디지털 시스템 설계 및 실험 결과보고서

이름 : 박지혁, 윤준석 | 학번 : 2016160145, 2016160109

실험제목	기말 프로젝트 - Fast Inverse Square Root Calculator
실험목표	① IEEE 754 표준에 대해 알아보고 Fast Inverse Square Root 코드를 분석하여 Verilog로 이를 구현한다. ② 실제 보드를 이용한 Fast Inverse Square Root Calculator를 설계한다.

실험결과

1. 사전지식

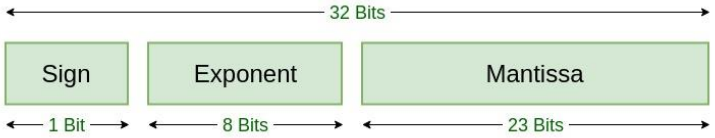
1) IEEE 754

(1) IEEE 754 표준

IEEE 754는 부동소수점을 표현하는 표준이다. 32 bit single precision는 반드시 따르도록 되어 있으며, 64 bit double precision, 43 bit 이상의 extended single precision, 79 bit 이상의 extended double precision 등으로도 구현된다. 32 bit인 경우 최상위 bit는 부호를 나타내며, 음수가 1, 양수가 0이 된다. 23~31번째 bit는 지수(exponent), 나머지는 가수부(mantissa)를 나타낸다.

IEEE 754 표준으로의 형식 변환 알고리즘은 다음과 같다.

1. 부호부를 sign bit으로 표현한다. (0 for positive, 1 for negative)
2. 절대값을 이진법으로 표현한다.
3. 정규화 한다. (이진수로 나타낸 수의 소수점을 왼쪽으로 이동시켜 왼쪽에 1만 남게 만들고, 이동시킨 횟수만큼 지수를 더하거나 뺀다.)
4. 정규화 된 값에서 가장 왼쪽의 1을 제외하고, 소수점 오른쪽의 수를 가수부로 정한다.
5. 지수에 bias 127을 더하여 지수부로 정한다.



Single Precision
IEEE 754 Floating-Point Standard

그림1 IEEE 754 Floating-Point Standard

(2) IEEE 754 exception definitions

IEEE 754는 지수부와 가수부를 분리해 표현하므로 아주 작은 수부터 아주 큰 수까지 넓은 범위의 값을 표현할 수 있고, 그 범위는 다음과 같다.

Level	Width	Range at full precision	Precision
Single precision	32 bits	$\pm 1.18\times10^{-38}$ to $\pm 3.4\times10^{38}$	Approximately 7 decimal digits
Double precision	64 bits	$\pm 2.23\times10^{-308}$ to $\pm 1.80\times10^{308}$	Approximately 16 decimal digits

이러한 범위를 넘어서는 값에 대해서는 아래와 같은 예외가 정의되어 있다.

* NaN :

NaN은 연산과정에서 잘못된 입력(inf+inf, 0/0)을 받아 계산을 하지 못했음을 나타내는 기호이다.

지수부(exponent)의 모든 bit는 1로 채운다.

가수부(mantissa)의 값은 0이 아니어야 한다.

부호부(sign)은 의미를 두지 않는다.

* inf :

overflow가 발생하였을때 표현하는 값을 정의한다.

지수부(exponent)의 모든 bit는 1로 채운다.

가수부(mantissa)의 모든 bit는 0으로 채운다.

부호부(sign)에 따라 양의 무한대와 음의 무한대로 구분된다.

* signed zero :

IEEE 754에서는 부호가 있는 -0, +0을 구분하고 있다.

단 if (x == 0)과 같은 작업에서 불확실한 결과를 낼 수 있기 때문에 -0 = +0 이라는 규칙을 정해 놓았다.

지수부(exponent)의 모든 bit는 0으로 채운다.

가수부(mantissa)의 모든 bit 역시 0으로 채운다.

부호부(sign)에 따라 +0 과 -0으로 구분된다.

* denormalized :

1.00 x 2^(-126) 보다 작아 표현 불가능한 수

지수부(exponent)의 모든 bit는 0으로 채운다.

부호부(sign)와 가수부(mantissa)의 값을 표현한다

(3) IEEE 754 addition

다음과 같은 예시로 IEEE 754의 덧셈 알고리즘을 간략히 설명하자.

$$A = (-1)^{s1} \times m1 \times 2^{e1}, \qquad B = (-1)^{s2} \times m2 \times 2^{e2}$$

1. 지수부 e1과 e2 중 작은 값을 큰 값과 같도록 맞춘다. 이때, 가수부는 변경되는 지수부 만큼 shift 한다.

예를 들어, e1 < e2 라면 A의 e1에 (e2-e1)을 더하여 e1을 e2로 만들고, 가수부 m1은 (e2-e1) 만큼 right shift 한다.

2. 가수부끼리 덧셈을 계산한다. ($(m1 \ll (e2 - e1)) + m2$)
3. 계산된 결과를 정규화 한다.
4. 가수부 bit에 맞도록 반올림한다. (Use Guard, Round, Sticky bits)

(4) IEEE 754 multiplication

다음과 같은 예시로 IEEE 754의 곱셈 알고리즘을 간략히 설명하자.

$$A = (-1)^{s1} \times m1 \times 2^{e1}, \qquad B = (-1)^{s2} \times m2 \times 2^{e2}$$

1. 가수부끼리 곱셈을 계산한다. ($m1 \times m2$)
2. 지수부끼리 덧셈을 계산한다. ($e1 + e2$)
3. 계산될 결과를 정규화 한다.
4. 가수부 bit에 맞도록 반올림한다. (Use Guard, Round, Sticky bits)

2) IEEE 754를 이용한 로그 근사

IEEE 754를 이용한 로그 근사는 본문의 FISR 알고리즘 구현에 있어 가장 핵심적인 부분이다.

임의의 양수 N을 Single Precision 32 bit으로 표현하여 $F = \{S, E[7:0], M[22:0]\}$로 나타냈다고 하자.

양수이기에 S는 항상 0이고, E는 M보다 23 비트 왼쪽에 있기에 F를 이진수로 해석하면

$$F = 0 \cdot 2^{31} + E \cdot 2^{23} + M = E \cdot 2^{23} + M$$

이다.

