
논리회로

최종보고서 – Team #1



제출일

2024-11-XX

과목

논리회로 03 분반

담당교수

이충현

전공

소프트웨어학부

학번

20211990

20220790

20241383

20241453

20246091

이름

김동우

이우중

오한빈

장우빈

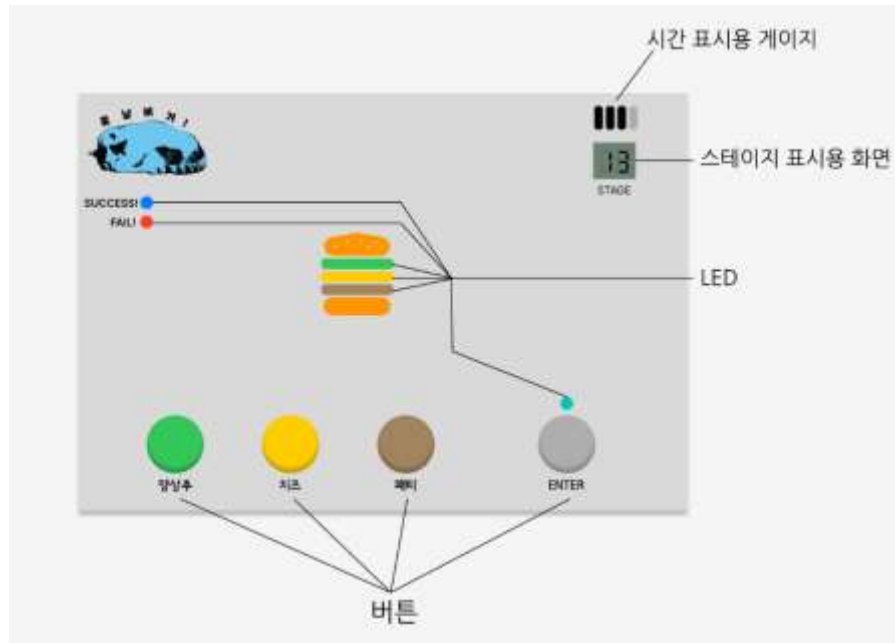
안종민

■ 목차

1. 중량버거 소개	a. 프로그램 실행 흐름 - Player의 관점에서 b. 프로그램 실행 흐름 - Circuit Designer의 관점에서
2. 구현 방식	a. 입/출력 정의 b. 내부 요소 분석 c. 순차적 명령어 분석 d. 회로 명세
3. 최종 결과 및 보완점	a. 프로그램 입력값 설정 b. 입력에 대한 출력 분석 c. 보완점 - 게임 완성에 필요한 부가 요소에 대하여 d. Team Contribution

1. 중낭버거 소개

a. 프로그램 실행 흐름 - Player의 관점에서

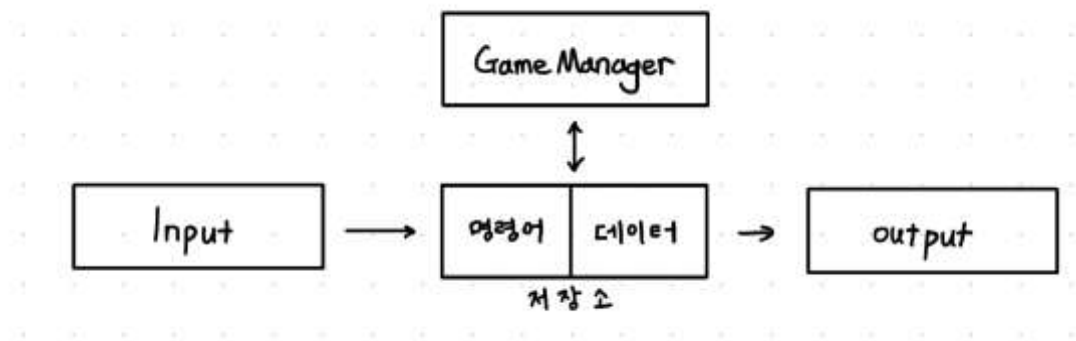


[사진 : 게임 “중낭버거”의 Stage 진행 방식]

중간보고서에 서술했듯이, 게임 “중낭버거”는 사용자(Player)가 지정된 레시피에 맞게 버거를 조리하는 게임이다. 각 재료에 맞는 버튼이 존재하고, 순서에 맞게 버튼을 누르고 ENTER 버튼을 입력하는 간단한 구조를 갖고 있다. 사용자(Player)의 관점에서, 프로그램(게임)이 실행되면 [Press Enter to Start]라는 문구를 보게 되고, Enter를 입력하면 본격적인 게임을 시작한다. 총 15-Stage로 구성되며, 각 Stage에서는 한 명의 손님의 주문을 처리한다는 컨셉이다. Player의 입력이 손님이 제시한 레시피와 일치한다면 다음 Stage로, 일치하지 않는다면 다시 첫 Stage로 돌아가게 된다. 모든 Stage를 클리어하였다면 [All Clear. Congrats!]를 보게 된다.

즉, Player는 4가지 버튼을 적절한 순서로 입력해 게임을 클리어하게 되는 것이 Player 입장에서의 게임의 흐름이다.

b. 프로그램 실행 흐름 – Circuit Designer의 관점에서



[그림 : 게임에 대한 가장 높은 차원의 추상화]

Circuit Designer는 입력을 기반, 올바른 연산을 수행해 적절한 출력을 설계할 의무를 지닌다. 이번 프로젝트에서는 사용자의 입력을 저장된 레지퍼와 비교하고 일치 여부에 따라 다른 작업을 수행하는 등의 연산이 필요하다. 이를 효과적으로 처리하기 위해, '저장소'와 'Game Manager'를 도입하였다. 저장소에는, Player가 입력해야 하는 정답과 각종 변수 data, 게임의 흐름을 처리하는 명령어(Instructions)가 저장되며, Game Manager는 8-bit 명령어를 기반으로 게임의 흐름을 제어한다. 이 두 요소는 입출력을 처리하는 가장 큰 요소로서, 각각은 다시 세부적인 요소들의 집합으로 나눌 수 있다. 이에 대한 명세는 [2-b. 구현 개요 – 전체적인 Structure 중심으로] 에서 다룰 것이다. 또한 이들을 이루는 회로의 자세한 구조를 [2-d. 회로 명세] 에서 살펴보도록 할 것이다.

2. 구현 방식

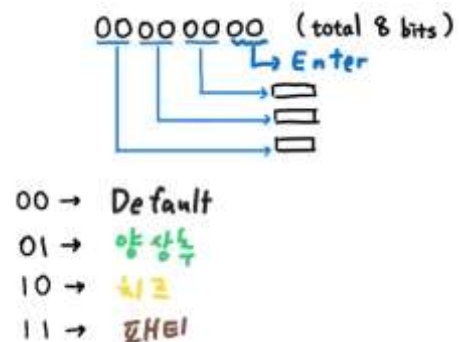
a. 입출력 정의

프로그램이 처리하는 입력(Input)과 출력(Output)을 잘 설정해야 효율적인 방식의 설계를 할 수 있다. Input과 Output의 정의 방식을 살펴보도록 하자.

<<Input 처리>>




[그림1 : Input device의 동작 방식]



[그림2 : 8-bit로 구성된 Input의 구성]

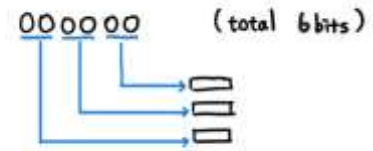
Input은 4개의 버튼에 2 bit씩을 할당하여, 총 8-bit로 이루어져 있다. 즉 3번의 유효한 재료 버튼과 Enter 버튼의 값을 저장하는 것이다. [그림2]처럼, 첫 6-bit는 재료 버튼을, 마지막 2-bit는 Enter의 입력을 의미한다. Input device에 저장된 bit가 8개가 되거나 Enter를 입력 받았을 경우 이를 Input으로 간주하고 처리를 시작한다.

