

# 설계 보고서

## 전자공학도의 윤리 강령 (IEEE Code of Ethics)

(출처: <http://www.ieee.org>)

나는 전자공학도로서, 전자공학이 전 세계 인류의 삶에 끼치는 심대한 영향을 인식하여 우리의 직업, 동료와 사회에 대한 나의 의무를 짊어 있어 최고의 윤리적, 전문적 행위를 수행할 것을 다짐하면서, 다음에 동의한다.

1. **공중의 안전, 건강 복리에 대한 책임:** 공중의 안전, 건강, 복리에 부합하는 결정을 할 책임을 질 것이며, 공중 또는 환경을 위협할 수 있는 요인을 신속히 공개한다.
2. **지위 남용 배제:** 실존하거나 예기되는 이해 상충을 가능한 한 피하며, 실제로 이해가 상충할 때에는 이를 이해 관련 당사자에게 알린다. (이해 상충: conflicts of interest, 공적인 지위를 사적 이익에 남용할 가능성)
3. **정직성:** 청구 또는 견적을 함에 있어 입수 가능한 자료에 근거하여 정직하고 현실적으로 한다.
4. **뇌물 수수 금지:** 어떠한 형태의 뇌물도 거절한다.
5. **기술의 영향력 이해:** 기술과 기술의 적절한 응용 및 잠재적 영향에 대한 이해를 높인다.
6. **자기계발 및 책무성:** 기술적 능력을 유지, 증진하며, 훈련 또는 경험을 통하여 자격이 있는 경우이거나 관련 한계를 전부 밝힌 뒤에만 타인을 위한 기술 업무를 수행한다.
7. **엔지니어로서의 자세:** 기술상의 업무에 대한 솔직한 비평을 구하고, 수용하고, 제공하며, 오류를 인정하고 수정하며, 타인의 기여를 적절히 인정한다.
8. **차별 안하기:** 인종, 종교, 성별, 장애, 연령, 출신국 등의 요인에 관계없이 모든 사람을 공정하게 대한다.
9. **도덕성:** 허위 또는 악의적인 행위로 타인, 타인의 재산, 명예, 또는 취업에 해를 끼치지 않는다.
10. **동료애:** 동료와 협력자가 전문분야에서 발전하도록 도우며, 이 윤리 헌장을 준수하도록 지원한다.

위 IEEE 윤리헌장 정신에 입각하여 report를 작성하였음을 서약합니다.

학 부: 전자공학부

제출일: 2022.06.26

과목명: 논리회로실험

교수명: 박성진 교수님

분 반: 수 8.5~11.5

학 번: 201821094 201920941

성 명: 안재혁, 박상현 [8조]

# 1. 설계 목표 및 요구 사항

## [1] 설계 목표

- FPGA kit에서 7-segment 모듈과 keypad 모듈을 이용하여 사칙연산 계산기를 만든다.
- Pin Map을 설정하고 clock pulse를 주었을 때, keypad를 입력하면 구현한 verilog 알고리즘에 맞춰 6자리 숫자 계산기가 동작하도록 한다.
- 계산기를 설계하며 clock signal과 status에 맞춰 동작하는 sequential circuit의 원리와 HDL인 verilog 문법의 이해를 돕는다.

## [2] 요구 사항

### ① 필수 구현

#### (1) 숫자와 연산기호를 번갈아 가며 입력 후 "="를 입력하여 연산 마무리

- 키패드 모듈의 숫자와 연산자(+, -, \*, /, %, ^)를 번갈아 입력하고 마지막에 숫자 입력 후 enter 버튼("=")을 누르면 연산이 마무리되어 최종 계산 결과값을 7-segment에 표시하도록 한다.
- 이때 키패드 모듈로 입력할 수 있는 숫자의 범위는 "-99,999 ~ 999,999" 까지이다. 해당 표현 범위를 넘어가면 error를 출력하도록 한다.
- 키패드마다 코드번호가 존재하며 키패드를 눌렀을 때 코드번호를 통해 해당 키패드가 입력된 것을 계산기가 인식해야 한다.
- 나머지(% , remainder) 연산도 필수 구현 연산자 중 하나이다. 하지만 키패드의 버튼은 총 16개이므로 키패드에 나머지 연산 버튼을 따로 구성할 수 없다는 한계가 있다. 따라서 특정 버튼을 두 번 눌러 나머지 연산을 하도록 구성해야 한다.
- 모든 계산식은 숫자와 연산식이 한 번씩 번갈아 가며 입력되어야 하며, 연산자가 두 번 연속 입력되면 계산기가 동작하지 않도록 해야 한다. 따라서 연산자가 두 번 연속으로 입력되면 7-segment에 error가 출력되도록 해야 한다. 다만 설계에서 구성할 계산기는 음수 계산도 가능하도록 설계할 예정이기 때문에, 연속으로 입력된 연산자가 "-"일 경우는 음수를 입력받는 것으로 인식하여 error가 출력되지 않도록 해야 한다.

#### (2) 연산자 우선순위 무시

- 연산자에는 우선순위라는 것이 존재하며, \*, / 연산이 +, - 연산보다 우선순위가 높으므로 계산식에서 \*, / 연산이 먼저 계산되어야 한다. 다만 설계에서 사용할 계산기는 앞서 언급한 우선순위를 무시해야 한다. 즉 하나의 식에 어느 연산자가 입력되어도 앞에서부터 차례대로 계산되어야 한다는 의미이다.
- 예시로 "1 + 2 \* 3 - 2 = ?"를 계산기에 입력하면 보통 "\*" 연산이 "+, -" 연산보다 우선순위가 높아 먼저 계산된다. 따라서 위 식의 결과는 "5"로 계산되어야 하지만, 설계에서 사용할 계산기에서는 "7"로 계산되어야 한다.

#### (3) 초기화 버튼 구현

- 초기화 버튼 AC(h4000)을 입력하였을 경우, 버튼을 누르기 전까지 계산된 모든 값이 초기화되고 계산기가 처음 피연산자를 입력받을 수 있는 상태로 돌아간다.
- 초기화 버튼을 눌렀을 경우 초기화가 되었다는 표시를 해주기 위해, "set\_no#" 변수 값을 조정하여 7-segment 특정 부분에 불이 들어오도록 설계한다.

#### (4) 추가 기능 구현

##### - 제곱(^, square)

: 기본적으로 구현해야 할 5가지 연산자(+, -, \*, /, %) 외에도 추가적으로 제곱 연산(^)을 넣어 계산이 가능하도록 하였다. 나머지 연산자와 마찬가지로 키패드 버튼이 16개라는 한계점이 있다는 이유로 특정 버튼을 두 번 눌러 제곱 연산이 되도록 해야 한다. 따라서 해당 계산기에선 "\*" 버튼을 두 번 눌러 제곱 연산을 구현할 것이다.

또한 설계에서 사용할 계산기는 enter 버튼을 누르지 않아도 연산자를 누를 때마다 계산이 되도록 설계할 예정이었기 때문에 식 중간에서 제곱 버튼을 눌러도 그 과정까지의 계산값에 제곱 연산이 되도록 구현해야 한다.

## - 음수 입출력 및 계산

: 이번 설계에서 기본적으로 구현되어야 할 사항 중 음수 출력도 포함이었다. 예를 들면  $3 - 5 = -2$  식에서  $-2$ 가 7-segment에 출력되어야 한다는 의미이다. 다만 이는 피연산자를 양수만 입력하였을 경우 나오는 경우이고, 피연산자가 음수일 경우는 포함하지 않았다. 따라서 피연산자에 음수가 입력되었을 때도 계산이 되도록 구성한다.

앞서 언급한 기본 구현 사항에서 연산자를 두 번 입력하였을 경우 error를 출력하도록 구성하지만, 단 두 번째 연산자가 "-"일 때 error를 출력하지 않고 입력받는 피연산자가 음수인 경우로 인식하도록 한다. 피연산자로 음수가 입력되어도 기본적인 연산(+, -, \*, /, %)은 다 이루어지도록 하며 제곱 연산도 마찬가지로 이루어지도록 한다.

단, 음수일 경우 최소로 표현할 수 있는 숫자의 범위는 -99,999이기 때문에 해당 숫자 미만으로 입력되거나 계산된다면 error를 출력하도록 해야 한다. 또한 앞에 "-"부호를 붙일 수 없는 경우가 있는데, 이는 밑 예외처리 부분에서 설명하겠다.

## ② 예외처리 구현

### (1) 0으로 나누었을 때 Error 출력

- 어느 숫자든 0으로 나눈 값은 존재하지 않지 않고 오로지 극한값을 이용하여 나타내야 한다. 다만 계산기는 극한값을 사용할 수 없기 때문에, 존재하지 않는 값으로 인식하고 error를 출력하도록 해야 한다. 또한 verilog 언어에서도 변수값을 0으로 나누다면 don't care(x)로 값이 도출되어 문법상 오류를 범하게 된다. 따라서 나눗셈 연산자 다음으로 피연산자가 0이 입력된다면 error 처리를 해주어야 한다.

### (2) Overflow(999,999 초과) 및 Underflow(-99,999 미만) 시 Error 출력

- 본래 구현 사항은 overflow에 대해서만 예외처리를 하는 것이었지만, 설계할 계산기는 음수 입출력과 계산 기능까지 가능하므로 underflow까지 고려하였다.
- 우리가 설계할 계산기는 출력할 수 있는 segment가 6개인 계산기이므로 overflow는 숫자가 999,999 초과 시 예외처리를 해주어야 하고, underflow는 숫자가 -99,999 미만 시 예외처리를 해주어야 한다.
- 이는 enter 버튼을 눌러 최종 결과값을 출력할 때와 계산식 입력 시 연산자를 입력하여 중간 결과값을 저장할 때로 나누어 예외처리를 고려해야 한다. 극단적인 예시로  $1 + 55,555 * 55,555 \dots$ 를 입력 시, 마지막에 enter 버튼을 누르지 않아도  $* 55,555$ 를 눌렀을 때 error 출력을 해야 한다.

### (3) 예외처리 추가 구현

※ 설계 요구사항에서 명시되어 있지 않았지만, 계산기를 설계하는 도중 몇 가지 필요한 예외처리 부분이 생겨 추가로 구현하는 것으로 계획하였다.

#### - 불필요한 연산자 두 번 연속 입력 시 Error 출력

: 계산기에 식을 입력할 경우, 피연산자와 연산자는 서로 번갈아 가며 입력이 되어야 한다. 즉 연산자는 두 번 연속으로 입력이 되면 예외처리를 해주어야 하는데, 설계할 계산기는 음수 계산까지 포함하기 때문에 예외적인 경우가 있다. 연속으로 입력된 연산자 중 두 번째 연산자가 "-"인 경우는 예외처리를 해주면 안 된다. 즉 (+, -), (-, -), (\*, -), (/ , -) 경우에는 error 출력이 아닌 피연산자가 음수임을 인식하도록 해야 한다.

또한 기본 연산인 나머지(%) 연산과 제곱(^) 연산도 예외처리에서 제외하는 경우인데, 나머지 연산은 나누기(/) 연산자를 두 번 눌러 실행하고 제곱 연산은 곱하기(\*) 연산자를 두 번 눌러 실행하기 때문이다. 즉 (/ , /), (\*, \*)의 경우도 예외처리에서 제외해야 한다.

#### - 계산값이 -0으로 저장될 경우 "-" 부호 제거

: 해당 계산기는 양수와 음수 계산을 모두 포함하므로, 알고리즘에 음수임을 인식하는 변수가 포함되어 있다. 하지만 0은 양수와 음수가 모두 아니기 때문에, 계산 과정 중에 임시 결과 변수가 음수인 상태에서 다음 피연산자와의 계산 결과가 0으로 되는 경우는 0도 음수로 인식하여 숫자 앞에 "-" 부호가 붙을 것이다. 따라서 이를 구분하기 위해 계산 결과가 0인 경우에서도 전 계산 결과가 음수인지 양수인지 구분하는 과정이 필요하다.

## 2. 요구사항 달성 정도

### [1] 필수 구현

#### ① 숫자와 연산기호를 번갈아 가며 입력 후 '='를 입력하여 연산 마무리

- 아래 사진을 참고하여 키패드 모듈의 숫자(-99,999 ~ 999,999) 사이의 숫자를 입력하고 연산자(+, -, \*, /, %, ^)를 입력한다. 그 이후에 다시 숫자(-99,999 ~ 999,999)를 입력하며, 위 방식으로 원하는 수식을 모두 입력한다. '='를 입력하면 연산이 마무리되고 최종 결과가 표시된다.

#### • 장치 키패드 구성

7	8	9	+
4	5	6	-
1	2	3	*
=	0	CLR	/

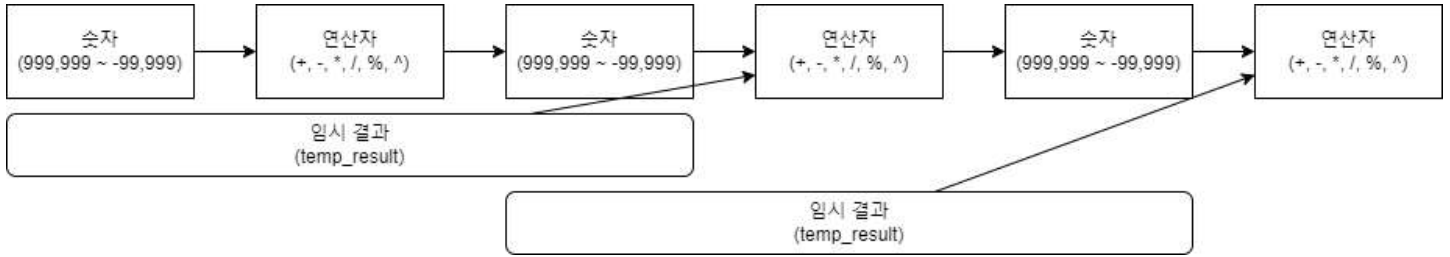
=>

'h0001	'h0002	'h0004	'h0008
'h0010	'h0020	'h0040	'h0080
'h0100	'h0200	'h0400	'h0800
'h1000	'h2000	'h4000	'h8000

- 주어진 example에서는 123, 456, 789 구성으로 키패드가 형성되어 있었다. 하지만 키보드의 keypad, 계산기의 keypad를 모두 관찰해본 결과, 모두 789, 456, 123 구성으로 키패드가 형성되어 있음을 확인하였다. 사용의 편의를 위해 우리는 각 keypad의 pd reg를 변경하여 위 키패드 구성으로 구현하였다. 변경 후 확실하게 사용성이 증가한 부분을 확인할 수 있었다.
- 나머지(% , remainder)를 입력하기 위해서는 /를 두 번 누르면 된다. 예를 들어 7 나누기 3을 하면 몫이 2이고 나머지는 1이 나오게 된다. 본 계산기에서는 "7 / / 3 ="으로 입력할 경우 7 나누기 3의 나머지인 1이 출력된다.
- 제곱(^ , square)의 경우에는 아래 추가 구현 부분에서 설명하였다.
- 본 계산기는 위 요구 사항을 완벽하게 달성하였다. 아무리 긴 수식을 입력하더라도, 순서를 어떻게 입력하더라도, 예기치 못한 오류가 발생하지 않는다. 또한 숫자 입력 시 0부터 입력하거나, 연산자를 두 번 연속하여 입력하는 경우(++ , -- 등), 입력한 숫자나 중간 연산 결과값이 -99,999 ~ 999,999 범위를 벗어나는 경우 모두 err가 표시되며 연산이 중지되도록 하여 정확한 계산 결과가 나오도록 구현하였다.

