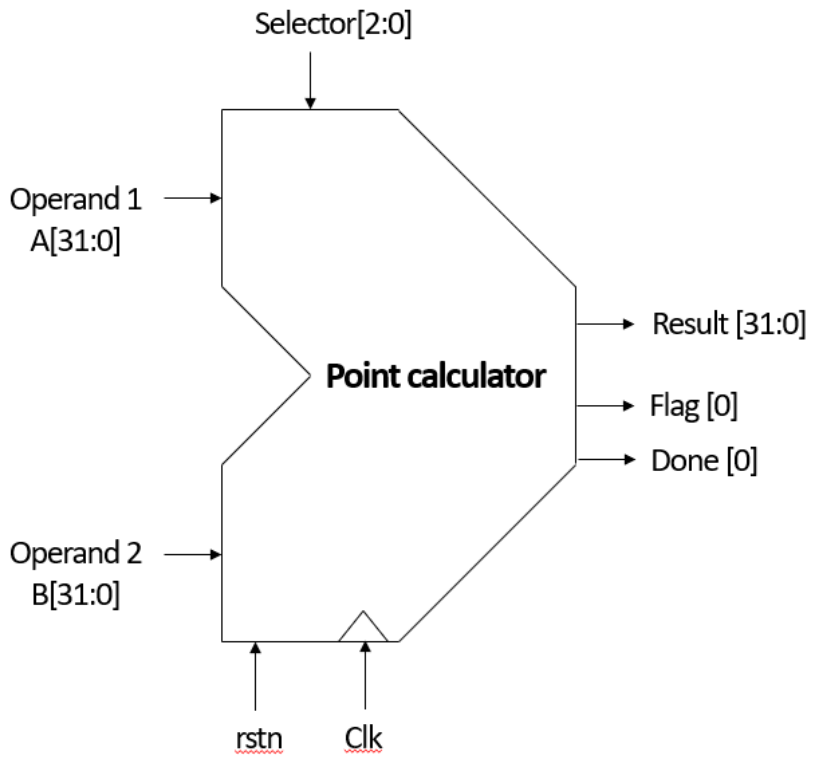
디지털시스템설계 LAB3

2016310936 우승민

LAB3 의 목표는 아래의 **Point calculator** 를 구현하는 것입니다.



여기서 selector 는 3 비트로 아래의 총 8 가지 중 원하는 계산을 골라줍니다.

Selector[2:0]	Operation of point calculator
000	Fixed-point A + Fixed-point B
001	Fixed-point A * Fixed-point B
010	Convert fixed-point A to floating point
011	Return large number between fixed-point A and B
100	Floating-point A + Floating-point B
101	Floating-point A * Floating-point B
110	Convert floating-point A to fixed point
111	Return large number between floating-point A & B

Selector 로 선택한 operation 의 operand1 과 operand2 의 올바른 결과를 출력하는 것이 이번 과제 목표입니다. 만약 계산 결과에서 overflow / underflow 가 일어날 경우에는 결과값은 상관하지 않고 flag bit 를 set 해 줍니다.

```
always @(posedge rstn) begin
    fix1 <= 0;
    fix2 <= 0;
    sign <= 0;
    flo1 <= 0;
    flo2 <= 0;
    mul1 <= 0;
    n <= 0;
    i <= 0;
    x <= 0;
    y <= 0;
    done <=0;
    flag <=0;
    result <=0;
end

always @(posedge clk) begin
    if(rstn == 1) begin
      fix1 = 0;
      fix2 = 0;
      sign = 0;
      flo1 = 0;
      flo2 = 0;
      mul1 = 0;
      result = 0;
      case(selector)
```

우선 rstn 값이 set 이 되어야 동작을 하기 때문에 rstn 값이 처음 set 되면 사용하는 모든 register 값들을 초기화 해주고, 그 이후 clock 이 positive edge 일 때 값들을 다시 초기화 한 후 selector 의 값에 따라 case 문을 실행하게 해주었습니다.

이제 selector 의 값이 000 ~ 011 일 때 계산해주는 fixed point 를 보면 최상위 비트는 sign bit 를 뜻하고 하위 16 비트는 소수점을 나타내고 나머지 16 부터 30 까지 15bit 는 정수부분을 표현합니다.

여기서 fixed point 는 2's complement 입니다. 따라서 범위는 (-2^15 ~ 2^15-2^(-16))입니다.