<<Output 처리>>

* Output device
1. 점수
R1 → 
8비트, 0 ~ 15

2. 시간 게이지
R2 →  10ms 단위로 상각!
8비트, 0 ~ 10

3. 레시피 출력 장치

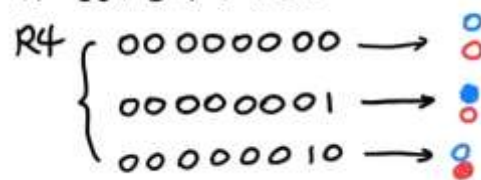


00 → 아무것도 출력하지 않음
01 → 양상속
10 → 치즈
11 → 포테이

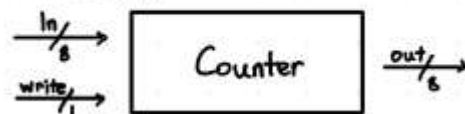
[그림1 : Output#1 - 점수, Output#2 - 시간 게이지]

[그림2 : Output#3 - 레시피 출력]

4. 성공/실패 출력 장치



5. Counter



1. 초기와 값 기준 10까지 카운트
(시간 기준은 1카운트 = 외부 30카운트)
2. 10에서 정지
3. 외부에서 값 읽기, 수정 가능

[그림3 : Output#4 - 최종 Clear 여부]

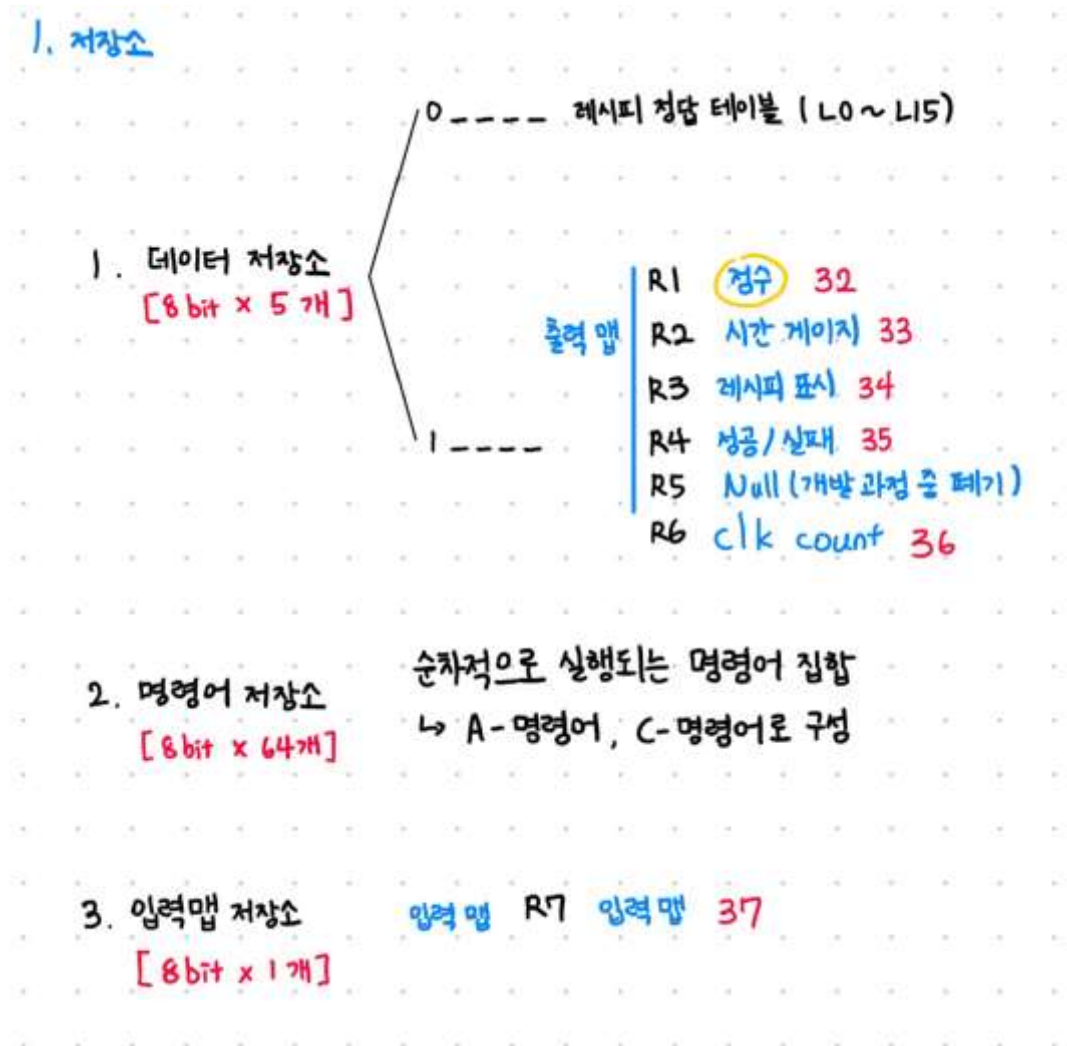
[그림4 : Output#5 - 시간 제한 설정 Counter]

Output의 개수는 Input보다 많다. 하나의 Stage마다 출력되는 Output과 전체 게임 시퀀스가 종료된 후에 출력되는 Output으로 구분해야 한다. Output#1, #2, #3는 Stage에 대한 Output으로, 한 Stage에 진입할 때마다 출력된다. Output#4는 Stage에서 Fail(잘못된 값을 입력 or 시간 초과)이 발생했거나, 전체 게임이 Clear되었다면(15개의 Stage가 전부 Clear된 경우) 출력된다.

Output#1은 Player가 얻은 점수(현재 Stage와 같은 의미)를 디지털 숫자 표기 방식으로 표현하며, 0~15의 범위를 갖는다. 아직 게임이 시작되지 않았다면 '00', Stage 5에서는 '05', 마지막 Stage에서는 '15'를 화면에 출력한다. Output#2는 Stage의 제한 시간 출력을 담당하며, 10칸의 게이지로 구성, 실시간으로 업데이트된다. 따라서 0~10의 범위를 가진다. Output#3는 Player가 입력해야 하는 '정답 레시피'를 출력한다. 2-bit를 3회 조합, 6-bit로 현재 레시피를 출력한다. Output#4는 최종 게임의 Clear 여부를 판단하며 LED 전등의 ON/OFF를 제어하여 Player에게 알린다. 마지막으로 Output#5는 제한 시간을 계산하는 Counter에 해당된다.

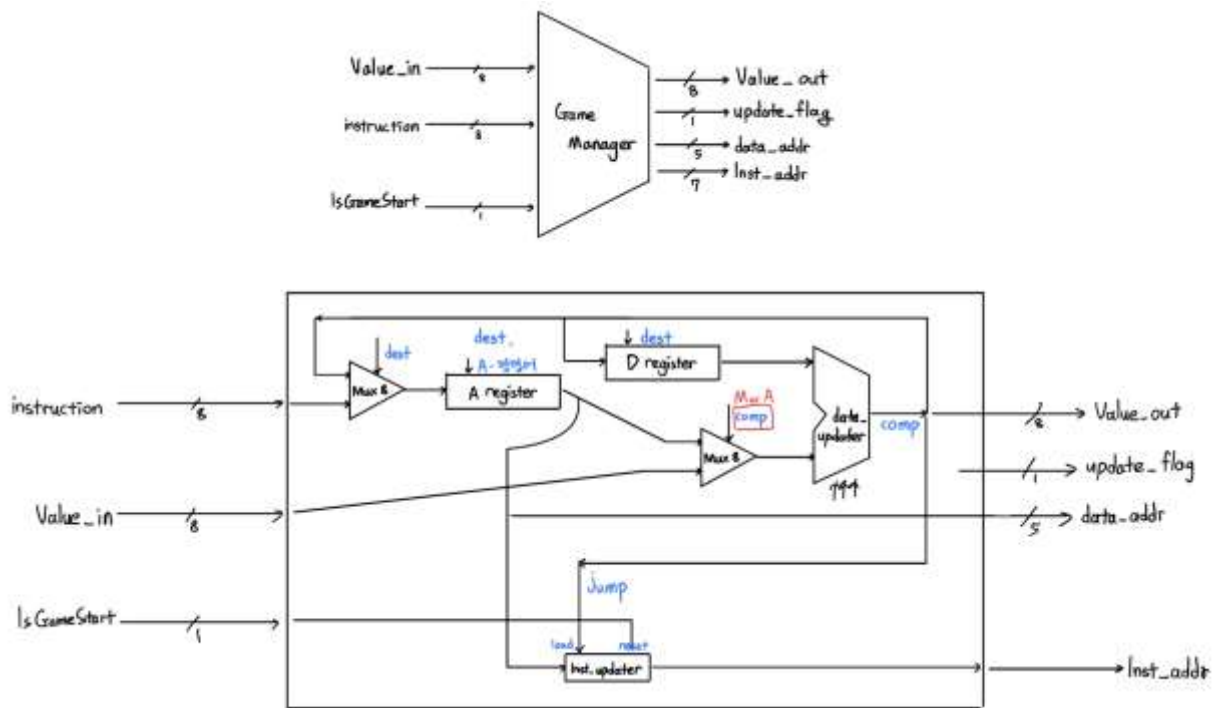
b. 내부 요소 분석

이 프로그램은, 설정된 일련의 명령어(instruction)를 순차적으로 실행하는 과정으로 설명할 수 있다. 64개의 명령어들이 서로 상호작용하며, 저장소와 Game Manager의 State를 변경해 나간다. 먼저 저장소와 Game Manager에 대해서 알아보도록 하겠다.