#### ② 연산자 우선순위 무시

- 연산자 우선순위라는 것이 존재한다. 따라서 수식에서 곱하기와 나머지는 먼저 연산해야하며, 동일한 연산 우선순위를 가진 연산자 사이에서는 앞쪽에 위치한 연산자부터 연산을 해야하는 규칙이 있다. 하지만 실제 계산기 우리가 구현하고자 하는 계산기는 이 연산자의 우선순위를 무시해야한다. 예를 들어 "1 + 2 \* 3 - 2 = ?"을 연산하는 과정에서 일반적으로 연산자 우선순위를 고려한다면 값은 5가 나와야 한다. 하지만 우선순위를 무시하여 앞에서부터 계산한다면 7이라는 값이 나오게 된다.
- 일반적인 계산기(ex. 아이폰 계산기 어플)의 경우 이러한 연산자 우선순위를 고려한다. 따라서 만약에 1 + 2 \* 3 - 2를 입력하게 된다면, 곱하기 혹은 나누기가 입력되는 경우에는 + 혹은 - 가 입력되기 전까지는 \* 와 / 연산만 하고 앞부분에 남아있는 + 나 - 연산은 하지 않는 방식으로 연산자 우선순위를 고려해 준다.
- 하지만 본 요구사항(연산자 우선순위 무시)를 달성하기 위해서는 새로운 방식의 구현이 필요했다. 따라서 우리는 연산자를 구분하지 않고, 매 연산자 입력 시 마다 계산이 되도록 하였으며, 그 계산의 결과는 "임시 결과 ([31:0] temp\_result)"에 매번 저장하도록 구현하였다.
- 본 계산기는 위 요구 사항을 완벽하게 달성하였다. +, -, \*, /, ^, % 등 어떠한 연산자가 입력되더라도 앞에 위치한 연산자부터 차례대로 계산이 되도록 구현하였다. 로직은 다음과 같다.



- 위의 방식으로 연산이 수행된다. "숫자 + 연산자 + 숫자"가 계산되어 임시 결과(temp\_result)에 저장되며, 그 뒤에는 "임시 결과 + 연산자 + 숫자" 으로 계산된다. 따라서 "임시 결과([31:0] temp\_result)"를 하나 추가하는 것만으로 연산자 우선순위를 무시할 수 있게 되었다.

### ③ 초기화 버튼

- 초기화 버튼을 누르면 기존의 모든 변수들이 초기화 되며, 상태 또한 최초의 상태(initial)이 되어야 한다. 따라서 'h4000이 입력 되는 경우 sw\_status는 sw\_start 상태가 되며, 모든 변수들은 0으로 초기화 되도록 하였다. set\_no#에는 모두 18이라는 변수를 주었다. 18이라는 변수는 7-segment의 가운데에만 불이 들어오는(g) 변수로 설정해 두었다.
- 본 계산기는 위 요구사항을 완벽하게 달성하였다. 초기화 되었음을 알려주기 위해 모든 7-segment에 한줄씩 들어오도록 하여 초기화가 되었음을 사용자가 완벽하게 인지할 수 있도록 구현하였다.

### ④ 추가 기능 구현

#### (1) 제곱(^, square)

- \* 를 2회 연속 입력할 경우 제곱으로 사용할 수 있도록 구현하였다. 이를 구현하기 위해서는 가장 핵심적인 부분은 \* 연산자를 연속해서 두 번 누르는 것과 수식 상에서 \* 연산자가 두 번 이상 사용되는 것을 구분 하는 것이었다. 즉, 적절한 순간에 변수를 주고, 적절하게 그 변수를 사용하고, 적절한 순간에 그 변수를 초기화 해 주어야 했다. 우리는 이를 구현하기 위해 square 이라는 변수를 추가하였다.
- square는 'h8000의 입력이 있을 경우 square 변수에 1을 지정한다. 단, 모든 경우에 변수를 주는 것이 아니라 set\_no#의 상태일 때만 square 변수를 1로 주도록 지정하였다. 제곱이라는 연산자는 숫자가 입력된 이후에 입력되는 연산자 이다. 즉, 정상적인 경우라면 set\_no#의 상태일 때 입력되기 때문이다.
- \*가 1회 입력된 후라면 square는 1이고, sw\_status는 sw\_start 상태이다. 따라서 sw\_start 상태일 때 'h8000이 입력되고 square이 1이라면 제곱이 되도록 연산자(input\_operator)에 6이라는 변수를 지정하였다.
- 또한 check\_err라는 변수를 추가하였다. check\_err는 result 버튼을 누른 후 1이 된다. check\_err 덕분에 result가 표시된 이후에도 우리는 제곱 연산을 수행할 수 있다. 최초로 코드를 작성하고 구현할 당시에는 check\_err 라는 변수를 추가하지 않았었다. 그러니 제곱 연산은 연산 도중에 사용하기에 힘들었고, 실제로 제곱 연산은 '=' 버튼을 눌러 result를 보고 그 뒤에 사용하는 경우가 잦음을 고려하여 제곱에 한해서는 '=' 버튼을 누른 이후에도 연산이 되도록 구현하고자 했다. 다행히 ckech\_err 변수를 하나 생성하고, \* 입력시 Err를 출력하는 부분에 관한 코드를 일부만 수정하니까 정상적으로 구현되었다. 덕분에 우리가 구현한 본 계산기는 result를 출력한 이후에도 제곱 연산을 수행할수 있으며, 제곱 연산을 2회 이상 수행하더라도 계속 연산이 수행된다. (ex. 2+2 = 4를 출력한 이후 제곱 버튼을 누르면 16이 출력됨. 이후에 제곱 버튼을 한번더 누를 경우 16의 제곱인 256이 출력됨)
- 곱셈 연산자 \*를 두 번 연속 입력 시 인식할 수 있도록 구성한다. square 변수를 이용하며, \*('h0800) 버튼을 한번 누르면 case->'h0800->default 경우에서 square 값은 1이 된다. 이어서 곱셈 버튼을 다시 한번 더 누르면 상태 파라미터 sw\_status는 sw\_start가 되며, square 값은 이미 1로 저장되어 있으므로 조건문을 통과하여 input\_operator에 6을 저장하도록 한다. (input\_operator의 6은 제곱을 의미한다.)

#### (2) 음수 입출력 및 계산

- 본 환경에서 모든 변수는 unsigned이다. 따라서 모든 값에는 부호에 관련된 data가 존재하지 않는다. 예를 들어, 우리는 연산의 편리성 및 NON-blocking 방식 특성상 매 입력 때 입력된 값을 정수화 시켜주는 temp\_operand라는 변수를 사용한다. 이를 통해 1, 2가 차례로 입력되더라도 우리는 12라는 정수로 바꾸어 연산을 하여 계산이 가

- 능하게 해주었다. 그러나 -, 1, 2를 차례로 입력해서 temp\_operand 에 -12 라는 정수를 입력할 수는 없었다. 따라서 부호에 관련된 정보를 저장하는 별도의 변수를 만들 필요를 느끼게 되었다.
- 우리는 temp\_result의 부호와 관련된 정보를 저장하는 [1:0] sign\_minus 라는 변수를 만들었고, temp\_operand의 부호와 관련된 정보를 저장하는 [1:0] input\_minus 라는 변수를 만들었다.
- 본 기능 구현과 관련된 자세한 설명은 "3. 상세한 설계 내용"에 작성해 두었다.
- 본 계산기는 위 요구사항을 완벽하게 달성하였다. 계산과정 혹은 결과로 음수가 출력되는데 어떠한 예기치 못한 오류도 발생하지 않으며, 음수를 더하거나, 빼거나, 곱하거나, 심지어 나누기까지 가능하다. 음수를 입력하거나 결과가 음수로 나온 경우, \*를 두 번 입력해 제곱을 해도 문제가 없으며, 부호와 관련한 어떠한 예기치 못한 오류도 발생하지 않는다.

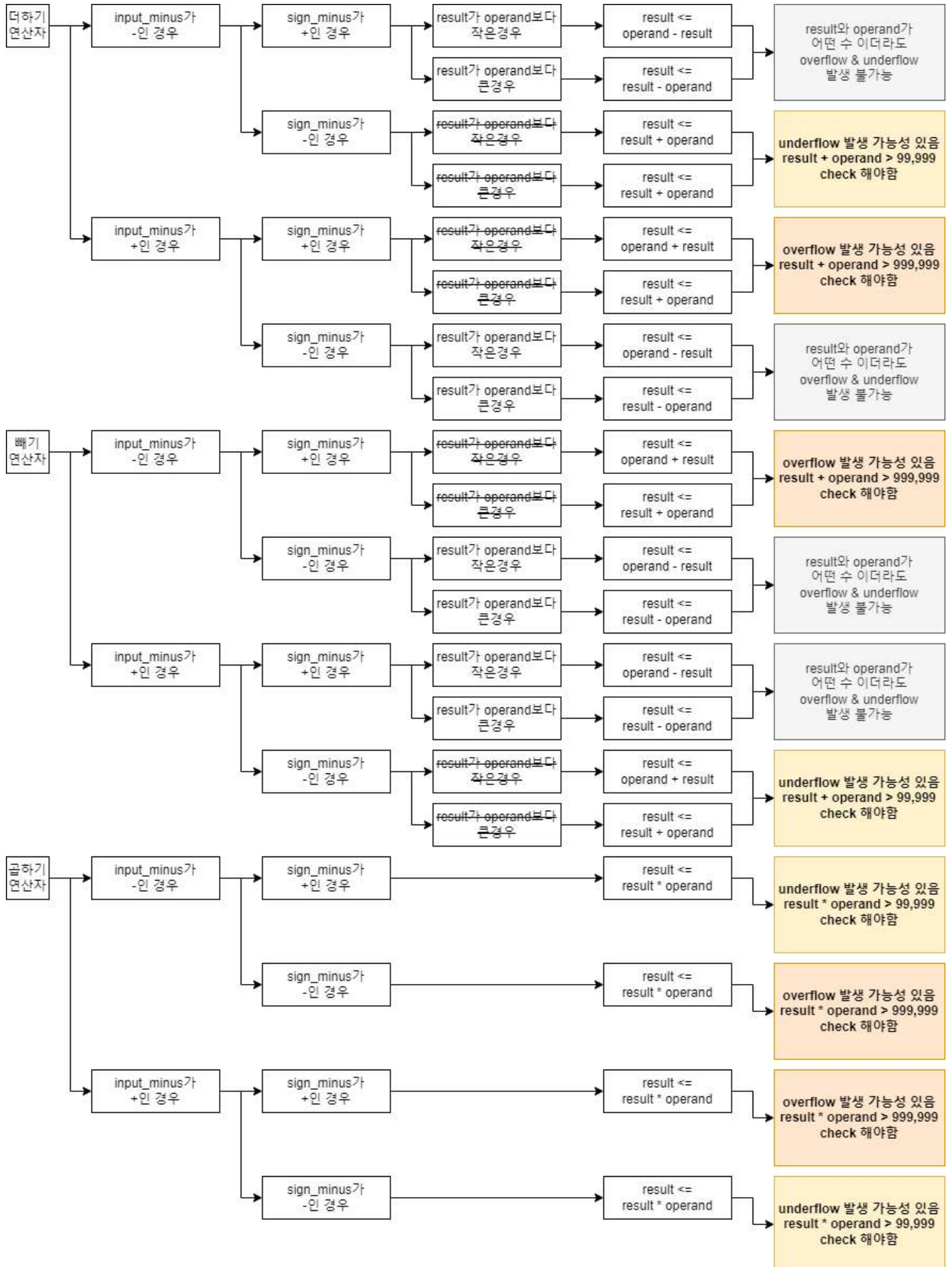
## [2] 예외 처리 구현

### ① 0으로 나누었을 때 Error 출력

- 어떤 식을 0으로 나누면 값이 존재하지 않는다. 따라서 본 계산기에서는 0으로 나누는 행위에 대한 값을 출력해 낼수가 없다. 따라서 우리는 0으로 나누는 경우 err 표시를 해주어야 한다.
- 이를 구현하기 위해 / ('h8000)을 입력 받은 경우 temp\_operand가 0인지 확인하도록 구현하였다. 만일 temp\_operand가 0이라면 이는 수학적으로 값을 도출해 낼 수 없으므로, 0을 출력하게 된다.
- 본 계산기는 위 요구사항을 완벽하게 달성하였다.

