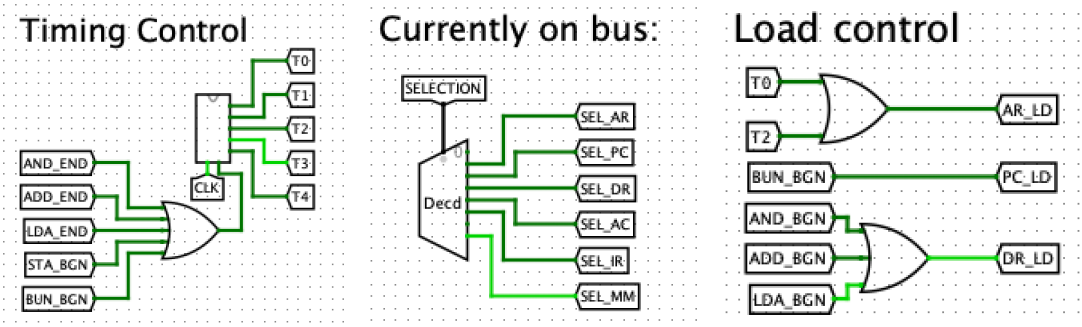
T2:



Fetch와 Decode의 마지막인 T2에는 IR이 버스에 올라가 있고 AR에 로드된다.
operand로 넘어온 주소가 AR에 저장된다.

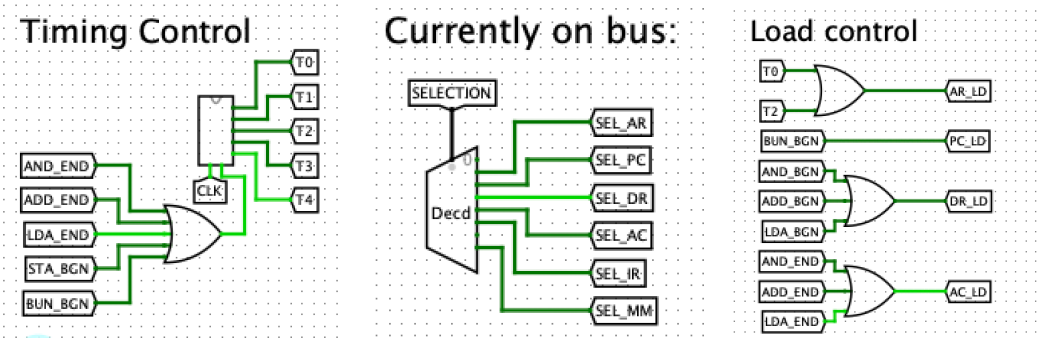
LDA

T3:



메모리에서 0x10번지를 읽은 값 0x00을 DR에 저장한다.

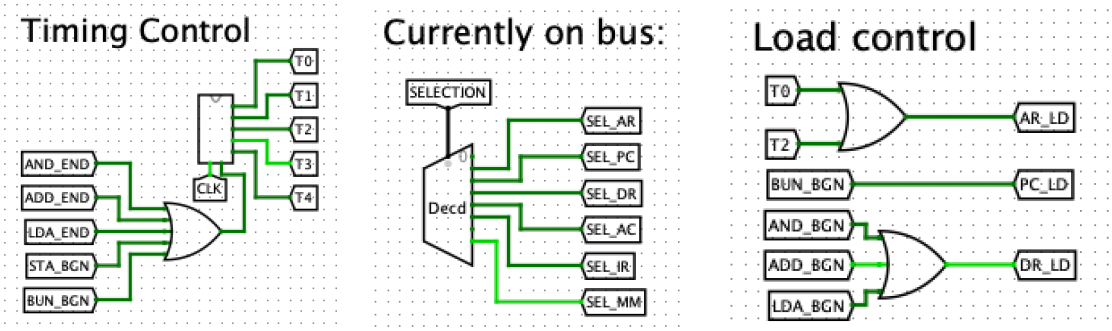
T4:



DR에서 AC로 0x00을 전송한다.