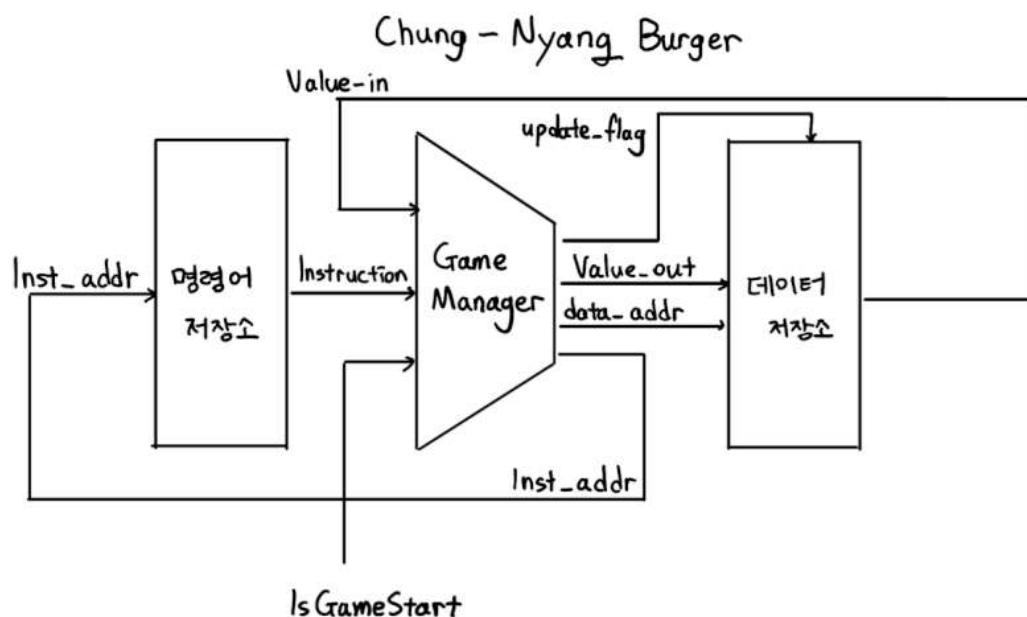
[그림 : 저장소의 구분을 도식화]

저장소는 크게 데이터 저장소, 명령어 저장소, 입력맵 저장소라는 세 부분으로 나눌 수 있다. 데이터 저장소는 레시피의 정답(Player가 입력해야 하는 값)과 Output으로 사용할 출력 맵이 저장되어 있다. R1~R7은 프로그램 구성 시에 참고한 용어이며, 이를 실제 호출할 때 주소처럼 사용하는 값은 빨간 글씨로 적힌 숫자이다. 따라서 후술할 명령어 분석에서 이 값들을 사용하여 State를 변경한다. 명령어 저장소는 State의 값을 변경할 수 있도록 구성된 명령어들을 저장하고 있다. 마지막으로 입력 맵 저장소는 Player가 입력한 Input을 저장하는 8-bit 크기의 저장소이다. 제한 시간을 제어하기 위해 R6를 Count의 용도로 사용하며, 이는 Game Manager에서 설정하게 된다.



[그림 : Game Manager의 I/O, 내부 구조를 도식화]

Game Manager는 명령어를 기반으로 저장소의 값들을 변화시키는 작업을 수행한다. Input으로는 업데이트값 (Value_in), 명령어(instruction), 게임이 진행중인지를 판별하는 Boolean bit(IsGameStart)를 각각 8-bit, 8-bit, 1-bit를 받는다. Output으로는 업데이트 수행 여부(update_flag), 업데이트할 data의 주소와 값 (data_addr, Value_out), 명령어의 주소(Inst_addr)를 1-bit, 8-bit, 8-bit, 7-bit 형식으로 내놓는다. 이때 update_flag가 1이라면 data_addr이 가리키는 값을 Value_out으로 변경하는 작업을 수행한다. 또한 이 Game Manager의 내부 구조는 위 그림처럼 Mux와 Register(처럼 작동하는 간이 저장소), updater를 적절히 조합하여 구성하였다.

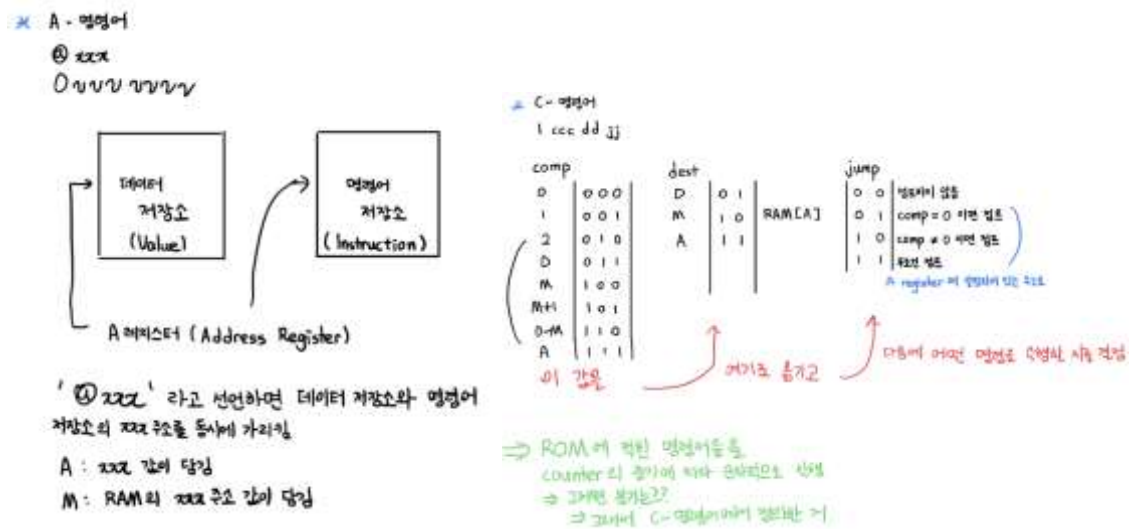


[그림 : 저장소와 Game Manager를 활용한 프로그램 회로도]

결론적으로, 저장소와 Game Manager를 적절히 조합하여 필요한 동작을 하도록 설계하였다. 회로의 구조는 컴퓨터의 그것에서 아이디어를 얻어 유사하게 설계하였으며, 다소 복잡하더라도 회로 내부에 대한 구현, 분업의 효율성 방면에서의 향상을 이루었기에 이 방식을 택하였다.

c. 순차적 명령어 분석

전술한 저장소와 Game Manager를 기반으로 게임의 흐름을 구성하는 순차적 명령어(Instruction)를 살펴보자.



[그림 : 명령어의 두 형태에 대한 명세. A(Address)-명령어와 C(Calculation)-명령어로 구분해 설명]

명령어는 A-명령어와 C-명령어의 두 가지 형태로 구분 가능한데, 각각은 '저장소의 주소에 접근', '계산(Calculation)을 통해 다음 명령어 수행 준비'라는 역할을 갖는다. A-명령어는 8-bit를 통해 실행한다. 이 때 첫 bit는 0으로 고정되고 나머지 7-bit의 조합이 A-레지스터에 저장되고, 이때 가상의 공간인 M-레지스터에는 A-레지스터가 가리키는 주소의 실제 data가 저장된다. C-명령어도 8-bit를 통해 실행되며 2~4번째 bit는 수행할 연산의 종류를, 5~6번째 bit는 이를 저장할 레지스터의 종류를, 7~8번째 bit는 jump의 조건을 나타낸다. A와 M 두 레지스터는 항상 pair로 유지되어, M-레지스터의 변경을 통해 A-레지스터에 저장된 주소가 가리키는 data를 변경할 수 있다.

그림에 나타난 RAM, ROM이라는 명칭은 프로그램 개발 과정에서 이해를 돕기 위해 사용한 것으로, 각각 데이터 저장소, 명령어 저장소를 의미한다. 이런 명령어 종류의 설정이나, 다소 복잡하다고 느껴질 수 있는 설계는 순차적 명령어의 흐름을 간결하고 구현 가능하도록 하기 위한 여러 시도의 결과물임을 밝힌다.

	0 '@R1	00100000 20		8 '@R7	00100101 25
	1 M=0	10001000 88		9 D=M	11000100 C4
	2 '@R2	00100001 21		10 '@8	00001000 8
	3 M=0	10001000 88		11 D==0 -> A	10110001 B1
	4 '@R3	00100010 22	PRESS ANY KEY TO START	12 '@R7	00100101 25
	5 M=0	10001000 88		13 M=0	10001000 88
출력맵 초기화	6 '@R4	00100011 23	읽음표시		
	7 M=0	10001000 88			

[그림 : 순차적 명령어의 흐름(1, 2). 표의 양식 - {역할 / 순서 / 명령어 / 2진수 / 16진수}]

이제 순차적 명령어를 따라가며 프로그램의 흐름을 본격적으로 살펴보도록 하자.

먼저 출력 맵을 초기화하는 작업을 수행해야 한다. R1~R4를 주소로 하는 레지스터를 0으로 초기화하게 된다.

다음 명령어들은 “입력 값이 0이 아님을 조건으로 게임 시작”을 정의한다. 입력맵(R7)에 접근한 다음 해당 값을 D-레지스터(변수로서 사용되는 레지스터)에 저장하고, 이 값이 0이라면(아무런 변화가 발생하지 않았다면) 다시 8 번째 명령어로 돌아간다. 즉 8~11까지의 명령어는 0이 아닌 입력을 탈출 조건으로 하는 loop를 구현한 것이다.

‘읽음 표시’ 단계는 input으로 받은 data를 재사용하지 않도록 ‘읽음 flag’를 지정한다. 입력맵(R7)을 0으로 설정하여 이를 구현한다.