### ② Overflow(999,999 초과) & Underflow(-99,999 미만) 시 Error 출력

- 계산의 결과가 999,999보다 크거나 -99,999보다 작은 경우에는 7-segment로 표시할 수 없기에 err를 표시해주어야 한다. 이를 위해 우리는 고려해야 할 것이 많이 있다.
- 먼저 우리는 음수의 입력과 출력을 구현하였기에 고려해야 하는 부분이 많다. 예를 들어보자. temp\_result가 999,990이고 temp\_operand가 100이고 연산자는 + 이다. 과연 우리는 overflow표시를 해야하는가? 또 temp\_result가 99,999이고 temp\_operand가 100이고 연산자는 -이다. 이때도 overflow표시를 해주어야 하는가?
- 우리는 위의 문제를 해결하기 위해 여러 수들을 대입해본 결과, 입력 값의 부호(input\_minus), 임시 결과 값의 부호(sign\_minus), 임시 결과 값과 입력 값의 대소를 고려해야 한다는 결론에 도달하였다. 그래서 위 조건들을 모두 비교하여 case들을 아래와 같이 나누었다. (사진 크기 문제로 인해 다음 장에 첨부함)
- 본 계산기는 위 요구사항을 완벽하게 달성하였다. overflow, underflow는 연산자를 입력하는 과정에서도 검사되며, '='을 눌러 최종 결과를 표시하는 과정에서도 검사되며, 최종 결과에서 ^(\* 2회, 제곱)을 입력하는 과정에서도 검사된다. 예를들어, "999,999 + 1 \* 2 = ?"을 계산하는 수식에서 '\*' 연산자를 입력하는 순간 999,999 + 1 이 overflow임을 확인하면서 err를 표시하게 된다.
- 아래 사진에서 취소선 표시를 해 둔 것은 굳이 대소비교를 하지 않아도 되어 코드 작성 시 해당 조건들을 삭제하여 코드의 길이를 간소화 하고 오류나 실수가 발생할 가능성을 최소화하였다.
- 나눗셈과 나머지, 제곱에 대해서는 Overflow 혹은 Underflow 코드를 작성하지 않았다. 먼저 나눗셈은 절대 overflow나 underflow가 발생할 수 없는 환경이다. 7-segment에 입력되는 값보다 늘 작은 값이 결과로 나오기 때문이다. 나머지와 제곱의 경우에는 마지막에 '='을 누르는 환경에서만 사용을 한다. 따라서 매 연산자들 사이에 위와같이 overflow 나 underflow 코드를 작성하지 않았고, 마지막에 result를 출력하는 과정에서 overflow나 underflow를 확인하도록 했다. 이를 통해 어떠한 숫자나, 연산자가 입력되더라도 모든 환경에서 overflow와 underflow를 구별 내 낼 수 있게 되었다.



### ③ 예외처리 관련 추가 구현

- 예외처리에서는 별도의 추가 구현을 요구하지 않았다. 하지만, 계산기의 설계 완성도를 높이고, 사용성을 증대하기 위해 몇 가지 예외처리(err)를 해야하는 부분이 있어 추가로 구현하였다.

#### (1) 불필요한 연산자 두 번 연속 입력 시 Error 출력

- 연산자를 두 번 이상 연속 입력해야하는 경우가 있다. 그러나 이 경우는 제한적이다. 제곱을 구현하기 위해 \*를 2회 연속해서 누르는 경우, 나눗셈을 구현하기 위해 /을 2회 연속해서 누르는 경우, 덧셈 후 음수 입력을 위해 + -를 누르는 경우, 뺄셈 후 음수 입력을 위해 - -를 누르는 경우, 곱셈 후 음수 입력을 위해 \* -를 누르는 경우, 나눗셈 후 음수 입력을 위해 / -를 누르는 경우에만 사용한다. 따라서 우리는 이 경우를 제외한 다른 경우에 연산자가 입력된다면 err 표시를 해야했다.
- 만일 err 표시를 하지 않게 된다면 연산자 관련 정보를 저장하는 input\_operand의 값이 이상하게 변해버린다. 우리는 non-blocking 방식의 장점을 살려 input\_operand 하나로 이전 연산자 관련 정보와, 입력된 연산자 관련 정보를 동시에 사용할 수가 있다. 이 장점을 살리기 위해서는 위와 같이 연산자를 불필요하게 사용하는 경우는 모두 err 처리를 해주어야 한다.
- 연산자 연속 입력 관련한 Error 출력은 sw\_status를 고려해야한다. sw\_status가 sw\_s1 ~ sw\_s6 일 때 입력되는 연산자는 err 표시를 해서는 안된다. sw\_s1 ~ sw\_s6의 상태는 숫자가 입력되어있는 상태이다. 즉, 이때 입력되는 연산자는 실제 사용자 연산을 하기 원하는 연산자 이기 때문이다.
- 즉, 연산자 연속 입력 관련한 Error 출력은 sw\_status가 sw\_start 상태일 때만 고려해야한다. 즉, 우리는 첫 번째 입력되는 연산자가 무엇인지 고려하는게 아니라 sw\_start 상태일 때 입력되어서는 안되는 연산자에 대해서 Error를 출력하는 방향으로 Error 출력을 구현하였다.
- +는 sw\_start 상태에서 입력되어서는 안된다. 사실 10이라는 숫자는 +10과 동일하다. 그러나 계산기 사용시 +10이라고 입력하지 않는다. 따라서 우리는 이 경우 Error를 표시하도록 구현하였다.
- -는 sw\_start 상태에서 음수를 구현하기 위해 입력이 된다. 따라서 Error를 표시해서는 안되며, 음수 입력으로 인식하고 -를 표시하면서 input\_minus를 1로 바꾸어야 한다. 관련된 자세한 설명은 "3. 상세한 설계 내용"에 작성해 두었다.
- \*는 sw\_start 상태에서 입력될 수 있다. 제곱을 구현하기 위해서는 입력된다. 하지만 그 외에는 입력되어서는 안된다. 따라서 square 변수를 확인해 Error를 표시하도록 구현하였다. square가 1이라는 뜻은 바로 직전 연산자가 \*였다는 의미이며, 이 경우에는 제곱을 연산해야 하므로 Error를 표시하는 것이 아니라 input\_operand를 6으로 해서 제곱연산을 수행해야한다.
- /는 sw\_start 상태에서 입력될 수 있다. 나눗셈의 나머지를 구현하기 위해서 입력된다. 위 제곱과 원리는 동일하다. 제곱에서는 square 변수를 사용했지만, 나눗셈에서는 remainder 변수를 사용한다.
- 위 과정을 통해 본 계산기는 위 요구사항을 완벽하게 달성하였다.

#### (2) -0이 출력되는 경우

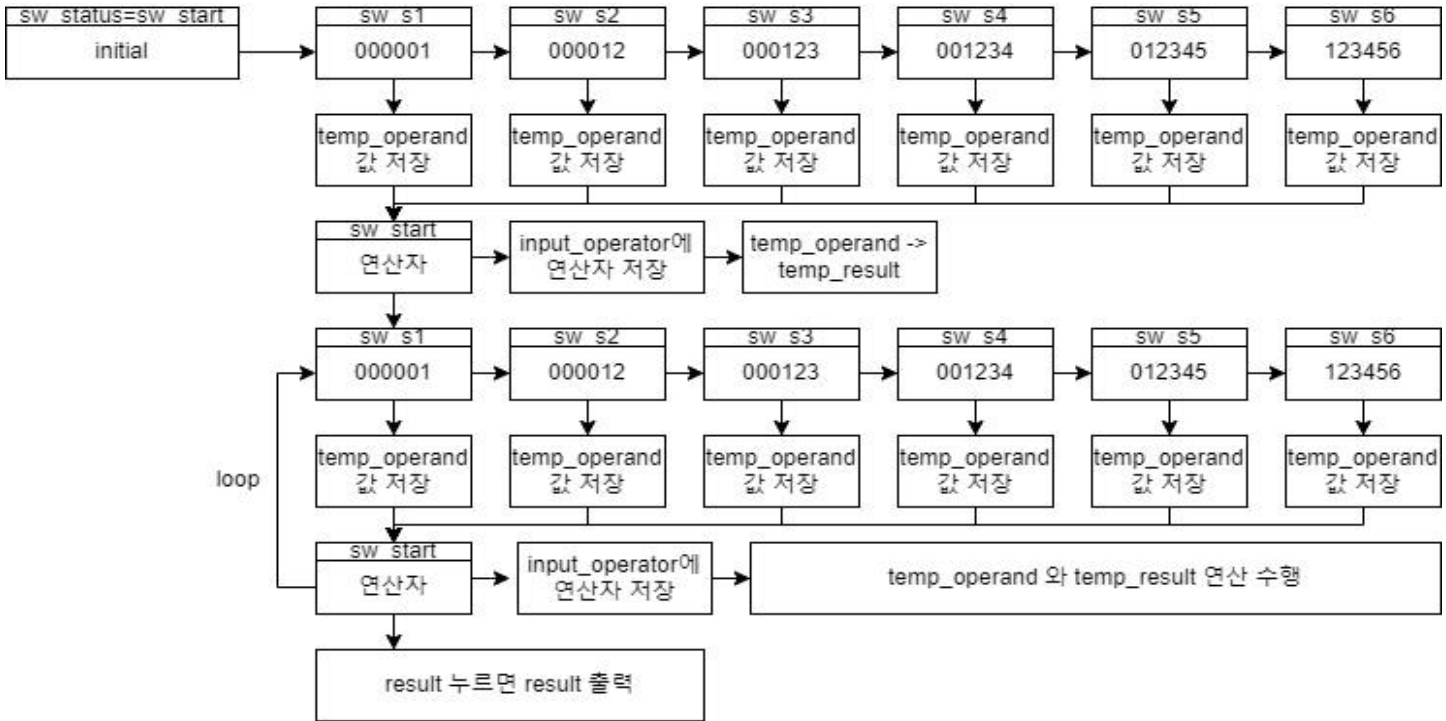
- 앞서 temp\_result와 temp\_operand가 모두 unsigned 변수라 부호를 결정해주는 별도의 변수가 필요하다고 했다. 이를 위해서 우리는 temp\_result의 부호를 결정하는 sign\_minus, temp\_operand의 부호를 결정해주는 input\_minus를 사용하고 있다. 그러나 이렇게 사용하다 보면 다른 경우에는 상관이 없지만 0이 나오면 상당히 애매한 상황이 연출 될 수 있다.
- 예를 들어  $8 - 8$ 의 경우에 input\_minus가 0이라  $8 - 8 = 0$ 의 값을 temp\_result에 저장하고, sign\_minus는 0을 저장하게 된다. 하지만  $8 + - 8$ 의 경우에는 input\_minus가 1이라서,  $8 - 8 = 0$ 이지만  $8 - 8 = -0$ 으로 생각해 sign\_minus는 1을 저장하게 된다. 이는 우리가 temp\_result를 연산하고, overflow & underflow를 확인하고, sign\_minus에 부호를 넣는 과정에서 temp\_result와 temp\_operand의 대소 비교를 하며 큰지, 작은지만 고려했기 때문이다.
- 이 문제를 해결하기 위해 아래와 같이 temp\_result 와 temp\_operand가 동일한 경우를 고려하기로 수정하는 것으로 해당 문제를 완벽하게 해결할 수가 있었다.



```
else begin
    if(temp_result<temp_operand)begin
        temp_result <= temp_operand-temp_result;
        sign_minus <= 0;
        input_minus <= 0;
    end
    if(temp_result==temp_operand)begin
        temp_result <= temp_result - temp_operand;
        sign_minus <= 0;
        input_minus <= 0;
    end
    else begin
        temp_result <= temp_result - temp_operand;
        sign_minus <= 1;
        input_minus <= 0;
    end
end
```

### 3. 상세한 설계 내용 (기능별 상세한 설명)

기본적으로 본 계산기는 아래와 같은 순서로 작동한다.



#### [1] 변수 설명

```

1  module test(clock_50m, pb, fnd_s, fnd_d, temp_result);
2
3  // input output.
4  input clock_50m;
5  input [15:0] pb;
6  output reg [5:0] fnd_s;
7  output reg [7:0] fnd_d;
8
9  // clock.
10 reg [15:0] npb;
11 reg [31:0] init_counter;
12 reg sw_clk;
13 reg fnd_clk;
14 reg [2:0] fnd_cnt;
  
```

<input & output 변수, clock signal 관련 변수>

```

16 // 7-segment
17 reg [4:0] set_no1;
18 reg [4:0] set_no2;
19 reg [4:0] set_no3;
20 reg [4:0] set_no4;
21 reg [4:0] set_no5;
22 reg [4:0] set_no6;
23 reg [6:0] seg_100000;
24 reg [6:0] seg_10000;
25 reg [6:0] seg_1000;
26 reg [6:0] seg_100;
27 reg [6:0] seg_10;
28 reg [6:0] seg_1;
  
```

<7-segment 출력 관련 변수>

#### • input & output 변수 및 clock 관련 변수

clock_50m	50MHz의 clock signal
pb	pushbutton, keypad 버튼 인식
fnd_s	7-segment 출력
fnd_d	7-segment 출력번호
npb	신호 선으로 pb의 not 연산한 값을 대입
init_counter	clock signal을 받아 확인
sw_clk	버튼 입력 여부 확인
fnd_clk	7-segment의 clock signal
fnd_cnt	fnd_clk의 edge 개수 확인

#### • 출력 자릿수 관련 변수

set_no#	7-segment의 가장 왼쪽 자리에서 #번째 자리를 나타내는 변수
seg_100000	7-segment의 10만의 자리 변수 출력
seg_10000	7-segment의 만의 자리 변수 출력
seg_1000	7-segment의 천의 자리 변수 출력
seg_100	7-segment의 백의 자리 변수 출력
seg_10	7-segment의 십의 자리 변수 출력
seg_1	7-segment의 일의 자리 변수 출력

• Keypad 입력 관련 변수

pb_1st	keypad에서 첫 번째로 감지된 값 저장
pb_2nd	keypad에서 두 번째로 감지된 값 저장
sw_toggle	keypad에서 버튼 눌림 여부 확인

• 계산기 상태(status) 관련 변수

sw_status	계산기 상태를 나타내는 변수
sw_idle	초기 상태(계산기 power-off인 상태)
sw_start	피연산자를 입력받지 않은 상태
sw_s#	#개의 숫자를 입력받은 상태