한편, N을 $F(S,E,M)$을 이용해 표현하면 mantissa에서 제거한 1.000과, exponent에 추가한 bias 127에 의해

$$N = \left(1 + \frac{M}{2^{23}}\right) \cdot 2^{E-127}$$

로 나타낼 수 있다.

이러한 IEEE 754의 표현법은 N의 bit representation인 F 그 자체가 N에 로그를 취한 값과 연관이 있다는 통찰을 준다. N에 로그를 취하면 $\log_2 N = \log_2\left(1 + \frac{M}{2^{23}}\right) + E - 127$ 이다. 한편, 0과 1 사이의 양수 x에 대해 $\log_2(1+x) = x + \mu$ (μ for correction term)으로 근사 가능하므로

$$\log_2 N = \frac{M}{2^{23}} + \mu + E - 127 = \frac{1}{2^{23}}(E \cdot 2^{23} + M) + \mu - 127 = \frac{1}{2^{23}}F + \mu - 127$$

이다. 이때, 로그 근사의 correction term μ는 $\mu = 0.0430$ 일 때 가장 적은 오차를 내는 것이 확인되었다. 따라서 $\log_2 N$은 F를 23번 right shifting한 값에 $\mu - 127$을 더하여 손쉽게 얻어질 수 있다.

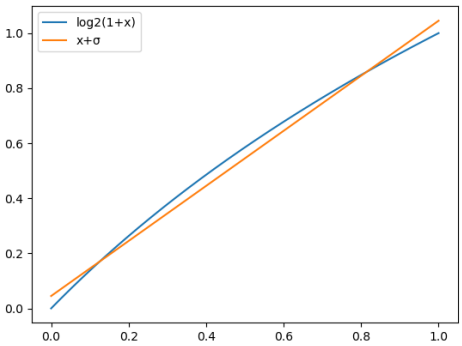


그림2 log2의 선형 근사 w/ correction term

3) Fast Inverse Square Root Algorithm

Fast Inverse Square Root (이하 FISR) 알고리즘은 루트와 나눗셈 계산이 없이 $1/\sqrt{y}$ 를 계산하여 기존의 계산법보다 4배 더 빠른 속도를 낸다. 이는 IEEE 754 표준을 이용한 일종의 트릭으로 구현 가능하다. FISR은 앞서 설명한 로그 근사를 이용하여 $1/\sqrt{y}$ 대신 $-\frac{1}{2}\log_2 y$를 계산하고 이를 다시 float32 타입으로 읽어낸다.

C로 작성된 아래의 원본 코드를 참조하여 FISR의 대략적인 원리를 설명하겠다.

```
float Q_rsqrt( float number ){
        long i;
        float x2, y;
        const float threehalfs = 1.5F;

        x2 = number * 0.5F;
        y = number;
        i = * ( long * ) &y; // evil floating point bit level hacking
        i = 0x5f3759df - ( i >> 1 ); // what the f***?
        y = * ( float * ) &i;
        y = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
//      y = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration, this can be removed

        return y;
}
```

· 7번째 줄에서 y = number 는 IEEE 754 표준을 따르는 float32 타입의 값으로, $F_y = \{ S_y, E_y[7:0], M_y[22:0] \}$로 구성된다.

· 8번째 줄에서 long i는 long 타입 포인터를 사용하여 메모리에 저장된 y의 float32 비트 정보들을 그대로 복사한다.

· 9번째 줄을 설명하기에 앞서 위에서 설명한 로그 근사를 바탕으로 간단한 수식을 전개하겠다.

$1/\sqrt{y}$의 해를 Γ 라 할 때, $\Gamma = 1/\sqrt{y}$ 의 양변을 로그 근사하면 다음과 같다.

$$\frac{1}{2^{23}}(E_\Gamma \cdot 2^{23} + M_\Gamma) + \mu - 127 = -\frac{1}{2}(\frac{1}{2^{23}}(E_y \cdot 2^{23} + M_y) + \mu - 127)$$

이를 정리하여 Γ와 y의 bit representation F_Γ, F_y로 나타내면 다음과 같다.

$$F_\Gamma = (E_\Gamma \cdot 2^{23} + M_\Gamma) = \frac{3}{2} \cdot 2^{23} \cdot (127 - \mu) - \frac{1}{2} \cdot (E_y \cdot 2^{23} + M_y)$$

$$= 5F3759DF_{16} - (F_y \gg 1)$$

따라서, 9번째 줄의 0x5f3759df는 단순히 보정항과 상수들을 정리한 값이다. 원본 코드에서는 이 수를 magic number라 부른다.

· 10번째 줄에서 이렇게 얻어진 값은 다시 `float` 타입으로 y의 메모리에 저장된다.

· 11번째 줄에서는 정확도를 높이기 위해, $f(y)=\frac{1}{y^2}-x$ 에 대해 뉴턴 반복법(Newton's Method)을 수행하여 $f(y)=0$을 만족하는 해 $y=\frac{1}{\sqrt{x}}$를 찾는다.

$\frac{d}{dy}f(y)\equiv f'(y)=-\frac{2}{y^3}$ 이므로, $y_{new}=y-\frac{f(y)}{f'(y)}$ 로 뉴턴 반복법을 수행하면

$$y_{new}=y-\frac{f(y)}{f'(y)}=y-\left(\frac{1}{y^2}-x\right)/(-\frac{2}{y^3})=y\,(\frac{3}{2}-\frac{x}{2}y^2)$$

으로 inverse square root로 수렴하는 해 y를 구할 수 있다.

뉴턴 반복법의 적용은 오차율을 탁월하게 낮춘다. 예시 입력값 0.01에 대해, 단 한번의 뉴턴 반복법만으로도 9.982522의 해를 얻어낼 수 있으며, 이는 본래의 답인 10.0과 단지 0.175%의 오차만을 포함한다. 뉴턴 반복법은 필요에 의해 추가적으로 더 사용될 수 있다.