	14 '@R1	00100000 20
스테이지 증가	15 M=M+1	11011000 D8
	16 '@16(CONST)	00010000 10
	17 D=A	11110100 F4
	18 '@R1	00100000 20
	19 D=D-M	11100100 E4
	20 '@48	00110000 30
클리어 확인	21 D==0 -> A	10110001 B1

[그림 : 순차적 명령어 흐름(3)]

‘스테이지 증가’는 점수(Stage Number)에 해당하는 R1을 1만큼 증가시켜 다음 Stage로 이동하였음을 의미한
며, ‘클리어 확인’ 단계에서는 전체 Stage가 클리어되었는지, 즉 점수가 16인지를 검사하는 과정을 거친다. D-레지
스터에 16을 저장하고, M-레지스터에 저장된 현재 Stage 정보를 빼 이 값이 0이라면 성공했을 경우 실행될 동작이
구현된 주소(48)로 이동한다.

	22	`@R1	00100000	20
	23	A=M	11001100	CC
	24	0	00000000	0
	25	D=M	11000100	C4
	26	`@R3	00100010	22
레시피 표시	27	M=D	10111000	B8
	28	`@R6	00100100	24
시간 초기화	29	M=0	10001000	88

[그림 : 순차적 명령어 흐름(4)]

Player에게 레시피 정보를 전달하는 '레시피 표시' 단계이다. 먼저 현재 Stage의 정보를 R1으로부터 읽어와 이를 A-레지스터에 저장하고, 레시피 정보가 저장되어 있는 데이터 저장소에 접근, 현재 Stage의 값을 index로 사용해 현재 Stage에 할당된 레시피 정보를 읽어온다. '시간 초기화' 단계는, 0부터 10까지의 값을 갖는 Time Gauge를 초기화해 다시 10을 가리키도록 한다.

	30	`@10(CONST)	00001010	A
	31	D=A	11110100	F4
	32	`@R6	00100100	24
	33	D=D-M	11100100	E4
	34	`@R2	00100001	21
시간 업데이트	35	M=D	10111000	B8
	36	`@53	00110101	35
시간 초과 확인	37	D=0 -> A	10110001	B1

[그림 : 순차적 명령어 흐름(5)]

'시간 업데이트' 단계에서는 전술한 Time Gauge를 설정한다. Clock Count가 저장된 R6를 10에서 빼서 이를 Time gauge가 저장되는 출력 맵인 R2에 저장하는 방식을 통해 업데이트한다. 이후 '시간 초과 확인'에서 이 D-레지스터의 값이 0이 된다면 게임이 실패하였다는 의미이므로, 실패 시 향해야 하는 주소(@53)로 이동하는 작업을 수행한다.

	38	`@R7	00100101	25
	39	D=M	11000100	C4
	40	`@30	00011101	1D
입력값 확인	41	D==0 -> A	10110001	B1
	42	`@R3	00100010	22
	43	D=D-M	11100100	E4
	44	`@53	00110101	35
입력값 검사	45	D!=0 ->A	10110010	B2
	46	`@12	00001100	C
다음 스테이지 이동	47	goto A	10000011	83

[그림 : 순차적 명령어 흐름(6)]

‘입력값 확인’ 단계에서는 입력 맵(R7)에 담긴 값을 D-레지스터에 저장하고, 아무런 input도 들어오지 않았다면 (if D==0) 30번 명령어로 이동해 시간을 다시 업데이트하는 loop를 구현하였다. 즉, 입력 값이 들어왔는지를 판별하는 procedure이다. ‘입력값 검사’ 단계는 Player가 입력한 값이 정답 레시피(R3에 저장)와 일치하는지 검사하고, 그렇지 않다면 실패 주소(@53)으로 이동하는 작업이 이루어진다. 정답과 입력이 일치하였을 경우, ‘다음 스테이지 이동’을 통해 ‘읽음 표시’ 단계로 이동하여 스테이지를 증가시키는 등 전술한 일련의 흐름이 재실행된다.

	48	`@R4	00100011	23
	49	M=1	10011000	98
	50	`@50	00110010	32
	51	goto A	10000011	83
게임 성공(Clear)	52			0
	53	`@R4	00100011	23
	54	M=2	10101000	A8
	55	`@55	00110111	37
게임 실패(Failed)	56	goto A	10000011	83

[그림 : 순차적 명령어 흐름(7). 57~63번 명령어는 NULL.]

게임의 성공/실패 여부에 따라 R4를 설정하고 무한 루프를 생성하는 단계이다. R4-레지스터는 default : 0, Clear : 1, Failed : 2의 값을 갖게 된다.

지금까지 알아본 순차적 명령어들은 게임의 전체적 Control과 흐름을 제어한다. 이를 구현하기 위해 A-명령어, C-명령어를 통해 비교 연산과 loop를 구현하여 설계하였다.

d. 회로 명세

지금까지 프로그램의 큰 구성품과 전체적인 흐름에 대해 살펴보았다. 2-d. 회로 명세에서는 저장소나 Game Manager을 Quartus에서 구현한 방법과 각 회로의 세부 구조를 명세한다. 먼저, Input Device의 State Table, State Diagram, Equation을 살펴보자.

<<Input Device>>

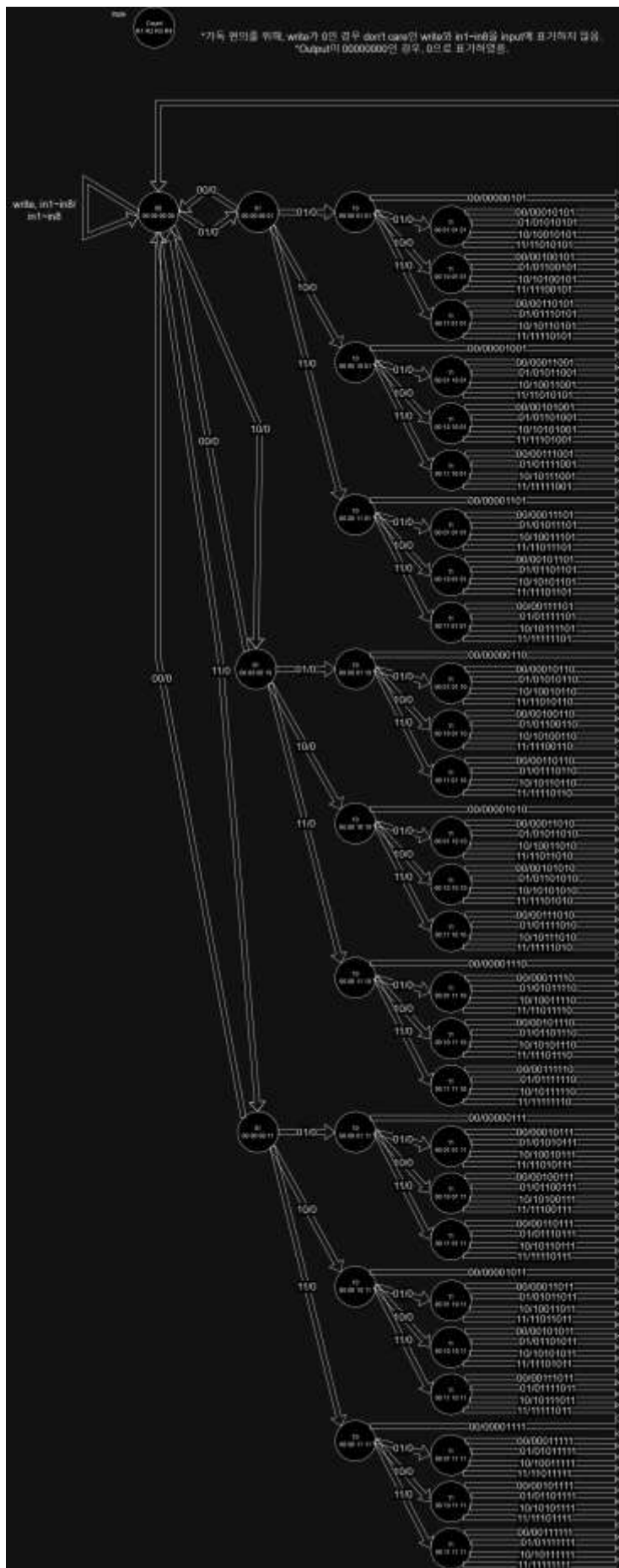
Input							Output							
BT1	BT2	BT3	Enter	write	in1~in8		Out1	Out2	Out3	Out4	Out5	Out6	Out7	Out8
0	0	0	0	0	dd dd dd dd		00		00		00		00	
d	d	d	d	1	in1~in8		in1 in2		in3 in4		in5 in6		in7 in8	
d	d	d	1	0	dd dd dd dd		R1(t)		R2(t)		R3(t)		R4(t)	
d	d	1	0	0	dd dd dd dd		00		00		00		00	
d	1	0	0	0	dd dd dd dd		00		00		00		00	
1	0	0	0	0	dd dd dd dd		00		00		00		00	
d	d	1	0	0	dd dd dd dd		00		00		00		00	
d	1	0	0	0	dd dd dd dd		00		00		00		00	
1	0	0	0	0	dd dd dd dd		00		00		00		00	
d	d	1	0	0	dd dd dd dd		00		00		00		00	
d	1	0	0	0	dd dd dd dd		00		00		00		00	
1	0	0	0	0	dd dd dd dd		00		00		00		00	
d	d	1	0	0	dd dd dd dd		11		R2(t)		R3(t)		R4(t)	
d	1	0	0	0	dd dd dd dd		10		R2(t)		R3(t)		R4(t)	
1	0	0	0	0	dd dd dd dd		01		R2(t)		R3(t)		R4(t)	