• 계산 과정(calculation)을 위한 변수

temp_result	계산 결과 출력 저장
temp_operand	입력받은 피연산자(숫자) 저장
input_operator	연산자 저장한 변수 <ul style="list-style-type: none"> <li>• 0 : 연산자 저장 없음</li> <li>• 1 : 덧셈 연산자</li> <li>• 2 : 뺄셈 연산자</li> <li>• 3 : 곱셈 연산자</li> <li>• 4 : 나눗셈 연산자</li> <li>• 5 : 나머지 연산자</li> <li>• 6 : 제곱 연산자</li> </ul>
sign_minus	계산 결과(temp_result)의 부호 결정 <ul style="list-style-type: none"> <li>• 0 : 계산 결과가 음수</li> <li>• 1 : 계산 결과가 양수</li> </ul>
check_print	최종 결과를 출력하기 위한 변수 (1이어야 출력 가능)
remainder	나머지 연산을 위한 변수 <ul style="list-style-type: none"> <li>• 0 : 나누기 연산 진행</li> <li>• 1 : 나머지 연산 수행</li> </ul>
square	제곱 연산을 위한 변수 <ul style="list-style-type: none"> <li>• 0 : 곱셈 연산 진행</li> <li>• 1 : 제곱 연산 수행</li> </ul>
input_minus	입력받은 피연산자의 부호 결정 <ul style="list-style-type: none"> <li>• 0 : 피연산자가 음수</li> <li>• 1 : 피연산자가 양수</li> </ul>
pb_minus	뺄셈 연산 결정 여부 <ul style="list-style-type: none"> <li>• 0 : 뺄셈 버튼이 눌리지 않음</li> <li>• 1 : 뺄셈 버튼이 눌림</li> </ul>
check_err	최종 결과값의 제곱 연산을 위한 변수 <ul style="list-style-type: none"> <li>• 0 : 제곱 연산 불가</li> <li>• 1 : 제곱 연산 가능(enter 이후)</li> </ul>
of	Error 발생 여부 <ul style="list-style-type: none"> <li>• 0 : 발생 안 함</li> <li>• 1 : 발생</li> </ul>

```

30 // switch(keypad) control
31 reg [15:0] pb_1st;
32 reg [15:0] pb_2nd;
33 reg sw_toggle;
34
35 // sw_status
36 reg [2:0] sw_status;
37 parameter sw_idle = 0;
38 parameter sw_start = 1;
39 parameter sw_s1 = 2;
40 parameter sw_s2 = 3;
41 parameter sw_s3 = 4;
42 parameter sw_s4 = 5;
43 parameter sw_s5 = 6;
44 parameter sw_s6 = 7;

```

<Keypad 및 status 변수>

```

46 // calculation
47 output reg [31:0] temp_result;
48 reg [31:0] temp_operand;
49 reg [2:0] input_operator;
50 reg [1:0] sign_minus;
51 reg [1:0] check_print;
52 reg [1:0] remainder;
53 reg [1:0] square;
54 reg [1:0] input_minus;
55 reg [1:0] pb_minus;
56 reg [1:0] check_err;
57 reg [1:0] of;

```

<Calculation 변수>

## [2] 초기값

※ 각 사진에 대한 설명을 문단으로 나누어 작성하였습니다.

```

60      initial begin
61          sw_status <= sw_idle;
62          sw_toggle <= 0;
63          npb <= 'h0000;
64          pb_1st <= 'h0000;
65          pb_2nd <= 'h0000;
66          set_no1 <= 18;
67          set_no2 <= 18;
68          set_no3 <= 18;
69          set_no4 <= 18;
70          set_no5 <= 18;
71          set_no6 <= 18;
72          input_operator <= 0;
73          sign_minus <= 0;
74          check_print <= 0;
75          square <= 0;
76          remainder <= 0;
77          input_minus <= 0;
78          pb_minus <= 0;
79          check_err <= 0;
80          of <= 0;
81      end

```

<사진 1>

```

84      always begin
85          npb <= ~pb;
86          sw_clk <= init_counter[20];
87          fnd_clk <= init_counter[16];
88      end

```

<사진 2>

```

91      always @(posedge clock_50m) begin
92          init_counter <= init_counter + 1;
93      end

```

<사진 3>

<사진 1>

- sw\_status <= sw\_idle : 계산기의 초기 상태인 sw\_idle을 상태 변수에 대입한다.
- sw\_toggle : 0은 버튼 눌리지 않은 상태를 나타낸다.
- npb, pb\_1st, pb\_2nd : 어떤 버튼도 눌리지 않았다는 것을 의미하기 위해 'h0000으로 대입한다.
- set\_no# : 초기화라는 의미를 나타내기 위해 7-segment에 "-"가 표시되도록 "18"이라는 값을 넣어준다.

1-9	각 숫자
10	t
11	P
12	A
13	b
14	C
15	d
16	E
17	r
18, 21	-
19	n
20	null

- 나머지 연산자의 초기값 의미는 "상세한 기능설명-변수 설명" 부분에 명시

<사진 2>

- 스위치는 버튼이 눌리면(pb) 0으로 값을 저장한다. 따라서 변수 pb 앞에 not을 붙여 변수 npb에 값을 저장한다. sw\_clk와 fnd\_clk는 32-bit init\_counter의 21번째와 17번째의 값을 가지도록 저장한다.

<사진 3>

- 주파수가 50MHz인 clock signal에서 값이 Low -> High로 바뀌는 positive edge일 때 init\_counter의 값은 1이 증가하여 sw\_clk와 fnd\_clk의 측정이 제대로 이루어지도록 한다.

### [3] 버튼이 눌릴 경우

#### ① 기본 틀

```
95 always @(posedge sw_clk) begin
96     pb_2nd <= pb_1st;
97     pb_1st <= npb;
98
99     if (pb_2nd == 'h0000 && pb_1st != pb_2nd) begin // button pressed
100         sw_toggle <= 1;
101     end
```

<사진 1>

```
103 if(of==0 && check_print==1) begin
104     if (temp_result >=0 && temp_result < 10) begin
105         if (sign_minus==0) begin
106             set_no1 <= 20;
107             set_no2 <= 20;
108             set_no3 <= 20;
109             set_no4 <= 20;
110             set_no5 <= 20;
111             set_no6 <= temp_result;
112         end
113     else begin
114         set_no1 <= 20;
115         set_no2 <= 20;
116         set_no3 <= 20;
117         set_no4 <= 20;
118         set_no5 <= 21; // signed
119         set_no6 <= temp_result;
120     end
121 end
```

<사진 2>

```
142 if (temp_result >=100 && temp_result < 1000) begin
143     if(sign_minus==0) begin
144         set_no1 <= 20;
145         set_no2 <= 20;
146         set_no3 <= 20;
147         set_no4 <= temp_result/100;
148         set_no5 <= (temp_result%100)/10;
149         set_no6 <= (temp_result%10);
150     end
151     else begin
152         set_no1 <= 20;
153         set_no2 <= 20;
154         set_no3 <= 21; // signed
155         set_no4 <= temp_result/100;
156         set_no5 <= (temp_result%100)/10;
157         set_no6 <= (temp_result%10);
158     end
159 end
```

<사진 3>

```
207 if (temp_result >=1000000) begin
208     set_no1 <= 20; //
209     set_no2 <= 20; //
210     set_no3 <= 16; //E
211     set_no4 <= 17; //r
212     set_no5 <= 17; //r
213     set_no6 <= 20; //
214     of <= 1;
215 end
216 check_print <= 0;
217 sw_status <= sw_start;
218 end
```

<사진 4>

```
220 if (sw_toggle == 1 && pb_1st == pb_2nd) begin //unpressed
221     sw_toggle <= 0;
```

<사진 5>

<사진 1>

- "posedge sw\_clk"는 버튼이 눌렸을 때를 의미하며, always 구문을 통해 버튼이 눌렸을 때 구현된 verilog 코드가 동작하도록 구현하였다.
- 버튼이 눌리면 pb\_2nd에 이미 눌렸던 버튼의 정보(pb\_1st)를 전달하고, pb\_1st에는 현재 눌린 버튼의 정보(npb)를 전달한다.
- 또한 처음에 버튼이 눌렸을 경우(pb\_2nd 입력이 0이고 pb\_2nd와 pb\_1st의 값이 다를 경우)에 sw\_toggle에 1을 대입함으로써 버튼이 눌렸음을 인지하도록 한다.

<사진 2>

- 7-segment 출력 여부를 결정하는 check\_print와 error를 판단하는 of의 값이 각각 1, 0일 경우 set\_no#에 값을 넣어줌으로써 출력을 하도록 한다.
- if문으로 temp\_result의 자릿수에 따라 경우를 나누었으며 왼쪽에 있는 사진은 temp\_result가 한 자릿수 일 경우이다. 경우는 한 개의 자릿수부터 여섯 개의 자릿수까지 나누었다.
- 이어서 sign\_minus를 통해 temp\_result가 양수 또는 음수인지 판별하고 set\_no#에 값 "21"을 대입하여 "-" 부호 대입 여부를 결정한다.

<사진 3>

- 3번째 사진은 temp\_result의 자릿수가 세 자리인 경우이다. 양수인 경우(sign\_minus = 0)는 set\_no# 변수 3개에 대하여 temp\_result를 이용한 계산을 통해 각각 값을 대입하는 과정이고, 음수인 경우(sign\_minus = 1)는 set\_no# 변수 4개에 대하여 하나는 "-" 부호 대입과 나머지 3개는 temp\_result를 이용한 계산을 통해 각각 한 자리씩 값을 대입하였다.

<사진 4>

- temp\_result의 자릿수가 6자리를 넘어가면 7-segment에 표시를 할 수 없으므로 overflow로 판단하여 Err를 표기한다. 이어서 출력이 끝나면 check\_print를 초기화하고 상태를 sw\_start로 만들어준다.

<사진 5>

- 만약 sw\_toggle이 1이고, 첫 번째 버튼과 두 번째 버튼의 값이 같으면 버튼이 눌리지 않았다고 판단하여 sw\_toggle은 0으로 초기화한다.



## ② 양수의 피연산자(피연산자 및 연산자 입력)

```
if(input_minus==0)begin
    case (pb_1st)
        'h0100: begin
            pb_minus <= 0;
            case (sw_status)
                sw_idle: begin
                    set_no1 <= 20;
                    set_no2 <= 20;
                    set_no3 <= 20;
                    set_no4 <= 20;
                    set_no5 <= 20;
                    set_no6 <= 20;
                end
                sw_start: begin
                    sw_status <= sw_s1;
                    set_no6 <= 1;
                    temp_operand <= 1;
                end
                sw_s1: begin
                    sw_status <= sw_s2;
                    set_no5 <= set_no6;
                    set_no6 <= 1;
                    temp_operand <= set_no6*10+1;
                end
            endcase
        end
    endcase
end
```

<사진 1>

```
sw_s2: begin
    sw_status <= sw_s3;
    set_no4 <= set_no5;
    set_no5 <= set_no6;
    set_no6 <= 1;
    temp_operand <= set_no5*100+set_no6*10+1;
end
sw_s3: begin
    sw_status <= sw_s4;
    set_no3 <= set_no4;
    set_no4 <= set_no5;
    set_no5 <= set_no6;
    set_no6 <= 1;
    temp_operand <= set_no4*1000+
    set_no5*100+set_no6*10+1;
end
sw_s4: begin
    sw_status <= sw_s5;
    set_no2 <= set_no3;
    set_no3 <= set_no4;
    set_no4 <= set_no5;
    set_no5 <= set_no6;
    set_no6 <= 1;
    temp_operand <= set_no3*10000+set_no4*1000+
    set_no5*100+set_no6*10+1;
end
```

<사진 2>

```
sw_s5: begin
    sw_status <= sw_s6;
    set_no1 <= set_no2;
    set_no2 <= set_no3;
    set_no3 <= set_no4;
    set_no4 <= set_no5;
    set_no5 <= set_no6;
    set_no6 <= 1;
    temp_operand <= set_no2*100000+set_no3*10000+
    set_no4*1000+set_no5*100+set_no6*10+1;
end
sw_s6: begin
    sw_status <= sw_idle;
    set_no1 <= 20;
    set_no2 <= 20;
    set_no3 <= 16;
    set_no4 <= 17;
    set_no5 <= 17;
    set_no6 <= 20;
end
endcase
end
```

<사진 3>

<사진 1>

- 먼저 알고리즘의 큰 틀은 입력받는 피연산자가 양수 또는 음수인지 판별하는 input\_minus 변수를 이용하여 조건문으로 나누었다. 왼쪽 사진의 구문은 양수 (input\_minus = 0)인 경우이다.
- 다음으로 keypad에서 눌린 버튼에 따라 case 구문으로 나누었다. 이때 설정한 변수는 첫 번째로 눌린 버튼을 가리키는 pb\_1st이며 왼쪽 사진은 'h0100(숫자 1)인 경우이다.
- 뺄셈 연산자 버튼이 아니므로 pb\_minus는 0으로 설정하고, 그 안에서도 계산기의 상태를 나타내는 sw\_status에 따라 case를 나누었다.
- sw\_idle부터 sw\_s6까지 나누었으며, sw\_idle은 초기상태이므로 7-segment에 아무 것도 표시하지 않도록 set\_no#을 20으로 설정하였다. 이어서 sw\_start부터 숫자(피연산자)를 입력받으면 sw\_s1로 상태가 바뀌며 set\_no6부터 숫자 1이 저장된다. 또한 피연산자를 저장하는 변수인 temp\_operand에 숫자 1이 저장된다.

<사진 2, 3>

- sw\_s1에서 숫자를 연속으로 이어서 받으면 상태는 점차 sw\_s2부터 sw\_s6까지 올라가며, set\_no#에도 자릿수에 따라 숫자가 저장된다. 여기서 temp\_operand에는 set\_no#가 각각 한 자릿수씩 저장되므로 10의 배수(10, 100, 1000 등)를 곱하여 저장한다.
- sw\_s6는 입력받은 피연산자가 7-segment로 표현할 수 있는 최대 자릿수인 6자리를 넘길 경우이다. 이때는 overflow가 발생한 것이므로 err 표시를 남기도록 set\_no#를 설정한다.(16은 E이고 17은 r을 나타낸다.)
- 'h0100뿐만 아니라 숫자를 가리키는 나머지 keypad 버튼도 옆 사진과 같이 동일한 알고리즘이다. 단, set\_no#와 temp\_operand에 대입되는 숫자만 다르다. 따라서 다른 숫자 키패드의 경우는 설명을 생략하겠다.
- keypad 순서는 일반 계산기 키패드에 맞춰 구현하여 'h0100이 숫자 1로 설정되었고 나머지 keypad도 마찬가지로 구성에 대한 자세한 설명은 "요구사항 달성 정도-필수 구현"에 명시하였다.