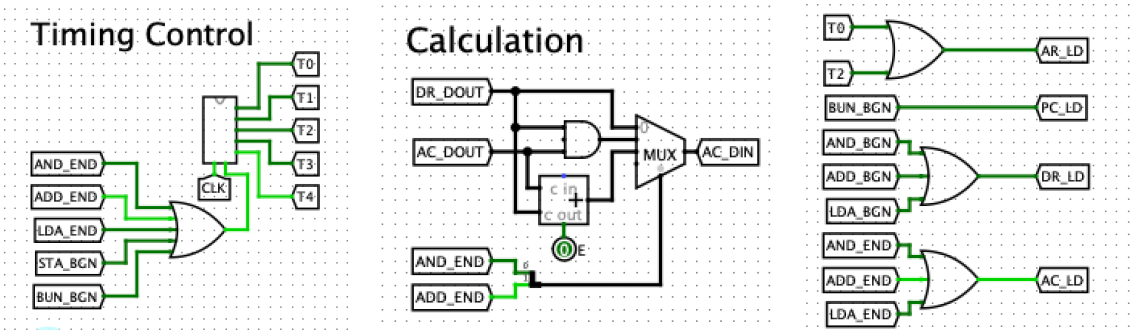
ADD

T3:



0x11번지에서 값 0x01을 읽어 DR에 저장한다.

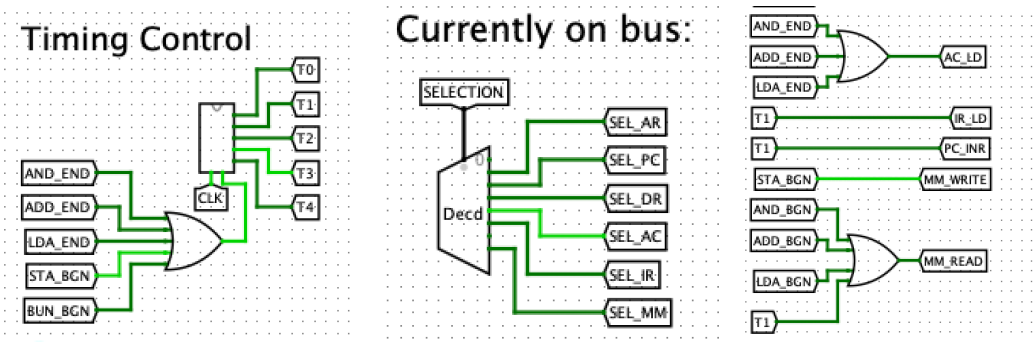
T4:



AC에 이미 들어가 있는 0x00과 DR에 방금 저장된 0x01을 더해서 AC에 0x01을 저장한다.

STA

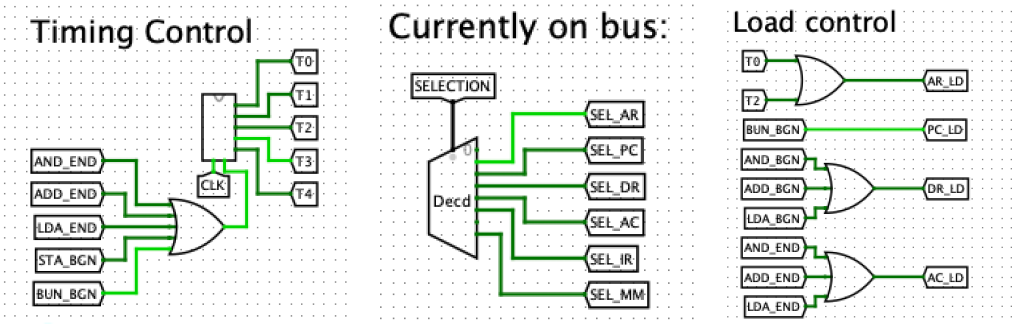
T3:



AC에 있는 값 0x01을 메모리의 0x10번지에 기록한다.