[그림1, 2 : Input Device의 State Table – 1. Input & Output]

Present State					Next State				
Count(t)*	R1(t)	R2(t)	R3(t)	R4(t)	Count(t+1)	R1(t+1)	R2(t+1)	R3(t+1)	R4(t+1)
dd	dd	dd	dd	dd	Count(t)	R1(t)	R2(t)	R3(t)	R4(t)
dd	dd	dd	dd	dd	00	00	00	00	00
dd	dd	dd	dd	dd	00	00	00	00	00
00	dd	dd	dd	dd	01	R1(t)	R2(t)	R3(t)	11
00	dd	dd	dd	dd	01	R1(t)	R2(t)	R3(t)	10
00	dd	dd	dd	dd	01	R1(t)	R2(t)	R3(t)	01
01	dd	dd	dd	dd	10	R1(t)	R2(t)	11	R4(t)
01	dd	dd	dd	dd	10	R1(t)	R2(t)	10	R4(t)
01	dd	dd	dd	dd	10	R1(t)	R2(t)	01	R4(t)
10	dd	dd	dd	dd	11	R1(t)	11	R3(t)	R4(t)
10	dd	dd	dd	dd	11	R1(t)	10	R3(t)	R4(t)
10	dd	dd	dd	dd	11	R1(t)	01	R3(t)	R4(t)
11	dd	dd	dd	dd	00	00	00	00	00
11	dd	dd	dd	dd	00	00	00	00	00
11	dd	dd	dd	dd	00	00	00	00	00

* Count(t): Inputdevice 회로에서, Increameter의 출력값으로 나와 Register2로 들어가는 2비트 값.

[그림3,4 : Input Device의 State Table – 2. Input & Output에 따른 현재 State와 다음 State]



[그림5 : Input Device의 State Diagram]

Variables↵

- Inputs: B1, B2, B3, E, W, I1, I2, I3, I4, I5, I6, I7, I8↵
- Outputs: O1, O2, O3, O4, O5, O6, O7, O8 ↵
- State Variables: C1, C2, R1, R2, R3, R4, R5, R6, R7, R8↵

Initialization: Reset to (0,0,0,0,0,0,0,0,0) ↵

↵

Equations↵

$$C1(t+1) = E' \cdot W' \cdot C1 \cdot (B1+B2+B3)' + E' \cdot W' \cdot (B1+B2+B3) \cdot (C1 \oplus C2) \quad \leftarrow$$

$$C2(t+1) = E' \cdot W' \cdot C2 \cdot (B1+B2+B3)' + E' \cdot W' \cdot (B1+B2+B3) \cdot C2' \quad \leftarrow$$

$$R1(t+1) = E' \cdot W' \cdot R1 \cdot (B1+B2+B3)' + E' \cdot W' \cdot (B1+B2+B3) \cdot C1 \cdot C2 \cdot (B2+B3) \quad \leftarrow$$

$$R2(t+1) = E' \cdot W' \cdot R2 \cdot (B1+B2+B3)' + E' \cdot W' \cdot (B1+B2+B3) \cdot C1 \cdot C2 \cdot (B1+B3) \quad \leftarrow$$

$$R3(t+1) = E' \cdot W' \cdot R3 \cdot (B1+B2+B3)' + E' \cdot W' \cdot (B1+B2+B3) \cdot C1 \cdot C2' \cdot (B2+B3) \quad \leftarrow$$

$$R4(t+1) = E' \cdot W' \cdot R4 \cdot (B1+B2+B3)' + E' \cdot W' \cdot (B1+B2+B3) \cdot C1 \cdot C2' \cdot (B1+B3) \quad \leftarrow$$

$$R5(t+1) = E' \cdot W' \cdot R5 \cdot (B1+B2+B3)' + E' \cdot W' \cdot (B1+B2+B3) \cdot C1' \cdot C2 \cdot (B2+B3) \quad \leftarrow$$

$$R6(t+1) = E' \cdot W' \cdot R6 \cdot (B1+B2+B3)' + E' \cdot W' \cdot (B1+B2+B3) \cdot C1' \cdot C2 \cdot (B1+B3) \quad \leftarrow$$

$$R7(t+1) = E' \cdot W' \cdot R7 \cdot (B1+B2+B3)' + E' \cdot W' \cdot (B1+B2+B3) \cdot C1' \cdot C2' \cdot (B2+B3) \quad \leftarrow$$

$$R8(t+1) = E' \cdot W' \cdot R8 \cdot (B1+B2+B3)' + E' \cdot W' \cdot (B1+B2+B3) \cdot C1' \cdot C2' \cdot (B1+B3) \quad \leftarrow$$

$$O1 = W \cdot I1 + (C1C2 + E) \cdot R1 \quad \leftarrow$$

$$O2 = W \cdot I2 + (C1C2 + E) \cdot R2 \quad \leftarrow$$

$$O3 = W \cdot I3 + (C1C2 + E) \cdot R3 \quad \leftarrow$$

$$O4 = W \cdot I4 + (C1C2 + E) \cdot R4 \quad \leftarrow$$

$$O5 = W \cdot I5 + (C1C2 + E) \cdot R5 \quad \leftarrow$$

$$O6 = W \cdot I6 + (C1C2 + E) \cdot R6 \quad \leftarrow$$

$$O7 = W \cdot I7 + (C1C2 + E) \cdot R7 \quad \leftarrow$$

$$O8 = W \cdot I8 + (C1C2 + E) \cdot R8 \quad \leftarrow$$

[그림6 : Input Device의 Equations]

다음으로, 프로그램에서 가장 큰 부분을 차지하는 세 가지 요소 데이터 저장소, 명령어 저장소, Game Manager의 Hardware 명세를 표를 통해 보도록 하자.

2. 데이터저장소(8bit, 64 address)						
load	Address	In	Present State	Next State	Out	
0	Address[6]	d	Read Address[6]	Read Address[6]	데이터저장소[Address[6]]	
1	Address[6]	In[8]	Write to Address[6]	Write complete	In[8]	
3. 명령어저장소(8bit, 64 address)						
load	Address	Present State	Next State = Out			
0	Address[6]	Read Address[6]	Read Address[6]			

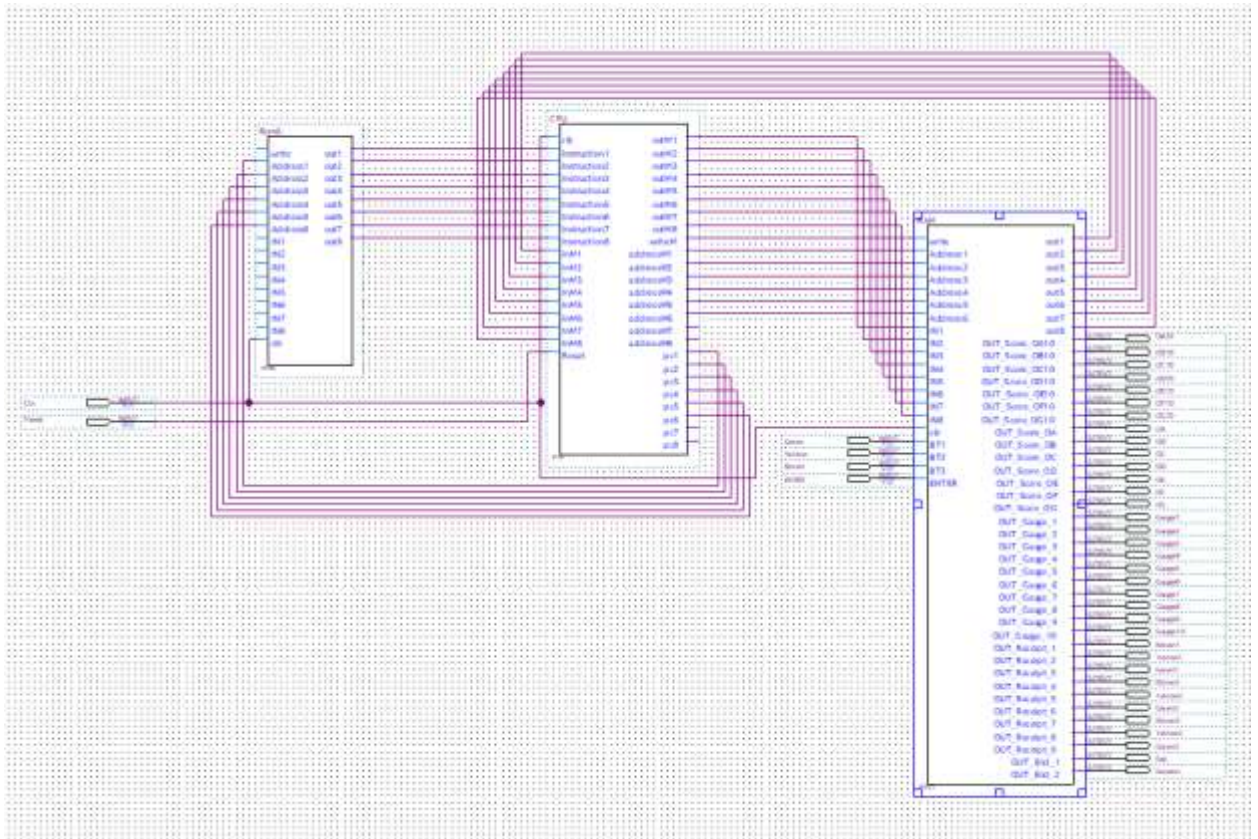
[표 : 데이터 저장소와 명령어 저장소의 Hardware 명세.]