※ keypad의 다른 숫자 버튼은 'h0100의 알고리즘과 크게 유사하므로 생략할 것이다.

('h0200, 'h0400, 'h0010, 'h0020, 'h0040, 'h0001, 'h0002, 'h0004, 'h2000)

```

'h0008: begin //+
case (sw_status)
  sw_idle: begin
    set_no1 <= 20;
    set_no2 <= 20;
    set_no3 <= 20;
    set_no4 <= 20;
    set_no5 <= 20;
    set_no6 <= 20;
  end
  sw_start: begin //twice + error
    set_no1 <= 20;
    set_no2 <= 20;
    set_no3 <= 16;
    set_no4 <= 17;
    set_no5 <= 17;
    set_no6 <= 20;
    sw_status <= sw_idle;
  end
  default : begin
    sw_status <= sw_start;
    input_operator <= 1;
    set_no1 <= 20;
    set_no2 <= 20;
    set_no3 <= 20;
    set_no4 <= 20;
    set_no5 <= 20;
    set_no6 <= 20;
  end
end

```

<사진 1>

```

if(input_operator==0) begin
  temp_result <= temp_operand;
  temp_operand <= 0;
end

if(input_operator==1) begin
  if(input_minus==1) begin
    if(sign_minus==0)begin
      if(temp_result<temp_operand)begin
        temp_result <= temp_operand - temp_result;
        sign_minus <= 1;
        input_minus <= 0;
      end
    else begin
      temp_result <= temp_result - temp_operand;
      sign_minus <= 0;
      input_minus <= 0;
    end
  end
  else begin
    if(temp_result+temp_operand>99999) begin //minus overflow
      set_no1 <= 20;
      set_no2 <= 20;
      set_no3 <= 16;
      set_no4 <= 17;
      set_no5 <= 17;
      set_no6 <= 20;
      of <= 1;
      sw_status <= sw_idle;
    end
  end
end

```

<사진 2>

```

  else begin
    temp_result <= temp_result + temp_operand;
    sign_minus <= 1;
    input_minus <= 0;
  end
end
else begin
  if(sign_minus==0)begin
    if(temp_result+temp_operand>999999) begin //plus overflow
      set_no1 <= 20;
      set_no2 <= 20;
      set_no3 <= 16;
      set_no4 <= 17;
      set_no5 <= 17;
      set_no6 <= 20;
      of <= 1;
      sw_status <= sw_idle;
    end
  else begin
    temp_result <= temp_result + temp_operand;
    sign_minus <= 0;
    input_minus <= 0;
  end
end
end

```

<사진 3>

#### <사진 1>

• pb\_1st의 case 구문에서 덧셈 연산자('h0008) 버튼이 눌렸을 경우이다. 피연산자(숫자) 버튼과는 계산기의 상태 sw\_status에 따라 case를 나누어 set\_no#에 값을 대입하는 것은 동일하지만, case가 sw\_idle, sw\_start, sw\_status으로 총 3가지만 존재한다.

• sw\_idle은 초기 상태를 나타내어 set\_no#에 20을 대입하여 아무 표시도 되지 않도록 한다.

• sw\_start는 피연산자를 입력받아야 하는 상황을 나타낸다. 따라서 연산자를 한 번 더 입력할 경우는 예외처리를 해주어야 하므로 Err를 출력하도록 set\_no# 값을 설정한다.

• default는 sw\_s1 ~ sw\_s5 중 하나인 경우를 의미하며, 피연산자를 이미 입력받은 상태이므로 연산자를 입력받을 상태임을 의미한다. 덧셈 연산이므로 input\_operator는 1을 대입하고 7-segment에는 아무 표시도 되지 않도록 20을 대입하였다. 또한 계산기의 상태는 다시 피연산자를 입력받도록 sw\_start로 되돌린다.

#### <사진 2, 3>

• 이후로 input\_operator에 따라 if 조건문으로 경우를 나누었다. 이는 연산자가 입력되는 즉시, 그전까지 입력된 식의 계산값을 temp\_result에 저장하기 위함이다. input\_operator가 0부터 5인 경우까지 나누었다.

• input\_operator = 0

0은 연산자가 입력되지 않은 상황이므로 맨 처음 피연산자 하나만 입력된 상황이라 할 수 있다. 따라서 입력받은 피연산자 temp\_operand를 temp\_result에 넘겨준다.

• input\_operator = 1

덧셈 연산자를 입력받은 경우이고, 조건문으로 입력받았던 피연산자(temp\_operand)의 마이너스 부호에 따라 나누고 이어서 계산된 값(temp\_result)의 부호에 따라 다시 나누었다. 각 if문이 무엇을 의미하는지 예시로 설명하겠다.

① input\_minus = 1 인 경우

[1] sign\_minus = 0 인 경우

if(temp\_result < temp\_operand)

=> ex) 3 + (-5) = -2

else

=> ex) 5 + (-3) = 2

[2] sign\_minus = 1 인 경우

if(temp\_result + temp\_operand > 99999)

=> ex) (-50,000) + (-50,000) = -100,000

\* underflow 예외처리

else

=> ex) (-5) + (-3) = -8

※아래 이어서 설명

```

else begin
    if(temp_result<temp_operand)begin
        temp_result <= temp_operand - temp_result;
        sign_minus <= 0;
        input_minus <= 0;
    end
    else begin
        temp_result <= temp_result - temp_operand;
        sign_minus <= 1;
        input_minus <= 0;
    end
end
end

if(input_operator==2) begin
    if(input_minus==1)begin
        if(sign_minus==0)begin
            if(temp_result+temp_operand>999999)begin //plus overflow
                set_no1 <= 20;
                set_no2 <= 20;
                set_no3 <= 16;
                set_no4 <= 17;
                set_no5 <= 17;
                set_no6 <= 20;
                of <= 1;
                sw_status <= sw_idle;
            end
        end
    end
end

```

<사진 1>

```

else begin
    temp_result <= temp_result + temp_operand;
    sign_minus <= 0;
    input_minus <= 0;
end
end
else begin
    if(temp_result<temp_operand)begin
        temp_result <= temp_operand-temp_result;
        sign_minus <= 0;
        input_minus <= 0;
    end
    else begin
        temp_result <= temp_result - temp_operand;
        sign_minus <= 1;
        input_minus <= 0;
    end
end
end
else begin
    if(sign_minus ==0)begin
        if(temp_result<temp_operand)begin
            temp_result <= temp_operand - temp_result;
            sign_minus <= 1;
            input_minus <= 0;
        end
    end
end

```

<사진 2>

```

else begin
    temp_result <= temp_result - temp_operand;
    sign_minus <= 0;
    input_minus <= 0;
end
end
else begin
    if(temp_result+temp_operand>999999) begin //minus overflow
        set_no1 <= 20;
        set_no2 <= 20;
        set_no3 <= 16;
        set_no4 <= 17;
        set_no5 <= 17;
        set_no6 <= 20;
        of <= 1;
        sw_status <= sw_idle;
    end
    else begin
        temp_result <= temp_result + temp_operand;
        sign_minus <= 1;
        input_minus <= 0;
    end
end
end
end

```

<사진 3>

<사진 1>

② input\_minus = 0 인 경우

[1] sign\_minus = 0 인 경우

if(temp\_result + temp\_operand > 999999)

=> ex) 500,000 + 500,000 = 1,000,000

\* overflow 예외처리

else

=> ex) 3 + 5 = 8

[2] sign\_minus = 1 인 경우

if(temp\_result < temp\_operand)

=> ex) (-3) + 5 = 2

else

=> ex) (-5) + 3 = -2

<사진 1, 2, 3>

• input\_operator = 2

뺄셈 연산자를 입력받은 상황이다. 마찬가지로 예시를 들어 설명하겠다.

① input\_minus = 1 인 경우

[1] sign\_minus = 0 인 경우

if(temp\_result + temp\_operand > 999999)

=> ex) 500,000 - (-500,000) = 1,000,000

\* overflow 예외처리

else

=> ex) 3 - (-5) = 8

[2] sign\_minus = 1 인 경우

if(temp\_result < temp\_operand)

=> ex) (-3) - (-5) = 2

else

=> ex) (-5) - (-3) = -2

② input\_minus = 0 인 경우

[1] sign\_minus = 0 인 경우

if(temp\_result < temp\_operand)

=> ex) 3 - 5 = -2

else

=> ex) 5 - 3 = 2

[2] sign\_minus = 1 인 경우

if(temp\_result + temp\_operand > 999999)

=> ex) (-50,000) - 50,000 = -100,000

\* underflow 예외처리

else

=> ex) (-5) - 3 = -8



```

if(input_operator==3) begin
    if(input_minus==1)begin
        if(sign_minus==0)begin
            if(temp_result*temp_operand>99999) begin //minus overflow
                set_no1 <= 20;
                set_no2 <= 20;
                set_no3 <= 16;
                set_no4 <= 17;
                set_no5 <= 17;
                set_no6 <= 20;
                of <= 1;
                sw_status <= sw_idle;
            end
        else begin
            temp_result <= temp_result * temp_operand;
            sign_minus <= 1;
            input_minus <= 0;
        end
    end
end

```

<사진 1>

```

else begin
    if(temp_result*temp_operand>99999) begin //plus overflow
        set_no1 <= 20;
        set_no2 <= 20;
        set_no3 <= 16;
        set_no4 <= 17;
        set_no5 <= 17;
        set_no6 <= 20;
        of <= 1;
        sw_status <= sw_idle;
    end
    else begin
        temp_result <= temp_result * temp_operand;
        sign_minus <= 0;
        input_minus <= 0;
    end
end
end

```

<사진 2>

```

else begin
    if(sign_minus==0)begin
        if(temp_result*temp_operand>99999) begin //plus overflow
            set_no1 <= 20;
            set_no2 <= 20;
            set_no3 <= 16;
            set_no4 <= 17;
            set_no5 <= 17;
            set_no6 <= 20;
            of <= 1;
            sw_status <= sw_idle;
        end
    else begin
        temp_result <= temp_result * temp_operand;
        sign_minus <= 0;
        input_minus <= 0;
    end
end
end

```

<사진 3>

```

else begin
    if(temp_result*temp_operand>99999) begin //minus overflow
        set_no1 <= 20;
        set_no2 <= 20;
        set_no3 <= 16;
        set_no4 <= 17;
        set_no5 <= 17;
        set_no6 <= 20;
        of <= 1;
        sw_status <= sw_idle;
    end
    else begin
        temp_result <= temp_result * temp_operand;
        sign_minus <= 1;
        input_minus <= 0;
    end
end
end
end

```

<사진 4>

<사진 1, 2, 3, 4>

- input\_operator = 3

곱셈 연산자를 입력받은 상황이다. 마찬가지로 예시를 들어 설명하겠다.

① input\_minus = 1 인 경우

[1] sign\_minus = 0 인 경우

if(temp\_result \* temp\_operand > 99999)  
=> ex) 1,000 \* (-1,000) = -1,000,000  
\* underflow 예외처리

else

=> ex) 3 \* (-5) = -15

[2] sign\_minus = 1 인 경우

if(temp\_result \* temp\_operand > 999999)  
=> ex) (-1,000) \* (-10,000) = 1,000,000  
\* overflow 예외처리

else

=> ex) (-5) \* (-3) = 15

② input\_minus = 0 인 경우

[1] sign\_minus = 0 인 경우

if(temp\_result \* temp\_operand > 999999)  
=> ex) 1,000 \* 10,000 = 1,000,000  
\* overflow 예외처리

else

=> ex) 5 \* 3 = 15

[2] sign\_minus = 1 인 경우

if(temp\_result \* temp\_operand > 99999)  
=> ex) (-1,000) \* 1,000 = -100,000  
\* underflow 예외처리

else

=> ex) (-5) \* 3 = -15

```

if(input_operator==4) begin
    if(temp_operand == 0) begin
        sw_status <= sw_idle;
        set_no1 <= 20;
        set_no2 <= 20;
        set_no3 <= 16;
        set_no4 <= 17;
        set_no5 <= 17;
        set_no6 <= 20;
        of <= 1;
    end
else begin
    if(input_minus==0) begin
        if(sign_minus==0)begin
            temp_result<=temp_result/temp_operand;
            sign_minus<=0;
            input_minus<=0;
        end
        else begin
            if(temp_result%temp_operand==0)begin
                temp_result<=temp_result/temp_operand;
                sign_minus<=1;
                input_minus<=0;
            end
            else begin
                temp_result<=temp_result/temp_operand+1;
                sign_minus<=1;
                input_minus<=0;
            end
        end
    end
end
end

```

<사진 1>

```

else begin
    if(sign_minus==0)begin
        temp_result<=temp_result/temp_operand;
        sign_minus<=1;
        input_minus<=0;
    end
    else begin
        if(temp_result%temp_operand==0)begin
            temp_result<=temp_result/temp_operand;
            sign_minus<=0;
            input_minus<=0;
        end
        else begin
            temp_result<=temp_result/temp_operand+1;
            sign_minus<=0;
            input_minus<=0;
        end
    end
end
end
end

```

<사진 2>

```

if(input_operator==5) begin
    if(sign_minus==0) begin
        temp_result<=temp_result%temp_operand;
        sign_minus<=0;
        input_minus<=0;
    end
    else begin
        if(temp_result%temp_operand==0)begin
            temp_result<=0;
            sign_minus<=0;
            input_minus<=0;
        end
        else begin
            temp_result<=temp_operand-(temp_result%temp_operand);
            sign_minus<=0;
            input_minus<=0;
        end
    end
end
endcase
end

```

<사진 3>

<사진 1, 2>

- input\_operator = 4

나눗셈 연산자를 입력받은 상황이다.

먼저 입력받은 피연산자(temp\_operand)가 0일 경우는 예외처리를 해주어야 한다. 어떤 숫자든 0으로 나눌 수 없기 때문이다.

입력받은 피연산자가 0일 아닐 경우는 마찬가지로 예시를 들어 설명하겠다.