2. 프로젝트 배경 및 계획

Inverse square root의 계산은 일반적으로 3D Graphics 분야에서 벡터를 정규화 하는데 많이 사용된다. 앞서 설명한 C언어로 작성된 FISR 알고리즘도 Quate III Arena라는 게임의 조명처리 과정에서 광원의 입사각 및 반사각을 계산하는데 사용된 것으로 유명하다. 해당 알고리즘은 float 실수를 이용하여 직접 1/sqrt(x)를 계산하는 과정보다 거의 4배 정도 더 빠르다. 한편, 노인의 낙상 방지를 위한 생활형 에어백 제품은 실생활 밀착형 제품이기에 24시간 작동 가능해야 하며, 가속도계로 얻어진 가속도 벡터의 정규화를 실시간으로 빠르게 처리하여 노인의 동작이 낙상인지 의도적인 행동인지를 구분해내야 한다. 따라서 해당 계산을 노인분들을 위한 생활형 에어백과 같이 빠른 벡터의 정규화 계산을 필요로 하는 장치에 사용한다면 power를 훨씬 적게 소모하며 빠르게 장치가 동작을 할 수 있을 것이라고 판단하였다.

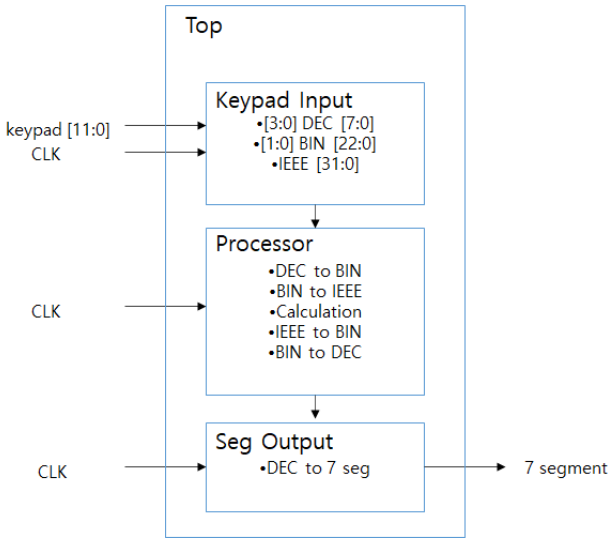


그림3 Top module schematic

전체 모듈의 구조는 위의 schematic과 같이 세 부분으로 구성되어 있다.

· Keypad Input 모듈은 FPGA보드의 keypad로부터 0~9, *, #을 입력 받아 메모리에 저장한다. 이때, *은 소수점을 표현하는 용도로 사용되었으며, #은 입력을 완료하는 ENTER키의 역할로 사용되었다.

· Processor는 FISR 연산과 이를 위해 필요한 변환들을 수행한다.

보다 구체적으로는, 입력된 값의 decimal-to-binary 변환, binary 값의 binary-to-IEEE 754 변환, 1/sqrt(x) 계산, 결과 값의 IEEE 754-to-binary 변환, binary 값의 binary-to-decimal 변환을 수행하여 필요한 계산 작업들을 행한다.

· Seg Output 모듈은 Keypad Input에 들어오는 값을 실시간으로 표시하며, Processor의 연산이 끝나는 즉시 계산된 값을 출력한다. Seg Output 모듈은 개발 시간을 단축하기 위해 강의자료의 7-segment 관련 코드를 재활용 하였으며, 다만 dot point를 추가해 소수점을 표현할 수 있도록 하였다.

따라서, FPGA보드에 입력을 시작하면 입력된 값들은 곧바로 7-segment display에 표시되고, # 키를 누르면 더 이상의 입력이 받아지지 않는다. Processor 모듈은 입력의 완료를 인지한 후 곧바로 1/sqrt(x) 연산을 수행하고, 연산된 결과는 다시 7-segment에 표시된다.

우선순위에 따라 구상한 계획은 다음과 같다. 개발 비용의 측면에 있어 Seg Output은 이미 코드의 안정성에 대한 검증이 완료되었기 때문에 큰 개발 비중을 들이지 않았고, Processor 모듈은 순수하게 논리적인 영역을

다루기 때문에 시뮬레이션만으로도 개발이 가능하여 두 번째의 우선순위를 두었다. 오히려 Keypad Input 모듈이 실제 하드웨어와 상호작용하여 입력 값을 받아 내고, 입력 값이 올바르지 않은 계산기는 계산기의 의미를 다하지 못하기 때문에 본 팀은 FPGA 보드를 사용할 수 있는 실습 시간에 Keypad Input 모듈의 개발에 가장 높은 우선순위를 두어 Input 모듈의 작동을 가장 먼저 검증하였다. 따라서 우선순위가 가장 높은 Keypad Input에 50% 정도의 개발 시간을 들이고, Processor의 개발에 40% 정도의 개발 시간을 들이기로 하였으며, 나머지 10%의 시간은 디버깅과 완성도 향상에 투자하기로 계획하였다. 개발 진행의 순서 역시 우선순위와 동일하게 진행되었다. 다만, Seg Output은 Keypad Input의 작동을 보기 위해 함께 개발되었다.

3. 과정

1) Keypad Input

Keypad input 모듈의 코드는 함께 제출한 코드 파일을 참고한다.

시뮬레이션과는 다르게 실제 모듈의 keypad로 입력을 넣는 경우, 버튼의 바운싱 문제가 생겨 여러 번의 입력이 들어간다. 따라서 디바운싱을 위해 pulse generator 모듈을 이용하여 Digital Filter Circuit을 구현해 해당 문제를 해결하였다.