4. Game Manager										
A-Instruction	Value_in	A register's Present State	A register's Next State	D register's Present State	D register's Next State	Value_Out	data_addr	Inst_addr	update_flag	
@pcx	데이터저장소(A register(t))	A register(t)	A register(t + 1) = xxx	d	d	d	A register(t)	PC(t) + 1	d	
C-Instruction										
About comp										
Instruction	Value_in	A register's Present State	A register's Next State	D register's Present State	D register's Next State	Value_Out	data_addr	Inst_addr	update_flag	
1000ddjj	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	0	A register(t)	PC(t) + 1	d	
1001ddjj	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	1	A register(t)	PC(t) + 1	d	
1010ddjj	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	2	A register(t)	PC(t) + 1	d	
1011ddjj	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	D register(t)	A register(t)	PC(t) + 1	d	
1100ddjj	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	Ram(A register(t))	A register(t)	PC(t) + 1	d	
1101ddjj	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	Ram(A register(t)) + 1	A register(t)	PC(t) + 1	d	
1110ddjj	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	D register(t) - Ram(A register(t))	A register(t)	PC(t) + 1	d	
1111ddjj	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	A register(t)	A register(t)	PC(t) + 1	d	
About dest										
Instruction	Value_in	A register's Present State	A register's Next State	D register's Present State	D register's Next State	Value_Out	data_addr	Inst_addr	update_flag	
1ccc00jj	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	Value_Out(t)	A register(t)	PC(t) + 1	0	
1ccc01jj	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	Value_Out(t)	Value_Out(t)	A register(t)	PC(t) + 1	0	
1ccc10jj	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	Value_Out(t)	Value_Out(t)	A register(t)	PC(t) + 1	1	
1ccc11jj	데이터저장소(A register(t))	A register(t)	Value_Out(t)	D register(t)	Value_Out(t)	Value_Out(t)	A register(t)	PC(t) + 1	0	
About jump										
Instruction	Value_in	A register's Present State	A register's Next State	D register's Present State	D register's Next State	Value_Out	data_addr	Inst_addr	update_flag	
1ccdd000	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	d	A register(t)	PC(t) + 1	d	
1ccdd001	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	0	A register(t)	A register(t)	d	
1ccdd001	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	comp != 0	A register(t)	PC(t) + 1	d	
1ccdd100	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	0	A register(t)	PC(t) + 1	d	
1ccdd110	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	comp != 0	A register(t)	A register(t)	d	
1ccdd111	데이터저장소(A register(t))	A register(t)	A register(t)	D register(t)	D register(t)	d	A register(t)	A register(t)	d	

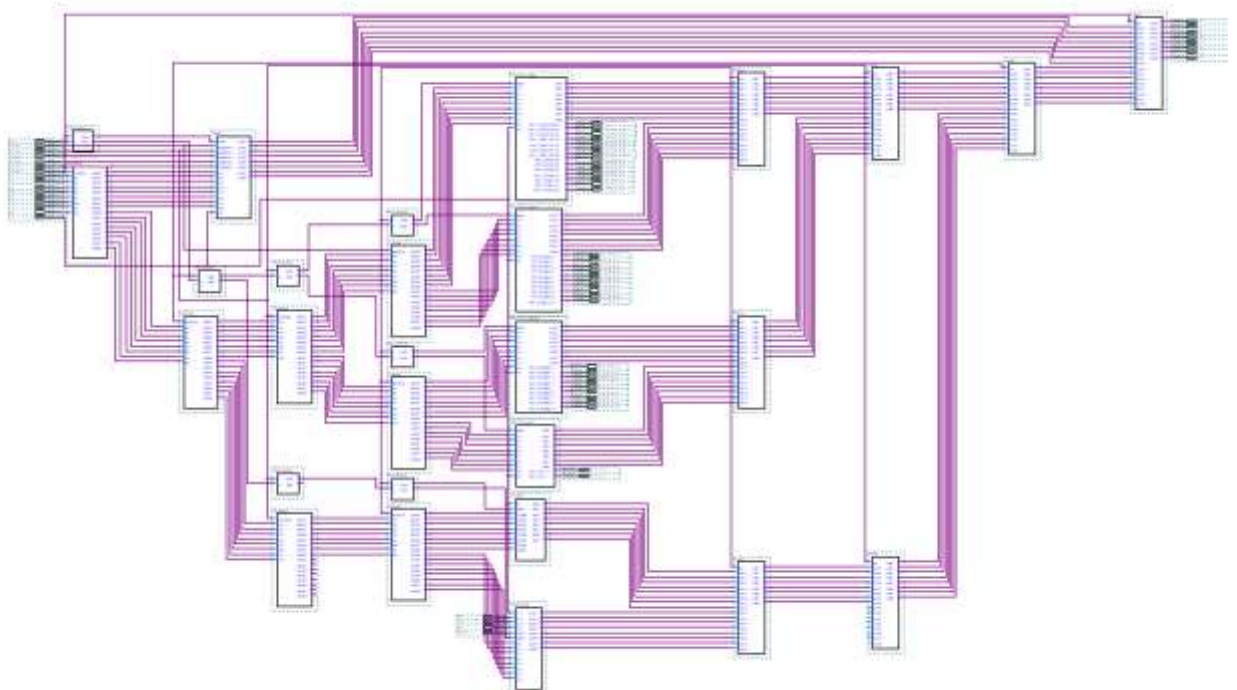
[표 : Game Manager의 Hardware 명세]

3. 최종 결과 및 보완점

a. 프로그램 입력값 설정

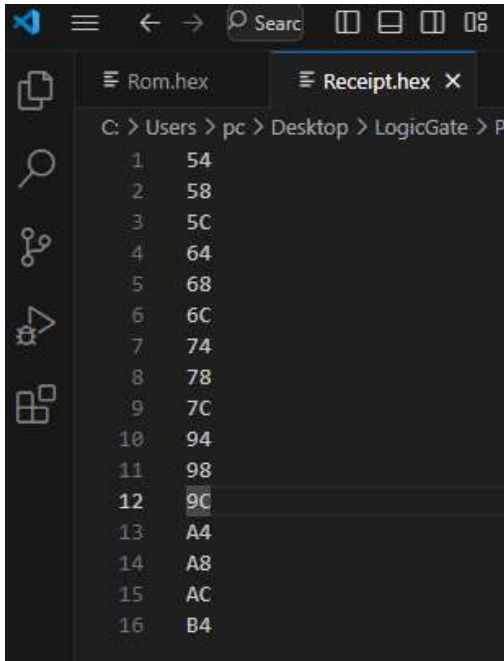


[그림 : '중낭버거' 프로그램의 전체 회로도]

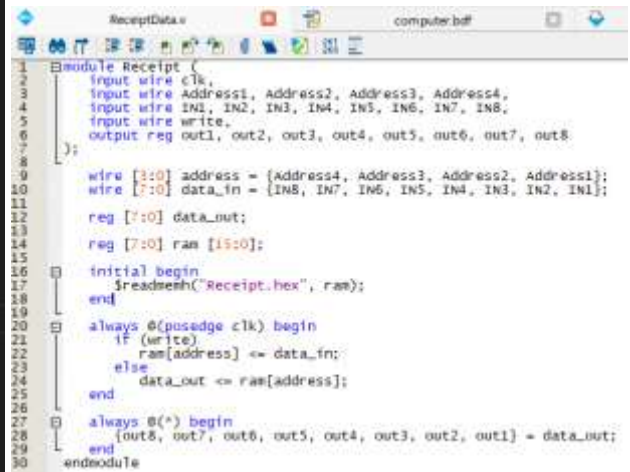


[그림2 : 데이터 저장소와 I/O 처리 수행 회로도]

[그림1]과 [그림2]는 2. 구현방식 에서 소개한 전체적인 구조를 Quartus를 통해 구현한 전체 회로도이다. 이들은 들어온 입력을 기반으로 연산하여 지정된 Output을 출력한다. 이를 위해 저장소에 저장된 값을 사용하는데, 이 저장소에는 프로그램의 실행 이전에 미리 여러 값(정답 레시피나 instruction, etc)이 저장되어 있어야 한다.