- ① input\_minus = 0 인 경우

- [1] sign\_minus = 0 인 경우

=> ex)  $5 / 3 = 1$

- [2] sign\_minus = 1 인 경우

if(temp\_result % temp\_operand == 0)

=> ex)  $(-6) / 3 = -2$

else

=> ex)  $(-5) / 3 = -2$

- ② input\_minus = 1 인 경우

- [1] sign\_minus = 0 인 경우

=> ex)  $5 * (-3) = -15$

- [2] sign\_minus = 1 인 경우

if(temp\_result % temp\_operand == 0)

=> ex)  $(-6) / (-3) = 2$

else

=> ex)  $(-5) / (-3) = 2$

<사진 3>

- input\_operator = 5

나머지 연산자를 입력받은 상황이다. 해당 연산은 입력받은 피연산자의 부호를 결정하는 input\_minus의 영향을 받지 않는다. 따라서 sign\_minus에 따라 경우를 나누었다.

- ① sign\_minus = 0 인 경우

=> ex)  $5 \% 3 = 2, 5 \% (-3) = 2$

- ② sign\_minus = 1 인 경우

if(temp\_result % temp\_operand == 0)

=> ex)  $(-6) \% 3 = 0$

else

=> ex)  $(-5) / 3 = 1, (-5) \% (-3) = 1$

※ 지금까지 설명한 input\_operator에 따라 case를 나누는 구문은 덧셈 연산자뿐만 아니라 뺄셈, 곱셈, 나눗셈 연산자, enter에서도 모두 같은 원리이다. 따라서 이후에 해당 구문이 나오는 부분은 생략할 것이다.

```

'h0080: begin //-
case (sw_status)
  sw_idle: begin
    set_no1 <= 20;
    set_no2 <= 20;
    set_no3 <= 20;
    set_no4 <= 20;
    set_no5 <= 20;
    set_no6 <= 20;
  end
  sw_start: begin
    input_minus <= 1;
    sw_status <= sw_s1;
    set_no1 <= 20;
    set_no2 <= 20;
    set_no3 <= 20;
    set_no4 <= 20;
    set_no5 <= 20;
    set_no6 <= 21;
  end

  default : begin
    sw_status <= sw_start;
    input_operator <= 2;
    pb_minus <= 1;
    set_no1 <= 20;
    set_no2 <= 20;
    set_no3 <= 20;
    set_no4 <= 20;
    set_no5 <= 20;
    set_no6 <= 20;
  end
end

```

<사진 1>

```

if(input_operator==6) begin
temp_result <= temp_operand*temp_operand;
end

```

<사진 2>

<사진 1>

• 입력받은 연산자가 'h0080일 경우 뺄셈 연산자를 입력받은 것이다. 기본적인 알고리즘은 덧셈 연산자와 동일하지만 몇 가지 차이점이 존재한다. 따라서 차이점만 골라 기능을 설명하겠다. (이후 곱셈 및 나눗셈 연산자 설명 방식도 동일)

• 덧셈의 경우에선 sw\_start에서 연산자가 다시 입력될 경우 예외처리를 해주었지만, 뺄셈에서는 피연산자(temp\_operand)의 음수 입력으로 받아들이기 때문에 예외처리를 하지 않았다. 따라서 input\_minus가 1이고 status도 숫자를 한 자리 입력받은 것으로 처리하여 sw\_s1으로 대입하였다.

• default 경우에선 덧셈과 동일하지만, 뺄셈 연산자 버튼을 눌렀으므로 pb\_minus의 값을 1로 설정하였다.

<사진 2>

• input\_operator가 6인 경우는 제곱 연산자가 대입된 경우이다. 덧셈, 곱셈, 나눗셈 연산자에는 해당 조건문이 없지만, 뺄셈 연산에서는 피연산자의 음수 대입도 있으므로, 제곱 연산 이후로 "-" 연산자는 연이어 입력이 가능하다. 따라서 다음과 같은 구문을 추가하였다.

```

'h0800: begin /*
case (sw_status)
    sw_idle: begin
        set_no1 <= 20;
        set_no2 <= 20;
        set_no3 <= 20;
        set_no4 <= 20;
        set_no5 <= 20;
        set_no6 <= 20;

    end
    sw_start: begin
        if(square==1) begin
            input_operator <=6;
            set_no1 <= 20;
            set_no2 <= 20;
            set_no3 <= 20;
            set_no4 <= 20;
            set_no5 <= 20;
            set_no6 <= 20;

        end
        else begin
            if(check_err==1)begin
                square <= 1;
                sw_status <= sw_start;
                set_no1 <= 20;
                set_no2 <= 20;
                set_no3 <= 20;
                set_no4 <= 20;
                set_no5 <= 20;
                set_no6 <= 20;
            end
        end
    end
end

```

<사진 1>

```

        else begin
            sw_status <= sw_idle;
            set_no1 <= 20;
            set_no2 <= 20;
            set_no3 <= 16;
            set_no4 <= 17;
            set_no5 <= 17;
            set_no6 <= 20;
            of <= 1;
        end
    end
end

default : begin
    square <= 1;
    sw_status <= sw_start;
    input_operator <= 3;
    set_no1 <= 20;
    set_no2 <= 20;
    set_no3 <= 20;
    set_no4 <= 20;
    set_no5 <= 20;
    set_no6 <= 20;
end

```

<사진 2>

#### <사진 1>

- 입력받은 연산자가 'h0800일 경우 곱셈 연산자를 입력받은 것으로 인식한다. 곱셈 연산자의 특징은 해당 버튼을 두 번 연속 입력 시, 제곱 연산자로 인식한다는 점이다. 따라서 square 변수를 사용하여 제곱 연산을 인식하도록 할 것이다.

#### <사진 1, 2>

- sw\_start(check\_err가 1인 경우) 또는 default 상태에서 곱셈 연산자를 누르면 square 변수의 값은 1로 대입된다. 이후 다시 한번 곱셈 연산자를 누르면 제곱 연산으로 변환되며 input\_operator에는 6이 저장된다.
- 먼저 sw\_start에서 check\_err가 1인 부분은 결론부터 말하자면, enter 버튼을 눌러 도출된 최종 결과값에서 제곱 연산을 수행하기 위해 만들어진 구문이다. check\_err는 enter 버튼을 누르면 값이 1로 대입된다. 이와 같은 구문이 없으면 enter 버튼을 눌러 도출된 값의 제곱값을 구할 수 없다.
- default 부분은 계산기에 식을 입력하는 도중 곱셈 연산을 누른 흔한 경우이다. 이때 곱셈 연산을 한번 눌렀으므로 input\_operator의 값은 3으로 대입되지만 해당 버튼을 한번 더 누르면 값은 6으로 변환된다.

```

'h8000: begin // /
case (sw_status)
    sw_idle: begin
        set_no1 <= 20;
        set_no2 <= 20;
        set_no3 <= 20;
        set_no4 <= 20;
        set_no5 <= 20;
        set_no6 <= 20;
    end
    sw_start: begin
        if(remainder==1) begin
            input_operator <=5;
            set_no1 <= 20;
            set_no2 <= 20;
            set_no3 <= 20;
            set_no4 <= 20;
            set_no5 <= 20;
            set_no6 <= 20;
        end
        else begin
            set_no1 <= 20;
            set_no2 <= 20;
            set_no3 <= 16;
            set_no4 <= 17;
            set_no5 <= 17;
            set_no6 <= 20;
            sw_status <= sw_idle;
        end
    end
end

```

<사진 1>

```

default : begin
    remainder <= 1;
    sw_status <= sw_start;
    input_operator <= 4;
    set_no1 <= 20;
    set_no2 <= 20;
    set_no3 <= 20;
    set_no4 <= 20;
    set_no5 <= 20;
    set_no6 <= 20;
end

```

<사진 2>

<사진 1, 2>

- 입력받은 연산자가 'h8000일 경우 나눗셈 연산자를 입력받은 것으로 인식한다. 나눗셈 연산자의 특징은 해당 버튼을 두 번 연속 입력 시, 나머지 연산자로 인식한다는 점이다. 따라서 remainder 변수를 사용하여 나머지 연산을 인식하도록 할 것이다.
- default 상태는 피연산자(숫자)를 이미 받고 연산자를 받을 상태이므로 나눗셈 연산자를 누르면 remainder 변수의 값은 1로 대입되며 input\_operator는 4로 저장된다. 또한 status는 sw\_start로 돌아간다. 이후 다시 한번 나눗셈 연산자를 누르면 나머지 연산으로 변환되며 input\_operator에는 5가 저장된다.
- sw\_start에서는 피연산자(숫자)를 받은 상태이다. 따라서 다른 연산자를 누르면 예외처리를 해야 한다. 하지만 나눗셈 연산자는 연속으로 두 번 누르면 나머지 연산으로 바뀌어야 하므로 조건문으로 remainder가 1인 경우를 추가하였다.



```

'h1000: begin // enter
case (sw_status)
  sw_idle: begin
    sw_status <= sw_start;
    set_no1 <= 20;
    set_no2 <= 20;
    set_no3 <= 20;
    set_no4 <= 20;
    set_no5 <= 20;
    set_no6 <= 20;
  end
  sw_start: begin
    if(input_operator==6) begin
      temp_result <= temp_result*temp_result;
      sw_status <= sw_s1;
      sign_minus <= 0;
      check_print <= 1;
    end
    set_no1 <= 20;
    set_no2 <= 20;
    set_no3 <= 20;
    set_no4 <= 20;
    set_no5 <= 20;
    set_no6 <= 20;
  end
end

```

<사진 1>

```

default: begin
  check_print <= 1;
  input_operator <= 0;
  check_err <= 1;
  if(input_operator==0) begin
    temp_result <= temp_operand;
  end
end

```

<사진 2>

```

'h4000: begin // clr
sw_status <= sw_idle;
set_no1 <= 20;
set_no2 <= 20;
set_no3 <= 20;
set_no4 <= 20;
set_no5 <= 20;
set_no6 <= 20;
input_operator <= 0;
sign_minus <= 0;
check_print <= 0;
square <= 0;
remainder <= 0;
input_minus <= 0;
pb_minus <= 0;
check_err <= 0;
of <= 0;
end

```

<사진 3>

#### <사진 1, 2>

- 입력받은 버튼이 'h1000이면 계산기는 enter 버튼으로 인식하여 최종 계산값을 temp\_result에 저장하고 7-segment에 출력하도록 도와주는 역할을 한다.
- sw\_status에 따라 코드가 나누어진다. sw\_start일 때, input\_operator가 6인 조건문이 추가되어 있다. 제곱 연산은 곱셈 연산자를 두 번 누르면 계산이 되어야 하므로, 연산자를 눌러 바뀐 상태인 sw\_start에서 enter를 눌러야 최종 결과가 도출되도록 해야 한다. 또한 default에서는 7-segment에 출력하기 위해 check\_print 변수를 1로 넣어주고 제곱 연산을 위해 check\_err를 1로 넣어준다. 그리고 계산이 종료되었기 때문에 input\_operator를 0으로 초기화해준다.
- 그 아래에 input\_operator가 0일 때의 조건문이 구현되어 있다. 이는 만일 처음에 피연산자만 입력하고 연산자를 입력하지 않은 채로 enter를 누르면, 입력한 피연산자를 바로 출력하도록 한 구문이다.
- 아래 input\_operator의 값에 따라 case 구문으로 나눈 알고리즘은 앞서 연산자에서 설명하였으므로 생략하겠다.

#### <사진 3>

- 입력받은 버튼이 'h4000이면 계산기는 clear 버튼으로 인식하여 계산기를 초기 상태로 되돌린다. 따라서 sw\_status는 sw\_idle로 대입하고, set\_no#은 아무 표시도 되지 않도록 20을 대입한다. 또한 나머지 변수들은 0으로 초기화해준다.

### ③ 음수의 피연산자(피연산자 및 연산자 입력)

```

else begin
    case (pb_1st) // each button
        'h0100: begin
            pb_minus <= 0;
            case (sw_status)
                sw_idle: begin
                    set_no1 <= 20;
                    set_no2 <= 20;
                    set_no3 <= 20;
                    set_no4 <= 20;
                    set_no5 <= 20;
                    set_no6 <= 20;

                end
                sw_start: begin
                    set_no1 <= 20;
                    set_no2 <= 20;
                    set_no3 <= 16;
                    set_no4 <= 17;
                    set_no5 <= 17;
                    set_no6 <= 20;
                    sw_status <= sw_idle;

                end
            end
        end
    end
end

```

<사진 1>

```

sw_s1: begin
    sw_status <= sw_s2;
    set_no5 <= 21;
    set_no6 <= 1;
    temp_operand <= 1;
    input_minus <= 1;

end

```

<사진 2>

※ 설계한 계산기의 전체적인 알고리즘은 입력한 피연산자가 양수 또는 음수에 따라 나뉘지는 큰 틀을 가진다. 지금까지 살펴본 알고리즘은 양수인 경우였기에, 이제 음수인 경우를 설명할 것이다. 다만, 전체적인 코드 내용은 크게 차이가 없으므로 몇 가지 차이점만 명시하여 설명하겠다.

<사진 1, 2>

- 왼쪽 사진은 숫자 1을 가리키는 keypad 'h0100의 코드이다. 앞서 양수인 경우와 다른 점은 set\_no#에 저장되는 값이다. 음수를 저장해야 한다는 점에서 숫자 앞에 "-" 부호를 붙여야 하고, 그만큼 표현할 수 있는 숫자의 범위는 1자리 감소한다.
- 먼저 sw\_start에서 보통 숫자를 입력하면 sw\_s1로 상태가 넘어가야 하지만, 해당 구문에선 예외처리를 하였다. 이는 앞에 "-" 부호를 입력하기 위함이다. 당연히 "-" 부호없이 숫자를 입력하면 양수가 되고, 이는 음수의 피연산자 구문에선 모순이 된다.
- sw\_s1은 이미 한 자리의 숫자가 입력되어있어야 하고, 추가로 한 자리를 더 입력하여 두 자리의 숫자를 만들어야 하는 부분이다. 하지만 음수의 피연산자는 두 자리의 숫자 대신 하나의 부호와 하나의 숫자로 이루어져 있다. 따라서 set\_no5에 21을 넣고 set\_no6에 1을 넣어 "-1"을 표현할 것임을 알 수 있다.
- 'h0100(숫자 1)뿐만 아니라 다른 숫자도 이와 같은 원리이다. 따라서 설명은 생략할 것이다.
- 연산자, enter, clear 버튼은 양수의 피연산자와 동일한 코드이다.