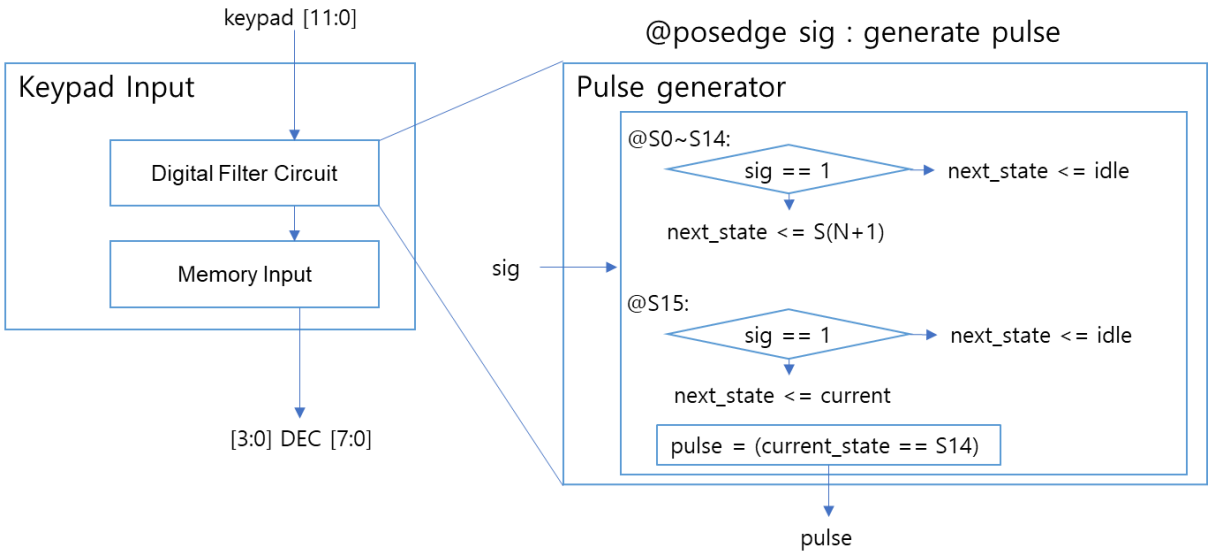


그림4 Keypad Input Schematic

Pulse generator 모듈에 신호가 입력되지 않거나, 신호가 중간에 끊긴 경우에는 next state(n_state)가 S0를 계속 유지하게 된다. 이때 만약 신호가 들어온다면 S0(idle)부터 next state가 올라가게 되고, S0부터 S14까지 신호가

계속 입력된다면 이를 하나의 지속적인 입력으로 간주하고 current state(c_state)가 S14인 경우에 pulse에 이를 할당하여 펄스 신호를 생성한다. 실제 보드의 keypad를 누르면 step function과 같은 형태로 한 번에 입력이 들어가는 것이 아니라 0과 1을 몇 번 왔다갔다하는 노이즈 이후에 입력이 들어가게 된다. 따라서 이를 통해 해당 문제를 해결할 수 있으며, 내부의 clock마다 신호가 들어가는 것이 아니라 실제 입력마다 신호가 들어가게 해준다. 실제의 Pulse generator 모듈은 flipped input을 받으므로, 올바른 작동을 위해선 port 연결을 할 때 !signal을 입력해야 한다.

출력부에서 7-segment를 작동시키는 Seg Output 모듈은 8자리의 digit을 각각 담당하는 메모리에 의해 작동되므로, 4비트의 8칸 Reg Array인 DEC 메모리 선언하여 Input으로부터 Digital Filter Circuit을 거쳐 받아낸 숫자들을 저장한다.

현재 입력하는 값은 7-segment의 가장 오른쪽 칸에 출력되며, 이전에 입력된 값은 왼쪽으로 옮겨가는 형태로 DEC 메모리를 다루었다.

또한 *을 소수점으로 정의하며, DEC 메모리에 값을 저장할 때 내부적으로는 *에 10을 할당한다. 소수점이 입력된 위치를 쉽게 파악하기 위해 *이 입력된 이후에 들어오는 입력의 개수를 세고, 해당 값을 dot에 할당한다. dot은 소수점의 위치를 파악할 때 사용된다.

해당 모듈에서 입력 값의 조건이 있는데, 이는 소수점 아래 세 자리 까지만 입력이 가능하다는 것이다. 이는 decimal to binary 변환의 편의를 위한 것으로 추후에 논하기로 한다. 따라서 입력되는 값은 크게 정수와 소수점 아래 숫자가 1개, 2개, 3개가 있는 경우만을 생각할 수 있다. segment 하나에 소수점이 통째로 들어가기 때문에 dot을 7-segment의 오른쪽 4개 자리수로 나타낸다. dot = 4'b0000은 소수점이 없음을 의미하고, dot = 4'b1000은 소수점 아래 세 자리가 있음을, dot = 4'b0100은 소수점 아래 두 자리가 있음을, dot = 4'b0010은 소수점 아래 한 자리가 있음을 의미한다. dot = 4'b0001은 소수점 아래 자릿수가 없음을 의미하므로 4'b0000과 다를 바가 없어 정의하지 않는다. 이후에 각 자리 숫자와 dot을 출력하여 계산 모듈인 Processor 모듈로 전달한다.

2) Processor

Processor 모듈의 코드는 함께 제출한 코드 파일을 참고한다.

Processor 모듈은 내부에 6개의 서브 모듈을 포함하고, 12 단계의 main state와 최대 4 단계의 substate로 구성되어 있다.

Processor 를 본격적으로 설명하기에 앞서, 연산에 필요한 변수들을 연산 순서에 맞춰 표현한 Pseudo Code와 각 단계를 도식으로 표현한 Schematic을 제시한다.

```
float Q_rsqrt( float number ){
    long i;
    float x2, y;
    const float threehalfss = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y;
    // transform from BIN to IEEE754
    i = 0x5f3759df - ( i >> 1 );
    // shift bits and subtract as bits
    y = * ( float * ) &i; // ignore
    y = y * ( threehalfss - ( x2 * y * y ) );
    // mul and sub as IEEE754(float32)
    return y; // transform from IEEE754 to BIN
}
```

```
Processor {
    reg [31:0] number (IEEE) <= input (BIN)
    reg [31:0] half (IEEE) <= 0.5 (BIN)
    reg [31:0] threehalfss (IEEE) <= 1.5 (BIN)
    reg [31:0] x2 <= number * half (float32_mul)
    reg [31:0] y <= 0x5f3759df - number >> 1
    reg [31:0] x2y <= x2 * y (float32_mul)
    reg [31:0] x2yy <= x2y * y (float32_mul)
    reg [31:0] newton_iter <= threehalfs - x2yy
    (float32_add)

    reg [31:0] result <= newton_iter * y
    (float32_mul)
    }
```