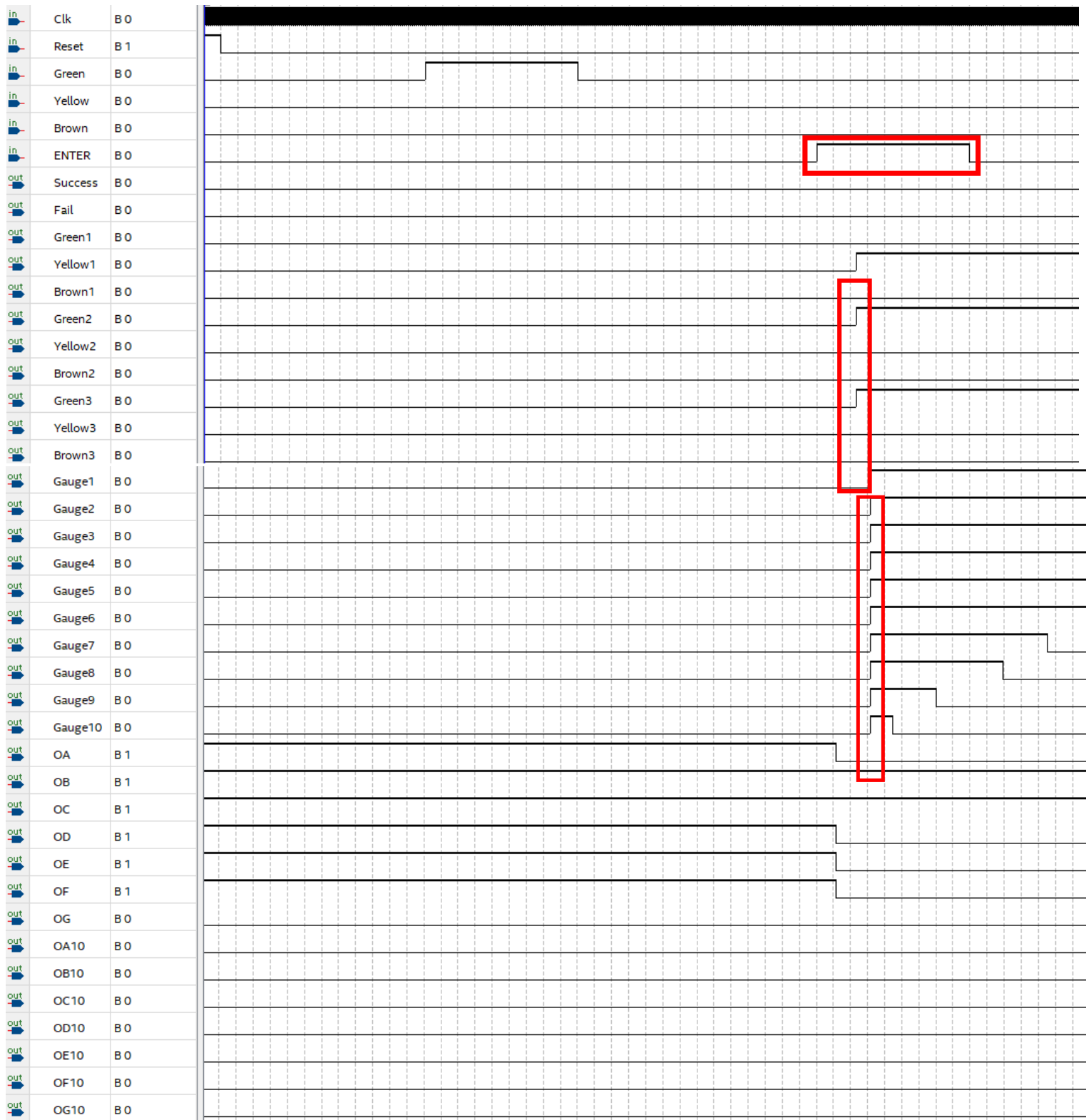
[그림1 : Receipt.hex]



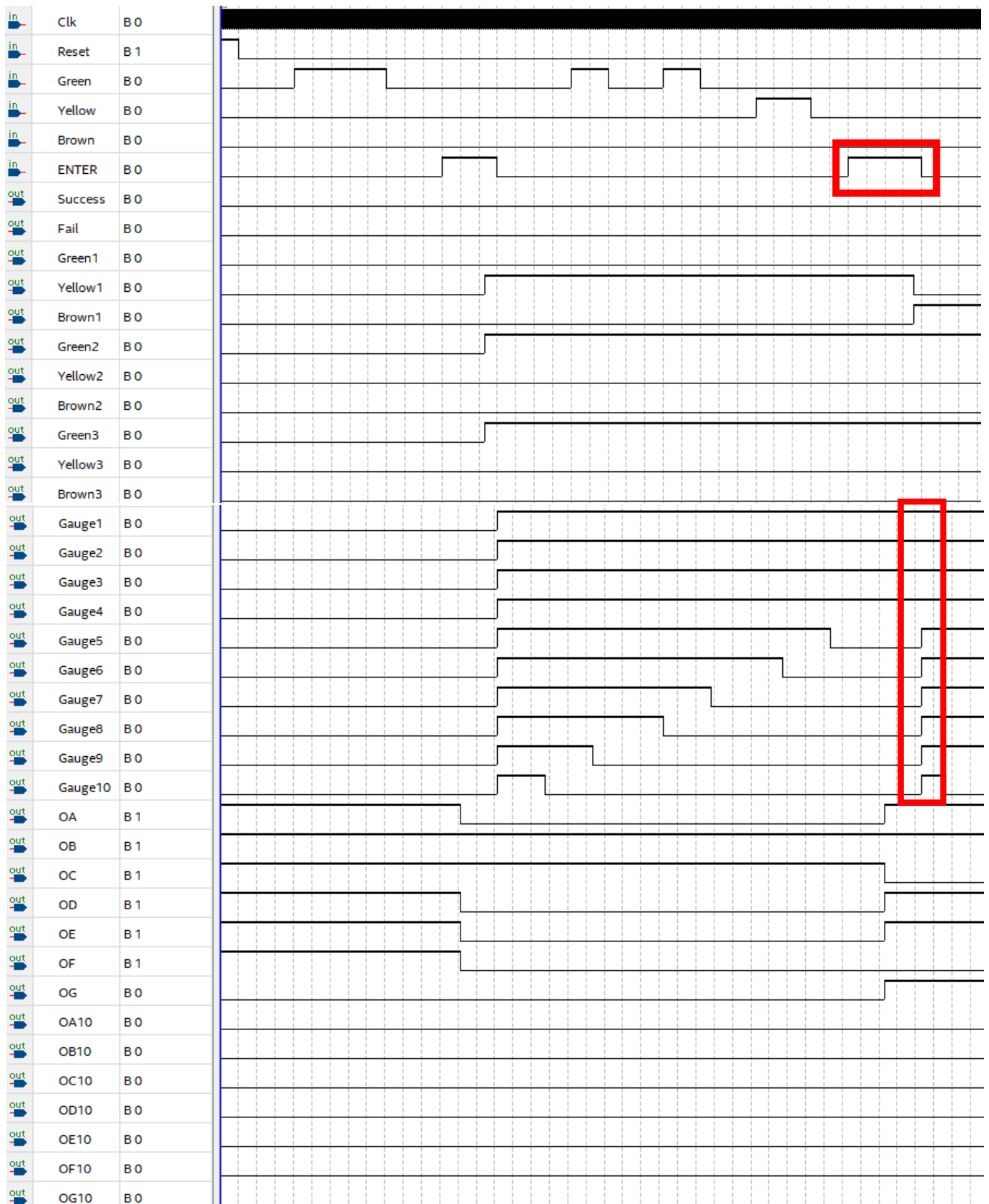
[그림2 : ReceiptData.v]

이를 수행하기 위해, ROM.hex와 Receipt.hex를 사용하였다. hex파일은 Quartus가 읽을 수 있는 16진수가 나열되어 있는 파일이고, 이 파일을 읽기 위한 일종의 코드 파일로서 Instruction.v와 ReceiptData.v를 사용하였다.