#### [4] 7-segment Display & Output

```
// 7-segment
always @(set_no1) begin
    case (set_no1)
        0: seg_100000 <= 'b0011_1111;
        1: seg_100000 <= 'b0000_0110;
        2: seg_100000 <= 'b0101_1011;
        3: seg_100000 <= 'b0100_1111;
        4: seg_100000 <= 'b0110_0110;
        5: seg_100000 <= 'b0110_1101;
        6: seg_100000 <= 'b0111_1101;
        7: seg_100000 <= 'b0000_0111;
        8: seg_100000 <= 'b0111_1111;
        9: seg_100000 <= 'b0110_0111;
        10: seg_100000 <= 'b0111_1000;
        11: seg_100000 <= 'b0111_0011;
        12: seg_100000 <= 'b0111_0111;
        13: seg_100000 <= 'b0111_1100;
        14: seg_100000 <= 'b0011_1001;
        15: seg_100000 <= 'b0101_1110;
        16: seg_100000 <= 'b0111_1001;
        17: seg_100000 <= 'b0101_0000;
        18: seg_100000 <= 'b0100_0000;
        19: seg_100000 <= 'b0101_0100;
        21: seg_100000 <= 'b0100_0000; // minus
        default: seg_100000 <= 'b0000_0000;
    endcase
end
```

<사진 1>

```
// fnd_clk. output
always @(posedge fnd_clk) begin
    fnd_cnt <= fnd_cnt + 1;
    case (fnd_cnt)
        5: begin
            fnd_d <= seg_100000;
            fnd_s <= 'b011111;
        end
        4: begin
            fnd_d <= seg_10000;
            fnd_s <= 'b101111;
        end
        3: begin
            fnd_d <= seg_1000;
            fnd_s <= 'b110111;
        end
        2: begin
            fnd_d <= seg_100;
            fnd_s <= 'b111011;
        end
        1: begin
            fnd_d <= seg_10;
            fnd_s <= 'b111101;
        end
        0: begin
            fnd_d <= seg_1;
            fnd_s <= 'b111110;
        end
    endcase
end
endmodule
```

<사진 2>

<사진 1>

- 앞서 지금까지 저장한 set\_no#의 값에 대하여 0부터 21까지 case 구문으로 나누어 7-segment의 display를 담당하는 구문이다. 7-segment의 출력은 총 8-bit에서 영역마다 나누어져 있으며, 이는 8-bit의 이진수를 seg\_# 변수에 대입하여 할 수 있다.
- 왼쪽 사진은 set\_no1인 경우이며, set\_no2 ~ set\_no5는 set\_no1와 같은 코드이므로 생략한다.
- 0 ~ 21까지의 출력 유형은 "상세한 기능설명-초기값" 부분에 명시하였다.

<사진 2>

- 7-segment의 출력을 위한 구문이다. <사진 1>의 구문을 거쳐 최종 출력은 여기서 이루어진다고 할 수 있다.
- fnd\_clk이 rising edge일 때 fnd\_cnt의 값이 1씩 증가하고 case 구문을 fnd\_cnt 값으로 나누었다. 이는 7-segment의 출력을 담당하는 signal인 fnd\_clk에 따라 fnd\_cnt 값이 결정되고, 7-segment 6자리 중 한 자리씩 fnd\_cnt에 따른 case 구문을 통해 출력하도록 하는 알고리즘이라 할 수 있다.
- 각 case마다 앞서 저장했던 seg\_# 변수의 값을 fnd\_d에 저장하고, fnd\_s에는 7-segment의 해당 자릿수를 출력하도록 6-bit의 이진수를 저장한다.



#### 4. 기능별 결과 사진

##### [1] 기본 사칙연산 + 나머지, 제곱 (양수)

① 덧셈 연산 (  $27 + 133 = 160$  )

초기 상태 or CLR 상태	27 입력	+ 입력 후 133 입력	= 입력시 결과 출력

② 뺄셈 연산 (  $166 - 122 = 44$  )

초기 상태 or CLR 상태	166 입력	- 입력 후 122 입력	= 입력시 결과 출력

③ 곱셈 연산 (  $44 * 3 = 132$  )

초기 상태 or CLR 상태	44 입력	* 입력 후 3 입력	= 입력시 결과 출력

④ 나눗셈 연산 (  $122 / 12 = 10 \dots 2$  )

초기 상태 or CLR 상태	122 입력	/ 입력 후 12 입력	= 입력시 결과 출력

⑤ 나머지 연산 (  $122 / 12 = 10 \dots 2$  )

초기 상태 or CLR 상태	122 입력	// 입력 후 12 입력	= 입력시 결과 출력

⑥ 제곱 연산 (  $12 ^ \wedge = 144$  )

초기 상태 or CLR 상태	12 입력	^ 입력	= 입력시 결과 출력

##### [2] 음수 입출력 사칙연산

① 덧셈 연산

(1) 음수 + 양수 (  $-7 + 9 = 2$  )

초기 상태 or CLR 상태	-7 입력	+ 입력 후 9 입력	= 입력시 결과 출력

(2) 양수 + 음수 (  $10 + -20 = -10$  )

초기 상태 or CLR 상태	10 입력	+ 입력 후 -20 입력	= 입력시 결과 출력

② 뺄셈 연산

(1) 음수 - 양수 (  $-5 - 7 = -12$  )

초기 상태 or CLR 상태	-5 입력	- 입력 후 7 입력	= 입력시 결과 출력

(2) 양수 - 음수 (  $50 - -7 = 57$  )

초기 상태 or CLR 상태	50 입력	- 입력 후 -7 입력	= 입력시 결과 출력

### ③ 곱셈 연산

(1) 양수 \* 음수 (  $5 * -10 = -50$  )

초기 상태 or CLR 상태	5 입력	* 입력 후 -10 입력	= 입력시 결과 출력

(2) 음수 \* 음수 (  $-10 * -7 = 70$  )

초기 상태 or CLR 상태	-10 입력	* 입력 후 -7 입력	= 입력시 결과 출력

### ④ 나눗셈 연산

- 음수의 나눗셈, 나머지에 관해서 수학적인 정확한 정의가 존재하지 않는다. 관련 내용을 고찰에 아주 자세하게 작성해 두었다. 우리는 본 프로그램을 구현하는 과정에서 해당 방식을 대전제로 설계하였다.

(1) 양수 / 음수 (  $11 / -2 = -5 \dots 1$  )

초기 상태 or CLR 상태	11 입력	/ 입력 후 -2 입력	= 입력시 결과 출력

(2) 음수 / 양수 (  $-7 / 5 = -2 \dots 3$  )

초기 상태 or CLR 상태	-7 입력	/ 입력 후 5 입력	= 입력시 결과 출력

### ⑤ 나머지 연산 ( $11 / -2 = -5 \dots 1$ )

초기 상태 or CLR 상태	11 입력	// 입력 후 -2 입력	= 입력시 결과 출력

### ⑥ 제곱 연산 ( $-10 ^ =$ )

초기 상태 or CLR 상태	-10 입력	^ 입력	= 입력시 결과 출력

## [3] 초기화 버튼

초기 상태 or CLR 상태	임의의 숫자 및 수식 입력	CLR 입력시 초기화

#### [4] 예외 처리 구현

##### ① 초기 상태 혹은 CLR 상태에서 연산자를 우선 입력한 경우

초기 상태 or CLR 상태	연산자 입력	Err 출력	다시 CLR 누르면 초기화

- 본 Err 발생 역시 아래 "연산자 2개 연속 입력시 발생하는 Err"와 동일한 코드를 통해 출력된다. 그 이유는 CLR 상태는 sw\_status가 sw\_start 인 상태이다. 이 상태에서 연산자가 입력되는 경우 Err를 출력하게 된다. 의미 없는 연산자 2개 연속 입력 또한 sw\_start 상태에서 숫자나 - 가 아닌 연산자가 입력되는 것이므로 동일한 코드를 통해 Err를 출력하는 것이다.

##### ② 의미 없는 연산자 2개를 연속해서 입력하는 경우

초기 상태 or CLR 상태	+ 입력	+ 재 입력 시	다시 CLR 누르면 초기화

초기 상태 or CLR 상태	+ 입력	* 재 입력 시	다시 CLR 누르면 초기화

```

465      'h0008: begin //+
475      sw_start: begin //twice + error
476          set_no1 <= 20;
477          set_no2 <= 20;
478          set_no3 <= 16;
479          set_no4 <= 17;
480          set_no5 <= 17;
481          set_no6 <= 20;
482          sw_status <= sw_idle;
483      end
  
```

단, 연산자 이후에 -를 입력하면 음수 입력으로 인식하여 Err가 발생하지 않는다.

초기 상태 or CLR 상태	+ 입력	- 재 입력 시	이후에 원하는 숫자 입력

```

1016      'h0080: begin //-
1017      case (sw_status)
1026          sw_start: begin
1027              input_minus <= 1;
1028              sw_status <= sw_s1;
1029              set_no1 <= 20;
1030              set_no2 <= 20;
1031              set_no3 <= 20;
1032              set_no4 <= 20;
1033              set_no5 <= 20;
1034              set_no6 <= 21;
1035      end
  
```

③ 0으로 나누는 경우 ( 임의의 숫자 / 0 = Err )

초기 상태 or CLR 상태	임의의 숫자 (ex.12) 입력	/ 입력후 0 입력	= 입력시 Err 출력

```

1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
if(input_operator==4) begin
    if(temp_operand==0) begin
        sw_status <= sw_idle;
        set_no1 <= 20;
        set_no2 <= 20;
        set_no3 <= 16;
        set_no4 <= 17;
        set_no5 <= 17;
        set_no6 <= 20;
        of <= 1;
    end
end

```

[5] Overflow, Underflow

① 연산 도중 Overflow 발생시 ( 999,998 + 5 \* ... -> 이 경우 두 번째 입력 연산자인 \*에서 Err 발생 )

초기 상태 or CLR 상태	999,998 입력	+ 입력후 5 입력	연산자 입력시 Err

```

3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
else begin
    if(temp_result+temp_operand>99999) begin //minus overflow
        set_no1 <= 20;
        set_no2 <= 20;
        set_no3 <= 16;
        set_no4 <= 17;
        set_no5 <= 17;
        set_no6 <= 20;
        of <= 1;
        sw_status <= sw_idle;
    end
end

```

② 연산 결과가 Overflow 발생시 ( 999,999 + -99 + 100 = ?? )

초기 상태 or CLR 상태	999,999 입력	+ 입력 후 - 99 입력	+ 입력후 100 입력

이후에 = 입력시 Err

● Err가 발생하는 코드가 "④ 연산 결과가 Underflow 발생시 ( -55,555 \* 3 = ? )" 와 동일하다.

③ 연산 도중 Underflow 발생시 ( -99,999 - 110 + ... )

초기 상태 or CLR 상태	-99,999 입력	- 입력 후 110 입력	연산자 입력시 Err

● Err가 발생하는 코드가 "① 연산 도중 Overflow 발생시"와 동일하다.

④ 연산 결과가 Underflow 발생시 ( -55,555 \* 3 = ? )

초기 상태 or CLR 상태	-55,555 입력	* 입력 후 3 입력	= 입력시 Err

```

208      if (temp_result >= 1000000) begin
209          set_no1 <= 20; //
210          set_no2 <= 20; //
211          set_no3 <= 16; //E
212          set_no4 <= 17; //r
213          set_no5 <= 17; //r
214          set_no6 <= 20; //
215          of <= 1;
216      end
217      check_print <= 0;
218      sw_status <= sw_start;

```

⑤ 양수 여섯 자리 입력 후 숫자를 더 입력하는 경우 ( 123,456 입력 된 상태에서 임의의 숫자 7을 입력한 경우)

초기 상태 or CLR 상태	123,456 입력	임의의 숫자 입력 시 Err

```

1408      sw_s6: begin
1409          sw_status <= sw_idle;
1410          set_no1 <= 20;
1411          set_no2 <= 20;
1412          set_no3 <= 16;
1413          set_no4 <= 17;
1414          set_no5 <= 17;
1415          set_no6 <= 20;
1416      end
1417

```

⑥ 음수 다섯 자리 입력 후 숫자를 더 입력하는 경우 ( -34,567 입력 된 상태에서 임의의 숫자 8을 입력한 경우)

초기 상태 or CLR 상태	-34,567 입력	임의의 숫자 입력 시 Err

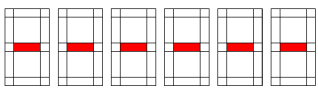
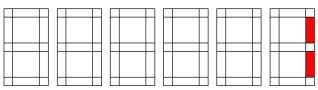
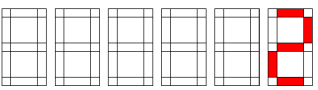
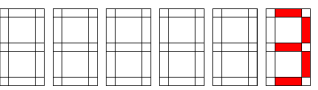
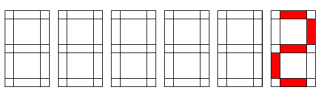
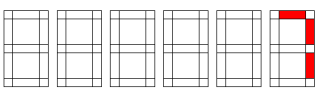
```

4099      sw_s6: begin
4100          sw_status <= sw_idle;
4101          set_no1 <= 20;
4102          set_no2 <= 20;
4103          set_no3 <= 16;
4104          set_no4 <= 17;
4105          set_no5 <= 17;
4106          set_no6 <= 20;
4107      end
4108

```

[6] 기타 정상 구현 확인

① 연산자 우선 순위 무시 ( 1 + 2 \* 3 - 2 = 7 )

초기 상태 or CLR 상태	1 입력	+ 입력 후 2 입력	* 입력 후 3 입력
			
- 입력 후 2 입력	= 입력 시 결과 출력		
			

## 5. 개인별 기여도 평가(기능별) 및 고찰

### [1] 개인별 기여도 평가

		안재혁 (201821094)	박상현 (201920941)
설계 예외 처리	Overflow, Underflow 발생 시 Err 출력	코드 분석, 설계 방향 제시, 결과 확인 및 검토	구현 방법 고안, 설계, 오류수정
	0으로 나누면 Err 출력	코드 분석, 설계 방향 제시, 결과 확인 및 검토	구현 방법 고안, 설계, 오류수정
	+, - 연산자를 2회 이상 연속 입력하였으면 Err 출력	구현 방법 고안, 설계, 오류수정	코드 분석, 설계 방향 제시, 결과 확인 및 검토
	-0이 출력되는 경우	구현 방법 고안, 설계, 오류수정	코드 분석, 설계 방향 제시, 결과 확인 및 검토
구현	^(제곱), %(나머지) 구현	구현 방법 고안, 설계, 오류수정	코드 분석, 설계 방향 제시, 결과 확인 및 검토
	음수 출력 및 연산 구현	코드 분석, 설계 방향 제시, 결과 확인 및 검토	구현 방법 고안, 설계, 오류수정
	숫자, 연산기호 입력 구현	구현 방법 고안, 설계, 오류수정	코드 분석, 설계 방향 제시, 결과 확인 및 검토
보고서	설계 계획서	설계 목표 제시, 과제 요구사항 분석 및 해결 방안 제시 및 관련 문서 작성, Verilog 문법 관련 자료 조사 및 설계에 필요한 내용 테스트	설계 과정에서 예상되는 문제점 분석 및 관련 해결 방안 제시, 동작 방식 제시 및 관련 문서 작성, 7-segment, button 관련 입출력 내용 자료 조사
	설계 보고서	프로젝트 결과 확인 및 검토, 설계, 추가 구현 관련 문서 작성	설계 목표 달성 정도 확인 및 검토, 수식 입력 및 계산과정 구현 관련 문서 작성

### [2] 고찰

[고찰 1] 제곱 연산과 관련하여 square 이라는 변수를 굳이 만들었어야 하는 생각이 든다. 물론 하나의 변수를 추가한다고 해서 성능에 큰 영향을 주지 않는다. 하지만 non-blocking의 특성을 이용했다라면 square이라는 변수가 없어도 되었을 것 같다는 생각이 보고서 작성 도중에 들었다

square이라는 변수는 (1) 첫 번째 \* 입력과 두 번째 \*을 구별하기 위해서, (2) sw\_start 상태에서 \*가 입력되었을때 제곱을 위해 입력된 것인지, 실수로 입력했기에 Err를 표시해야 하는 것인지 이 두 가지를 해결하기 위해 만들어진 변수이다.