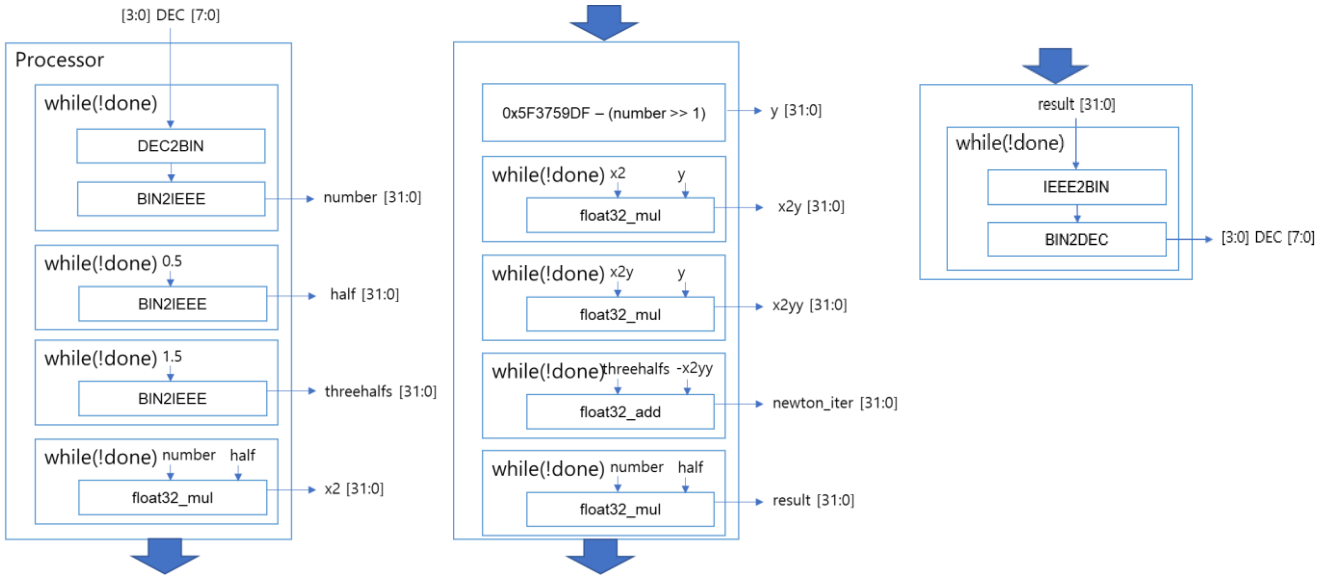


그림5 Processor Schematic

각각의 서브 모듈은 Finite State Machine 형태로 작동하며, 작동이 완료되면 return 값과 done 신호를 동시에 출력하고 다시 idle state로 돌아간다. 서브 모듈을 제어하는 Processor 모듈은 이를 이용해 enable 신호와 parameter 를 인가하여 서브 모듈을 작동시킨 후, done 신호가 출력될 때 까지 기다린다. Processor 모듈이 서브 모듈의 done 신호를 catch 하면 return 값을 reg에 저장하는 방식으로 서브 모듈을 제어한다. 이와 같은 방식으로 Processor 모듈은 각 단계를 순차적으로 실행하며, 제한된 circuit을 여러 번 재사용 하여 회로의 면적을 줄인다.

Processor 모듈은 기능적으로 크게 세 부분으로 나누어져 있다. 먼저 input 모듈에서 받은 값을 IEEE 754로 변환하는 부분, 이를 이용하여 1/sqrt(x)를 계산하는 부분, 그리고 계산된 값을 다시 십진수로 변환하는 부분이 있다.

(1) DEC2BIN, BIN2IEEE (Tricks in DEC2BIN)

앞서 각 자리에 해당하는 값들과 소수점의 위치를 입력으로 받아왔다. 이를 이용하여 decimal-to-binary (DEC2BIN) 에서는 정수부, 소수부로 먼저 나눠서 변환을 진행한다. 정수부의 경우 각 자리 마다 10^{n-1}을 곱하고 더하는 방식으로 하나의 큰 정수를 만들고, 이를 다시 이진수로 변환하여 계산하였다. 정수의 이진수 변환은 Verilog의 자체 Syntax으로 큰 노력 없이 쉽게 가능하였다. 반면에 소수부의 경우에는 이진수로 변환한 값을 얻기 위해서는 반복적으로 나눗셈 연산을 수행해줘야 하지만 이는 cost가 너무 크기 때문에 다른 방법으로 유사한 결과를 얻을 수 있도록 구현하였다. 앞서 소수점 아래 세 자리로 입력에 제한을 뒀는데, 해당 세 자리 값을 1000이 아닌 1024로 나누면 정확한 값은 아니지만 소수점 아래 두 자리까지 매우 근사한 값을 얻을 수 있다. 이때 이진수에서 1024로 나누는 계산은 shifting과 동일하므로, 소수부는 소수점 첫째자리에 100, 둘째자리에 10, 셋째자리에 1을 곱한 이후에 10칸 shifting 하여 반복적인 나눗셈 계산 알고리즘 없이 효율적으로 이진 변환 값을 계산한다.

계산한 정수부와 소수부를 각각 23bit 크기의 BIN1과 BIN0 메모리에 저장한다. BIN1에는 우측 정렬되어 값이 들어가고, BIN0에는 좌측 정렬로 값을 저장한다. 정수부가 있는 경우와 없는 경우로 나누어 binary-to-IEEE 754 (BIN2IEEE) 의 계산 과정을 따라가서 sign, exponent, mantissa를 각각 구한다. 이때 소수점 왼쪽에 1만 남게 shift하는 과정을 정수부가 있는 경우에는 BIN1의 23번째 비트, 정수부가 없는 경우에는 BIN0의 23번째 비트가 1이 올때까지 shift를 시켜서 shift한 횟수를 세고, 정수부를 shift한 경우에는 해당 값이 아닌 23에서 해당 값을 뺀 숫자가 원래 지수이므로 이를 계산 과정에서 추가한다.

(2) float32_mul, float32_add (Round to nearest Even Rule)

앞의 BIN2IEEE로부터 주어진 입력을 IEEE 754 표준 형식으로 변수 number로 저장한다. 그 외에도 알고리즘을 수행하는데 필요한 두 상수 0.5와 1.5를 BIN2IEEE를 재사용하여 마찬가지로 IEEE 754 표준 형식으로 변수 half와 threehalfs에 저장한다.

이후의 계산 과정에서는 `y * ( threehalfs - ( x2 * y * y ) )` 를 계산하기 위해 float32 곱셈기와 float32 뺄셈기(덧셈기)가 필요하다.

float32_mul과 float32_add는 개발 시간의 단축을 위해 Reference를 참고하였으며, 이를 본 팀의 목적에 맞게

변형하여 사용하였다.

float32_mul은 13가지 단계를 통해 앞서 사전지식에서 설명한 IEEE 754 표준의 곱셈 알고리즘을 FSM으로 작동시킨다.