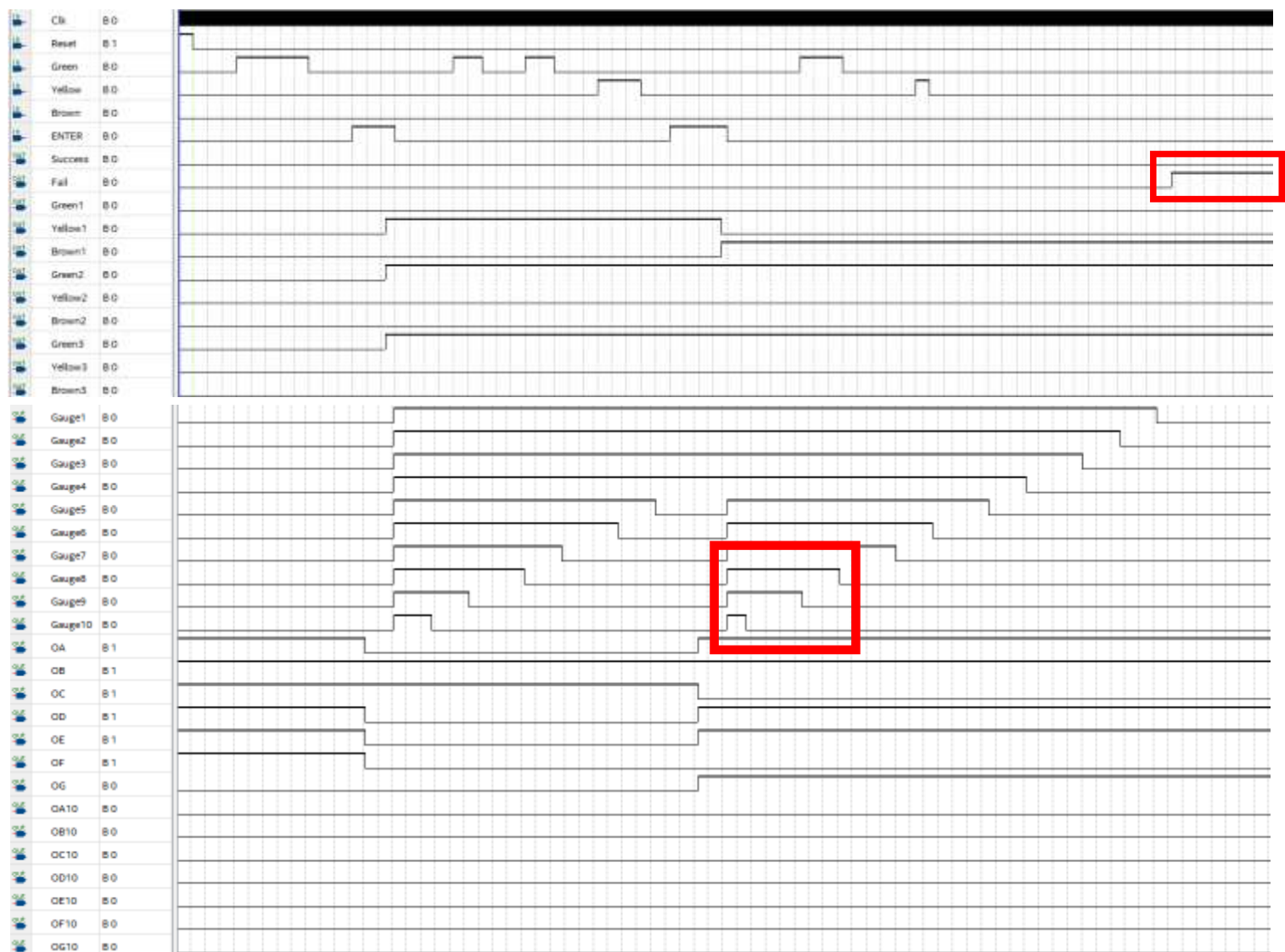
b. 입력에 대한 출력 분석



[그림1 : PRESS ANY KEY TO START 실행 시 결과]



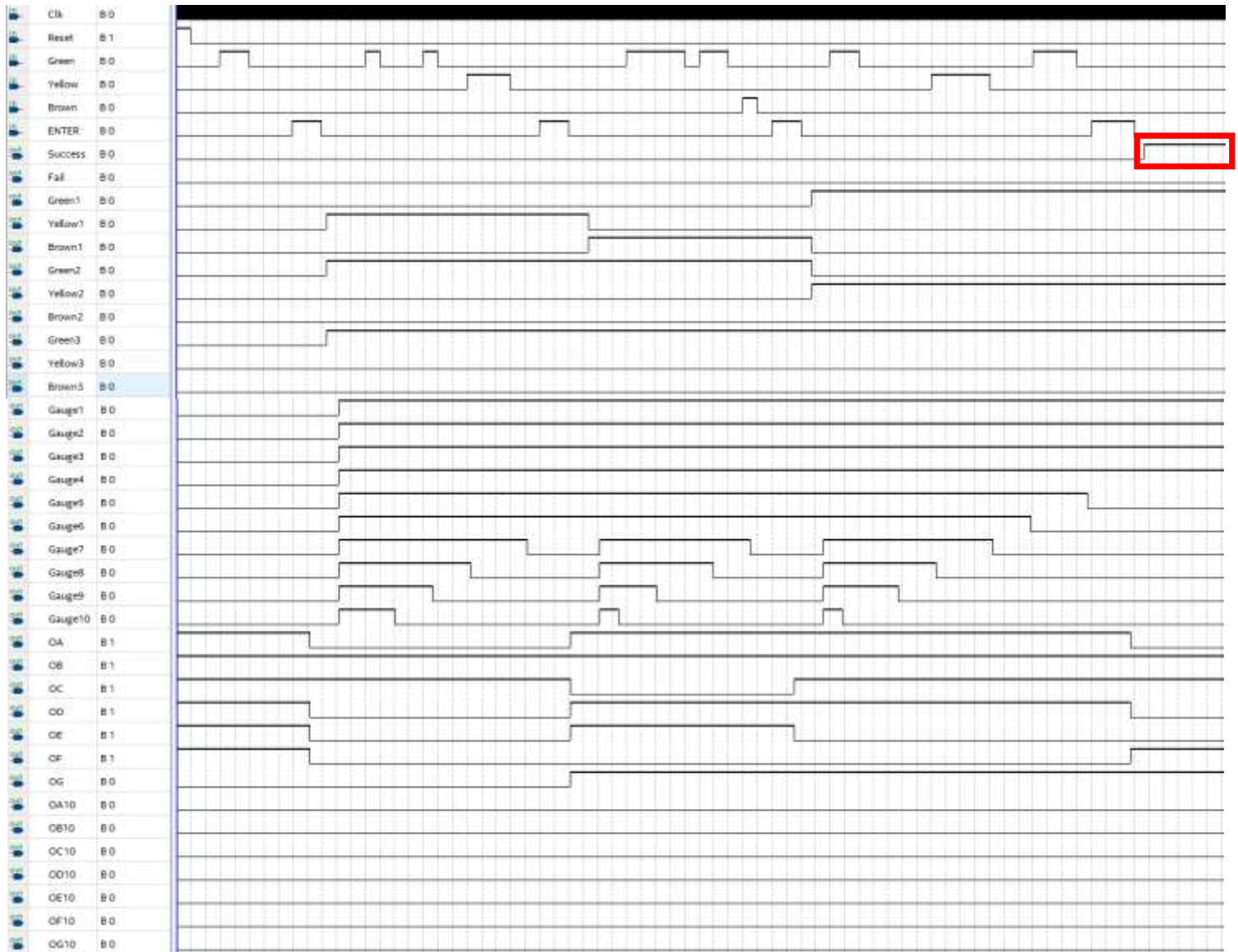
[그림2 : 다음 Stage로 이동]



[그림3 : Failed Case 1 : 시간 초과]



[그림4 : Failed Case 2 : 잘못된 Input]



[그림5 : All Clear. 게임 종료]

기존 설계대로 16-Stage를 구현한다면 시뮬레이션 시간의 한계로 디버깅이 불가능한 문제가 발생하므로, 16번째 명령어를 @4로 설정하여 최종 Clear의 조건은 4개의 Stage Clear로 변경하여 디버깅을 시도하였고, 해당 시뮬레이션의 결과가 [그림5]이다.

c. 보완점 – 게임 완성에 필요한 부가 요소에 대하여

지금까지 설명한 모든 프로그램은, 논리회로로 구성되어 있다. 당연하게도 이들만으로는 온전한 게임의 구현이라고 말하기 어렵다. 따라서 우리의 회로들의 Output이 적절히 활용되어야 완전한 ‘게임’이라고 지칭할 수 있다. 이를 위해 고안된 부가 요소는 다음과 같다.

1. Input을 위한 4가지 버튼

게임을 진행하기 위한 최소한의 버튼을 의미한다. [양상추, 치즈, 패티, Enter]의 4가지 버튼을 상정하였다.

2. Output을 출력하기 위한 Device

간단한 LED 램프를 통해 성공/실패 여부와 Time Gauge 등을 출력하여야 하며, 디지털 숫자 출력을 통해 현재 Stage를 Player가 인식할 수 있도록 해야 한다.

3. Front-End (UI)

게임의 진행을 Player에게 가시적으로 전달하기 위한 UI가 필요하다. 팀 프로젝트에서 진행한 프로그램은 게임 내부의 구성을 위한 것이고, 이는 Quartus의 Wave-Form으로 구성되기 때문에 '게임'의 방식으로 누구에게나 전달할 수 있도록, 알기 쉬운 UI가 필요할 것이다.

결론적으로, 팀 프로젝트에서 설계한 논리회로 프로그램에 추가적인 여러 요소들이 가미되어야 하며 이것들이 성공적으로 연결되었을 때 비로소 온전한 게임을 구성할 수 있을 것이다.

d. Team Contribution

20211990 김동우 - 보고서 작성, 발표 담당, 전체 구조 설계

20220790 이우중 - Program Counter, 데이터 저장소, Mux, DeMux 설계

20241383 오한빈 - 총괄

20241453 장우빈 - InputDevice, ALU 설계

20246091 안종민 - OutputDevice 설계, 발표 ppt 제작