하지만 (1) 첫 번째의 경우에는 sw\_status를 보고 구분해 낼 수가 있다. sw\_status가 sw\_s1 ~ sw\_s6라면 이는 첫 번째 \* 입력이며, sw\_status가 start라면 이는 두 번째 \*이다. 또한 (2) 두 번째의 경우에도 input\_operator=3인지 확인한다면 해결할 수 있다. sw\_start 상태에서 input\_operator가 3이 아니라는 뜻은 직전에 입력한 연산자가 \*이 아니라는 뜻이다. 즉 이 경우에는 Err를 표시하면 된다. 하지만 input\_operator가 3이면 직전에 입력한 연산자가 \* 이므로 제곱 연산을 수행하도록 input\_operator에 6을 저장하면 된다.

한가지 추가로 고려해야 하는 것은, 연산이 모두 끝난 이후(=을 눌러 연산의 결과를 본 후)에 제곱을 눌러 확인하고자 하는 경우를 생각해 보아야 하는데, 이 경우 역시 input\_operator를 조금만 고려했으면 되었을 것 같다. =을 입력하면 input\_operator는 0이 된다. 즉, 'h0800(\*)이 sw\_start 상태에서 입력되었을 때 input\_operator=0이고, temp\_result가 0이 아니라면 input\_operator에 3을 저장하고 다시 한번 더 'h0800(\*)이 sw\_start 상태에서 입력된다면 input\_operator=6을 주면 제곱 연산을 수행할 수 있었다고 생각한다.

이는 비슷한 원리로 작동하는 나눗셈의 나머지 연산 (% , // 두 번으로 구현, square 대신 remainder 변수 사용,

input\_operator=5)에서도 비슷하게 적용해서 변수를 줄일 수 있었을 것이라고 생각한다.

[고찰 2]

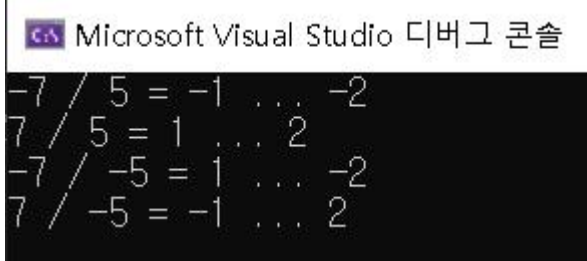
수학적으로 음수의 나눗셈, 음수의 나머지에 관해 정확한 정의가 존재하지 않는다. 예를 들어 -7 나누기 5는 -2라고 해야하는가 -1이라고 해야하는가? 프로그램들마다 음수의 나눗셈 계산 방식이 다르다. Excel에서는 몫을 계산할 때 QUOTIENT 함수를 사용하며, 나머지를 계산할 때 MOD 함수를 사용한다. 아래는 Excel을 사용하여 -7 나누기 5를 계산한 결과이다.

A	B	몫	나머지
-7	5	-1	3

Excel은 음수의 나눗셈에서 몫은 양수 / 양수를 한 후에 - 부호를 붙이는 방식으로 계산했다. 그러나 이 방식의 가장 큰 문제점은 역셈하면 식이 성립하지 않는다는 문제가 존재한다. A 나누기 B 이고 몫이 C이고 나머지가 D라면  $B * C + D = A$ 가 되어야 한다. 그러나 Excel의 계산을 역셈하면  $5 * -1 + 3 = -2 \neq -7$  이다. 즉, 우리는 Excel의 방식을 사용 할 수가 없게 되었다.

C++ 에서는 어떠한 방식으로 나눗셈의 몫과 나머지를 연산하는지 확인해보기로 하였다. 따라서 아래와 같이 코드를 작성하였고 print를 확인해 보았다.

```
1  #include <stdio.h>
2
3  void test(int a, int b)
4  {
5      printf("%d / %d = %d ... %d",a, b, a/b, a%b);
6  }
7
8  int main(){
9      test(-7, 5);
10     test(7, 5);
11     test(-7, -5);
12     test(7, -5);
13
14     return 0;
15 }
```



Excel과 다르게 C++에서는 양수 / 양수의 값을 몫으로 정하고 음수가 포함되어있다면 - 부호를 붙이는 방식으로 진행하였으며, A 나누기 B 이고 몫이 C이고 나머지가 D라면 나머지는  $A - (B * C)$  으로 계산하였다. 이 방식의 단점은 나머지가 음수가 된다는 것이다. 이렇게 프로그램마다 연산 방식이 다르고, 정해진 방식이 별도로 존재하지 않는다고 생각하여 우리는 다음과 같이 임의로 두 가지 대전제를 만들었고 그 대전제를 꼭 따르도록 코드를 구현하였다.

1. 나머지는 무조건 자연수로 나와야 한다.



## 2. 역셈 과정에서도 식이 성립해야한다.

첫 번째 대전제를 만든 이유는 -1개가 남는다는 것에 대해 동의할 수가 없었기 때문이다. 나머지는 나누었을 때 남는 숫자이다. -1이 남는다는 것은 1이 부족하다는 것과 동일하다. 그리고 1이 부족하다는 것은 절대 나머지라고 할 수가 없다. 두 번째 대전제를 만든 이유는 너무나도 당연했다. 역셈이 성립하지 않는 수학은 존재하여서는 안된다고 생각했다. 계산기는 어떠한 방법으로 계산을 하더라도 늘 동일한 값이 나와야 하기에 역셈은 무조건 성립해야한다. 따라서 위의 두가지 대전제를 만들었다.

이 두 가지 대전제를 따르기 위해서는 Excel과 C++ 어떠한 방식도 사용할 수가 없었다. Excel은 두 번째 대전제를 만족하지 못했으며, C++는 첫 번째 대전제를 만족하지 못하였다. 결국 우리만의 나눗셈, 나머지 진리표를 생성하였으며, 이 표를 따르도록 코드를 구현하였다.

기본적으로 양수 ÷ 양수 꼴에서는 문제가 없었기에 양수 ÷ 양수 꼴을 기반으로 음수 영역에 대한 진리표를 만들었고 다음과 같은 진리표를 생성해 낼 수 있었다.

6 ÷ 3 = 2 ... 0	6 ÷ -3 = -2 ... 0
5 ÷ 3 = 1 ... 2	5 ÷ -3 = -1 ... 2
4 ÷ 3 = 1 ... 1	4 ÷ -3 = -1 ... 1
3 ÷ 3 = 1 ... 0	3 ÷ -3 = -1 ... 0
2 ÷ 3 = 0 ... 2	2 ÷ -3 = 0 ... 2
1 ÷ 3 = 0 ... 1	1 ÷ -3 = 0 ... 1
0 ÷ 3 = 0 ... 0	0 ÷ -3 = 0 ... 0
-1 ÷ 3 = -1 ... 2	-1 ÷ -3 = 1 ... 2
-2 ÷ 3 = -1 ... 1	-2 ÷ -3 = 1 ... 1
-3 ÷ 3 = -1 ... 0	-3 ÷ -3 = 1 ... 0
-4 ÷ 3 = -2 ... 2	-4 ÷ -3 = 2 ... 2
-5 ÷ 3 = -2 ... 1	-5 ÷ -3 = 2 ... 1

양수 ÷ 양수 꼴을 우선 작성한 뒤, 나누는 수가 음수가 되더라도 나머지가 자연수가 될 수 있도록 진리표를 작성하였다. 나누는 수가 음수인 영역에 대해서는 몫에 -를 붙이는 것으로 진리표를 작성하였다. 이 방식을 통해 우리는 우리가 정한 두 개의 대전제인 나머지가 양수이며, 역셈이 성립하는 계산 방식을 만들 수 있었다.

이 부분에서 다소 아쉬운 점은 음수의 몫, 나머지에 관해 전문가의 견해를 듣지 못한 점이다. 음수 구현을 어느 정도 하던 중에 이러한 수학적 문제가 발생한 점을 확인하였다. 그리고 그 시점은 수학 관련 전문가에게 조언을 구하기에는 다소 늦은 시간이었다. 일부 문서에서는 제수(나누는 수)는 양수이어야 하므로 " 6 ÷ -3 "또한 Err로 표시해야 한다는 내용이 있었다. 조금 더 많은 전문적인 자료를 찾았으면 어땠을까 하는 생각이 들었다.

### [고찰 3]

임베디드 시스템을 구현할 때 생각해야 할 중요한 요소 중 하나는 알고리즘의 효율성이다. 비록 우리가 설계한 것은 기본 연산과 제곱 그리고 음수 계산이 되는 간단한 계산기지만, verilog 언어로 구현하였기 때문에 알고리즘의 효율성을 고려해야 하며 최대한 계산기가 빠른 시간 안에 로딩이 되는 것이 좋은 알고리즘이라 할 수 있다. 이는 코드를 작성하고 쿼터스에서 compile을 실행할 때 진행되는 로딩 시간을 보고 생각하게 되었다. 기본 연산(+, -, \*, /, %)과 제곱 기능을 구현할 때는 반복문과 조건문 그리고 대입 연산이 많지 않았으므로 로딩 시간이 매우 길지 않았다. 하지만 음수 연산을 위해 앞서 제곱 기능까지 구현했던 코드를 한 번 더 구현하니 로딩 시간이 3~4분으로 매우 길어졌음을 확인했다. 컴파일을 진행하는 프로세서 입장에서 분석하고 가동해야 할 일이 더 많아져서 시간이 오래 소요되는 것은 당연했다. 음수 계산 부분을 넣기 전에는 대략 2800줄의 알고리즘이었지만 넣은 후에는 대략 5300줄까지 증가하여 확연히 연산 개수를 세지 않아도 알고리즘의 효율성이 감소했음을 알 수 있었다.

그리하여 계산기를 구현할 때 최대한 코드의 반복되는 부분과 필요 없는 부분을 줄이고, 변수 사용 횟수도 최대한

고려하였다. 하지만 우리에게 주어진 설계 시간은 최대한 효율적인 알고리즘을 구현하는 데 있어서 충분하지 않았고, 현재 완성한 알고리즘도 효율성이 매우 좋은 알고리즘이라고 생각하지 않는다. "자료구조 및 알고리즘"에서 배웠던 빅오표기법을 이용하면 우리가 구현한 알고리즘은 대략  $O(n \log n)$ 이라 할 수 있으며, 빅오표기법의 원리에 의하면 좋은 알고리즘은 아니라고 할 수 있다. 따라서 우리에게 조금 더 설계 시간이 주어졌더라면, 매우 간략하고 속도 빠른 알고리즘을 구현할 수 있었다고 생각한다.

또한 이번 설계에서 쓰인 verilog의 포인트는 blocking(=)과 non-blocking(<=) 사용이라고 생각한다. C언어는 컴파일러가 코드의 맨 처음부터 순차적으로 컴파일을 진행하지만, verilog는 non-blocking을 사용하면 해당 부분은 컴파일을 다른 코드 부분과 동시에 진행한다는 특징이 있다. 특히 이번 설계에서는 clock signal을 이용한 sequential circuit이며, 해당 회로를 verilog 언어로 구현하기 위해선 clock의 rising edge마다 status가 변화하기 때문에 non-blocking 사용이 매우 중요했다. 실제로 non-blocking을 적절하게 사용하지 못하면 status가 변화할 때 해당 변수가 대입할 값을 인식하지 못하는 상황이 발생하였고, 수 차례의 코드 수정 끝에 해당 사실을 알게되었다. 따라서 status가 변화할 때 동작하는 문법인 always 구문에선 꼭 blocking을 써야 하지 않는 이상 보통 non-blocking을 사용해야 한다. 그리고 Verilog로 계산기뿐만 아니라 다른 기기를 설계할 때 "변화하는 status에 따라 어떤 변수가 무슨 값으로 어느 타이밍에 대입되는지"를 잘 생각하면서 코드를 작성해야 한다는 점을 깨달았다.

## 6. 참고 문헌

- "7-segment" <<https://makernambo.com/77>>
- "overflow" <네이버 지식백과>  
<<https://terms.naver.com/entry.naver?docId=756117&cid=42341&categoryId=42341>>
- "underflow" <네이버 지식백과>  
<<https://terms.naver.com/entry.naver?docId=592778&cid=42340&categoryId=42340>>
- John F. Wakerly, 『DIGITAL DESIGN Principles and Practices』, Fifth edition, Pearson