EN 신호를 enable 트리거로 받아 작동을 시작하며, 연산이 끝나면 output과 DONE 신호를 반환하고 다시 idle state로 복귀한다.

한편 본 코드에서는 rounding 과정을 특히 주목할 필요가 있다. round state 단계에서 인접 짝수 모드 근사(Round to nearest even)를 규칙으로 반올림을 수행하는데, Guard bit와 Round bit, Sticky bit를 이용해 이를 구현하였다. 이때 Guard bit란 LSB(최소 비트)로 근사 결과값을 최소 유효값을 의미하고, Round bit란 Guard bit 오른쪽의 근사를 통해 없어지는 첫 번째 비트를 의미하며, Sticky bit란 Round bit 오른쪽의 나머지 비트들을 OR 연산한 값을 의미한다.

그림6 Guard bit, Round bit, and Sticky bit

Round to nearest even에서는 근사할 값이 두 개의 가능한 값의 중간값인 경우 최소 유효값(LSB)가 짝수(=0)이 되도록 근사하고, 그 외의 경우에는 일반적인 반올림 방법을 따른다. 이를 예시를 통해 네 가지 상황으로 정리하면 다음과 같다.

Ex) 1/4 자리(소수점 아래 2번째)로 근사하는 경우

근사할 값	근사 결과	설명	반올림
10.00011 (= 2+3/32)	10.00 (=2)	근사하는 자리 오른쪽 비트 011 < 중간값 100	내림
10.00110 (= 2+3/16)	10.01 (=2+1/4)	근사하는 자리 오른쪽 비트 110 > 중간값 100	올림
10.11100 (= 2+7/8)	11.00 (=3)	근사하는 자리 오른쪽 비트 100 = 중간값 100, 근사하는 자리 비트가 1(홀수) --> 올림하여 짝수로 만든다.	올림
10.10100 (=2+5/8)	10.10 (=2+1/2)	근사하는 자리 오른쪽 비트 100 = 중간값 100, 근사하는 자리 비트가 0(짝수) --> 이미 짝수이므로 내림한다.	내림

본 코드에서는 내림을 구현할 필요가 없으므로(유효 숫자 아래의 값을 전부 버리면 된다), 올림의 조건만을 판단하여 올림을 구현하였다.

근사하는 자리 오른쪽 비트(Guard)의 비트가 1이고 Round 혹은 Sticky 비트가 존재한다면 이는 중간값보다 큰 값임을 의미하므로 올림의 조건이 된다. 만약, Guard가 1이지만 Round 와 Sticky 비트가 0 이어서 중간값과 같은 값이더라도 근사하는 자리 비트(mantissa의 LSB)가 1 이라면 이는 역시 올림의 조건이 된다.

이를 구현한 코드는 다음과 같다.

```
round:
begin
  if (guard && (round_bit | sticky | z_man[0])) begin
    z_man <= z_man + 1;

    if (z_man == 24'hffffff) z_exp <= z_exp + 1;
  end
  state <= pack;
end
```

float32_add 역시 float32_mul과 비슷한 방식으로 사전설명의 덧셈 알고리즘을 따라 작동하며, 동일한 rounding 규칙을 따른다.

(3) IEEE2BIN, BIN2DEC

이제 계산된 값을 다시 십진수로 바꿔서 7-segment output으로 전달한다. IEEE 754를 이진수로 전환하는 과정은 앞서 이진수를 IEEE 754로 바꾸는 역과정으로 코드를 작성하였다. 값이 0인 zero case와 정수부가 있는 경우, 없는 경우로 나누어 똑같이 정수부는 23 bit인 BIN1, 소수부는 BIN0으로 저장하여 이진수를 십진수로 전환하는 모듈로 넘겨준다.

처음에 모듈 입력이 최소값이 0.001이므로 결과로 나올 수 있는 값의 정수부가 최대 두 자리 수인 것을 이용하여 십진수 변환 모듈을 작성하였다. 정수부는 각 자리별로 2^{n-1}을 곱하여 더하는 방식으로 계산이 가능하지만, 소수부의 계산은 소수를 곱하는 것이 안되므로 0.5 대신 $5x10^{8}$, 0.25대신 $25x10^{7}$, … 를 각 비트마다 곱하여 더한 후에 각 자리 값을 구한다. 이때 해당 부분에서는 다른 방식으로 각 자리 수를 구하지 못하여 / operator와 % operator를 사용하여 값을 계산하였다. 이때 얻은 값을 segment output 모듈로 전달한다.

3) Seg Output

Seg output 모듈의 코드는 함께 제출한 코드 파일을 참고한다

해당 모듈은 6주차 실험에서 사용한 7-segment 모듈과 거의 동일하다. 이때 수업시간과는 다르게 dp에 해당하는 값을 추가하였고 8칸의 display를 모두 사용하도록 코드를 수정하였다. 처음에 keypad를 이용하여 값을 입력하는 경우에도 입력된 값을 segment output 모듈로 전달하여 7-segment에서 입력된 값을 직접 볼 수 있게 하며, 계산이 종료된 이후에 계산된 값을 마찬가지로 해당 모듈로 전달하여 계산된 값을 볼 수 있다.

4. 결과

Vivado/ModelSim을 통한 시뮬레이션을 통해, 두 종류의 시뮬레이션 Tool에서 Verilog의 합성이 잘 진행되고, 올바른 결과가 나옴을 확인할 수 있었다. 그러나, 실제 FPGA 보드에서는 작동이 특정 state에 갇혀 다음 단계로 넘어가지 못함을 확인할 수 있었고, 이는 서브 모듈이 메인 모듈에게 정해진 CLK period 내에 신호를 보내지 못해 생기는 문제임을 확인하였다. 본 팀은 이러한 과정의 디버깅을 위해 코드의 각 단계마다 flag를 삽입하여, 특정 단계마다 정해진 LED를 밝히도록 하여 메인 모듈 혹은 서브 모듈이 어느 단계에 머물러 있는지를 판별하였다.

우선 시뮬레이션을 통해 계산된 출력은 다음과 같다.