BUN

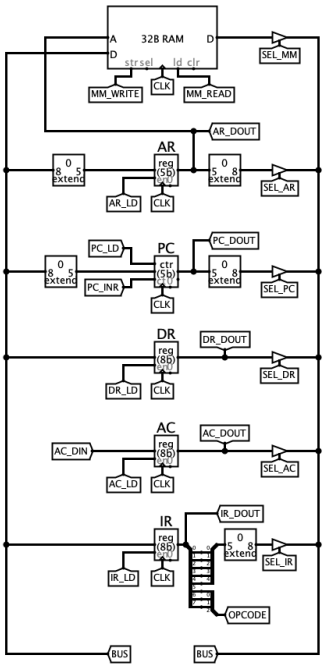
T3:

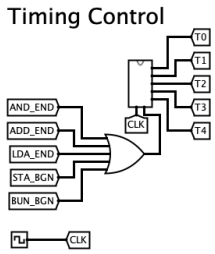


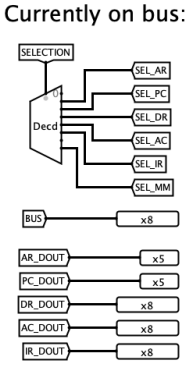
AR에 있는 0x00을 PC로 전송한다. PC가 0x00이 되면 다음 루프가 계속된다.

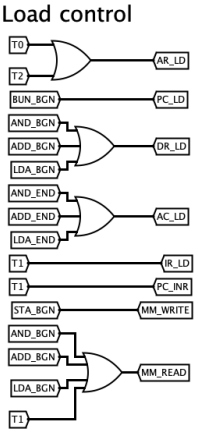
6. 결론

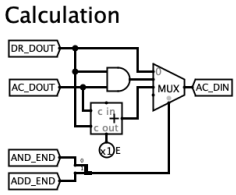
완성된 모습이다.
addLoop 프로그램 테스트를 위하여 작은 디스플레이를 추가했다.

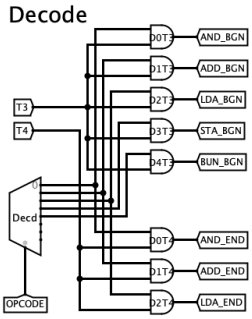


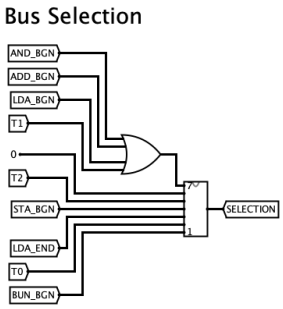


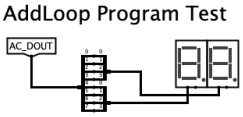


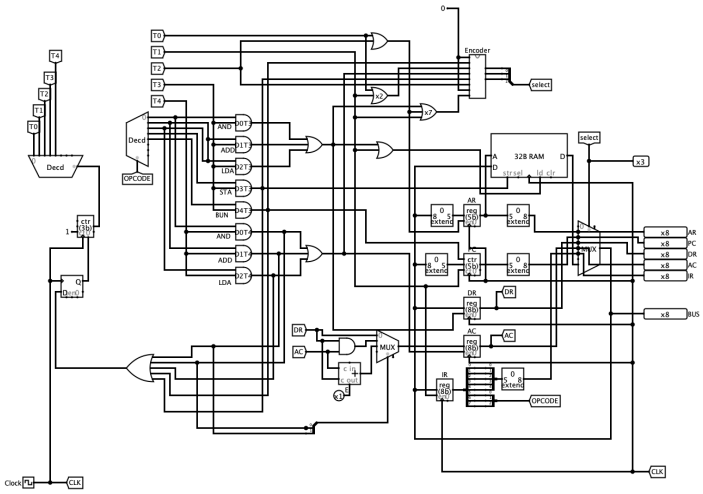












동작은 간단하지만 직접 구현하기 위해 노력을 쏟아부었고 많은 시행착오를 겪었다. 팀에서 처음으로 제작한 버전은 다음과 같이 회로들이 통합되어 있다.

이것이 더 컴퓨터에 가까운 형상을 취하고 있었으나, 하드웨어 디버깅은 쉬운 일이 아니었기에 모듈로 쪼개어 만들 생각을 하였다.

이에 Tunnel을 적극 활용하여 배선의 수를 많이 줄였고, 독립적으로 모듈을 작성하였다. 인코더와 같이 기본으로 제공되지 않는 회로들은 메인 회로에 직접 embed하지 않고 별도의 회로로 작성하여 가져다 사용함으로써 회로의 부피를 줄였다. 또한 Tunnel을 변수처럼 사용해 어느정도 회로를 추상화하여 동작 파악과 이해가 빠르도록 설계하였다.

배운 것을 활용해 두 차례 직접 구현해보았다. 평생 DR이 무엇인지 잊지 못할 것이다.