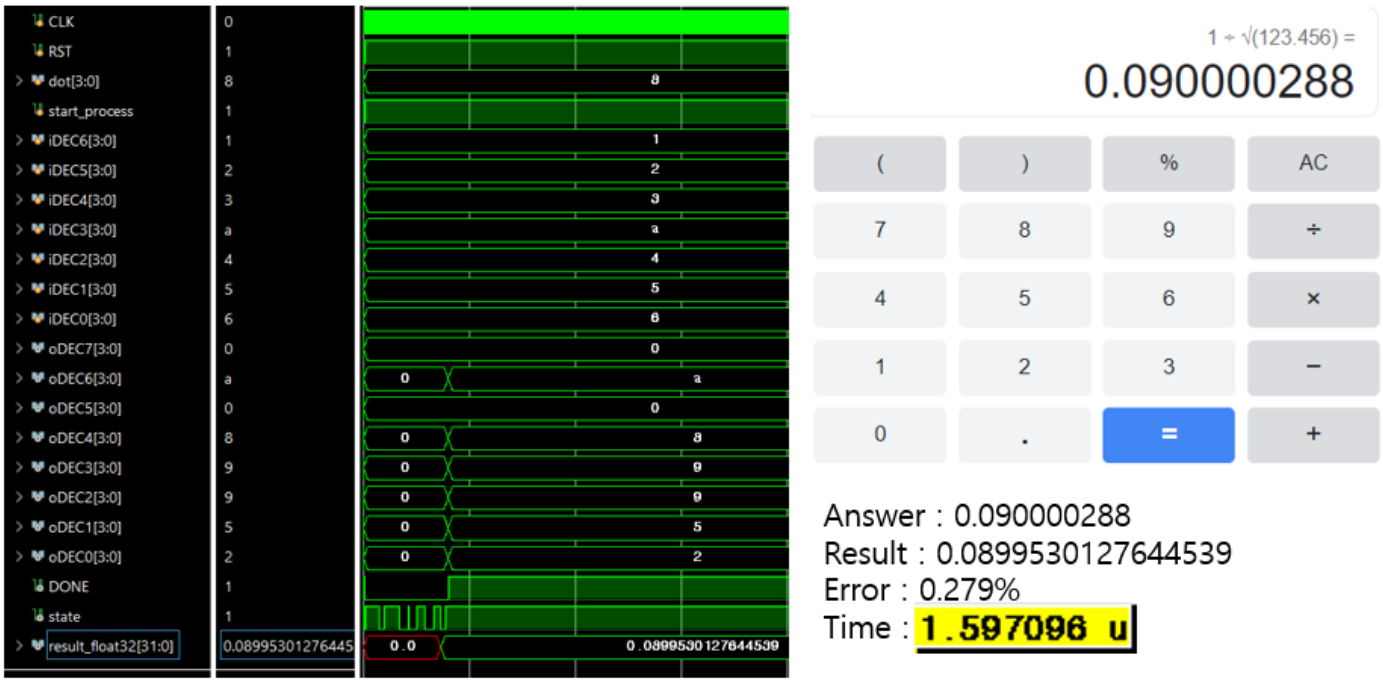
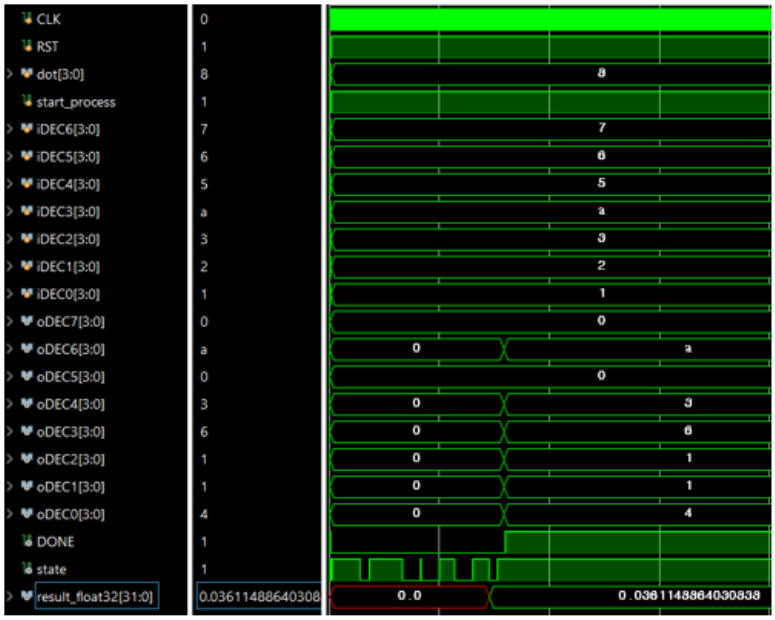


그림7 입력값 123.456의 경우



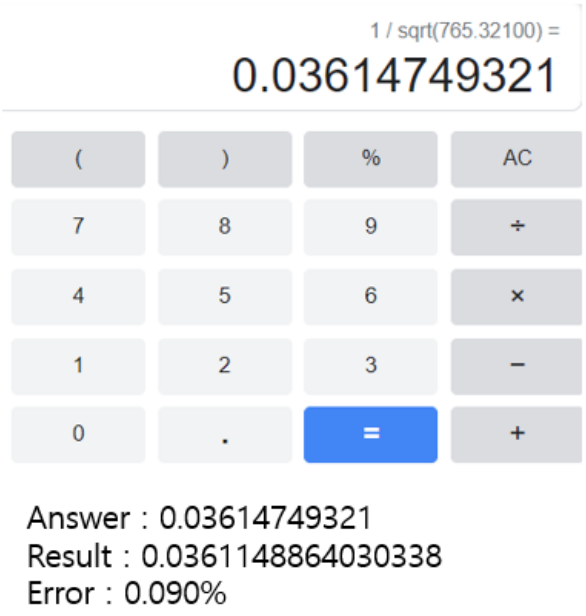


그림 8 입력값 765.321의 경우

입력 123.456에 대해 정답과 0.279%로 근소한 차를 갖는 결과를 출력하였으며, 이때 연산 시간은 10ns CLK period를 기준으로 1.59 us가 걸렸다.

더 큰 입력인 765.321에 대해서는 정답과 0.090%로 훨씬 더 작은 차이를 갖는 결과를 출력하였으며, 이때 연산 시간은 동일하게 소모되었다.

입력이 작을수록 오차가 커지는 이유는 DEC2BIN과 IEEE 754의 부동소수점 표기 방식 때문이다. DEC2BIN은 십진수 소수를 이진수로 변환하기 위해 1000을 곱하고 이를 다시 1024로 나눈다. 때문에, 소수점 두 자리 이하로 필연적인 오차가 생성되게 된다. 만일 입력되는 값이 더 작아진다면 소수점의 오차는 본래 값에 비해 더 큰 비중을 차지하게 될 것이다(입력되는 값은 그대로이나 오차가 생기는 자리 수 위치는 동일하므로). 게다가 IEEE 754는 부동소수점 표기를 하기 때문에 큰 수와 작은 수를 동일한 Mantissa 비트로 표현한다. 이를 비유적으로 설명하자면 1/10에서의 1과 1/100000에서의 1이 연산 중에 동일한 비중의 값으로 취급된다고 설명할 수 있다. 따라서 더 작은 값을 입력할수록 입력된 값에 비해 큰 오차가 생성되고, 이러한 값이 IEEE 754를 통해 실제 값에 비해 더 큰 비중으로 연산 과정으로 넘겨진다. 또한 연산 과정인 1/sqrt(x) 역시 x가 클수록 더 0에 가까운 값을 갖기에 더 큰 입력에 대해 결과값과 참값이 둘 다 0에 가까워져 결과값과 참값이 차이가 줄어들게 되는 것도 또다른 이유로 들 수 있다. 그러므로, 본 계산기는 더 큰 입력에 대해 보다 더 정확한 결과를 갖는다. 그러나, 본래의 취지인 생활형 에어백의 가속도 벡터 정규화 계산에서는 갑작스런 큰 가속도 변화를 겪는 상황의 연산이 중요하므로, 이러한 오차는 DEC2BIN의 연산 속도를 향상시킨 대가를 감안하여

Trade-Off 될 수 있는 것으로 보인다.

실제 보드에서는 값이 입력되는 것까지는 확인이 되지만, processor에서 DEC2BIN 모듈에서 BIN2IEEE 모듈로 신호가 전달이 되지 않는 것을 state를 LED에 표시하는 방법을 통해 확인하였다. 앞서 DEC2BIN 모듈 내에서도 신호가 전달이 잘 되지 않는 것을 확인했는데, 이때 신호를 전달하기 전에 state를 추가하여 신호 길이를 늘려주는 등의 방법을 통해 앞선 문제를 해결하였다. 그러나 이후의 문제에 대해서는 시간 내에 해결을 하지 못하였다.

5. 한계 및 추후 발전 방향

가장 큰 문제는 역시 실제 보드에서 작동이 잘 되지 않는다는 것이다. 앞서 언급한 것과 같이 CLK 주기 내에 신호를 인식하지 못하여 발생한 문제로 파악되며, 추후에 연구가 가능하다면 해결이 가능할 것으로 사료된다.

또 다른 문제로는 십진수를 이진수로 변환하는 과정에 있어서 1000으로 나누는 것이 아닌 shifting을 통해 cost와 정확도의 Trade-Off로 인해 약간의 오차가 발생한다는 문제가 있고, 다시 이진수를 십진수로 변환하는 과정에서 / operator와 % operator를 사용하여 cost가 높아진 것을 확인할 수 있다. 후자의 경우 shift and add 3 알고리즘을 통해 코드의 수정이 가능하며, 이를 통해 cost를 낮출 수 있다.

추후 발전 요소로는 다음을 고려할 수 있다.

1. 소수점 입력 범위 확장

현재 설계한 모듈의 경우 입력을 소수점 아래 세자리만 받을 수 있으며, dot point를 포함하여 최대 8자리 숫자만 받을 수 있게 설계가 되어있다. 따라서 이러한 점을 수정하여 입력이 가능한 범위를 더 넓혀서 설계할 수 있을 것이다.

2. 소수점 디스플레이 방식 변경

현재는 소수점이 DEC 메모리 한 칸을 온전히 차지하여, 세그먼트 디스플레이 상으로도 소수점이 segment 한 칸을 차지하게 된다. 그러나, 연산에 사용하는 DEC 메모리와 실제 디스플레이에 사용되는 메모리를 구분하여 사용한다면, 소수점을 앞 자리의 숫자 뒤에 붙여 표현하는 것도 가능할 것이다.

3. 파이프라인 설계

현재 설계된 코드는 순차적으로 각 서브 모듈을 켜고, 반환값을 받고, 서브 모듈을 끄는 방식으로 작동한다. 이러한 방식은 구현이 편리하나, 반환값을 기다리는 동안 메인 모듈이 계속 Busy waiting을 한다는 단점이 있다. 따라서 반복 사용되는 BIN2IEEE 모듈과 float32_mul 모듈을 파이프라인 설계하여 훨씬 더 빠른 시간 내에 연산이

완료되도록 할 수 있을 것이다.

4. DEC2BIN, BIN2DEC 최적화

앞서 언급하였듯이 DEC2BIN과 BIN2DEC에 연산 속도를 위해 희생한 부분과, / 및 % 연산자의 사용으로 최적화되지 못한 부분이 존재하므로 이를 개선하여 보다 최적화된 보드의 동작을 기대할 수 있을 것이다.

토의	● 2주간 2000 라인에 달하는 볼륨의 프로젝트를 통해, 프로젝트의 진행 계획과 시간 배분에 대해 고찰하고 계획을 세워 실행해 볼 수 있었다. ● IEEE 754 표준을 이해하였으며, float32의 덧셈과 곱셈, 반올림에 대해 학습하였다. ● FISR 알고리즘을 이해하고 이를 Verilog로 구현할 수 있었다. ● 디버깅이 어려운 실제 하드웨어 환경에서의 문제 해결을 경험할 수 있었다.

References

- https://en.wikipedia.org/wiki/Fast_inverse_square_root
- https://github.com/dawsonjon/fpu
- SevenSegentController - DIGITAL SYSTEM DESIGN AND LABORATORY
- https://nybounce.wordpress.com/2016/06/24/
- https://roadtodeveloper.tistory.com/32