➢ Fixed point[31:0]

Selector 가 3'b000 인 addition 의 경우부터 설명하겠습니다. Fixed point 는 2's complement 이기 때문에 operand1, 2 의 sign bit 값에 따라 3 가지 경우로 나누었습니다.

1. Positive + Positive
2. Negative + Negative
3. Positive + Negative

```
3'b000 : begin
  flag = 0;
  if(operand1[31:31] == operand2[31:31]) begin
    if(operand1[31:31] == 0) begin // positive + positive
      sign = operand1[31:31];
      fix1 = operand1[30:0] + operand2[30:0];
      if(fix1[31:31] == 1) flag = 1;
      result[31:31] = sign;
      result[30:0] = fix1[30:0];
    end
    else begin // negative + negative
      sign = operand1[31:31];
      fix1 = ~operand1 + 1;
      fix2 = ~operand2 + 1;
      fix1 = fix1 + fix2;
      if(fix1[31:31] == 1) flag = 1;
      else result[30:0] = fix1[30:0];
      result = ~result + 1;
    end
  end
  else begin  // positive + negative
    if(operand1[31:31] == 0) begin
      fix1 = operand1;
      fix2 = ~operand2 + 1;
    end
    else begin
      fix1 = operand2;
      fix2 = ~operand1 + 1;
    end
    if(fix1[31:0] >= fix2[31:0]) begin
      result[30:0] = fix1[30:0] - fix2[30:0];
      result[31:31] = 0;
    end
    else begin
      result[31:0] = fix2[31:0] - fix1[31:0];
      result = ~result + 1;
    end
  end
  done <= 1;
end
```

positive + positive 일 경우에는 operand1, 2 의 [30:0] bit 를 단순히 더해주었습니다. 만약 31 개의 bit 2 개를 더했는데 32bit 가 set 되면 범위를 벗어난 것이므로 flag 를 set 해주었습니다.

Negative + negative 경우는 operand1, 2 를 2's complement 의 방식으로 양수로 바꾼 후 같은 방식으로 해준 후 마지막에 다시 음수로 바꾸어 주었습니다.

Positive + negative 경우는 절댓값이 반드시 작아지므로 flag bit 는 건드릴 필요가 없고 negative 인 값만 양수로 바꾸어 준 후 빼 주었습니다. 만약 절댓값이 양수가 큰 경우에는 그대로 값을 출력하였고, 음수가 큰 경우에는 뒤집어서 출력하였습니다.

Selector 가 3'b001 인 multiplication 의 경우도 마찬가지로 operand1, 2 의 sign bit 값에 따라 3 가지 경우로 나누었습니다.

1. Positive * Positive
2. Negative * Negative
3. Positive * Negative

Fixed point 2 개를 곱하는 것은 integer 2 개를 곱한 후 2^32 을 나누어 준 것과 같습니다. 다만 결과 값이 63bit 로 소수는 하위 32bit 이고, 정수는 상위 31bit 인데 표현 가능한 범위에 따라 나누어 주면 소수는 [32:17] 정수는 [46:18] 입니다.

```
3'b001 : begin
  flag = 0;
  sign = operand1[31:31] ^ operand2[31:31];
  if(operand1[31:31] == operand2[31:31]) begin
    if(operand1[31:31] == 0) begin // positive * positive
      mul1[63:0] = operand1[30:0] * operand2[30:0];
      result[31:31] = sign;
      if(mul1[63:47] !== 17'b0) flag = 1;
      result[30:0] = mul1[46:16];
      if(mul1[15:15] == 1) begin
        result = result + 1;
        if(result[31:31] != sign) flag = 1;
      end
    end
    else begin // negative * negative
      fix1 = ~operand1 + 1;
      fix2 = ~operand2 + 1;
      mul1[63:0] = fix1 * fix2;
      result[31:31] = sign;
      if(mul1[63:47] !== 17'b0) flag = 1;
      result[30:0] = mul1[46:16];
      if(mul1[15:15] == 1) begin
        result = result + 1;
        if(result[31:31] != sign) flag = 1;
      end
    end
  end
  else begin  // positive * negative
    if(operand1[31:31] ==1) fix1 = ~operand1 + 1;
    else fix1 = operand1;
    if(operand2[31:31] ==1) fix2 = ~operand2 + 1;
    else fix2 = operand2;
    mul1[63:0] = fix1 * fix2;
    if(mul1[63:48] !== 16'b0) flag = 1;
    result[31:0] = mul1[47:16];
    if(mul1[15:15] == 1) begin
      result = result + 1;
      if(result[31:31] != 0) flag = 1;
    end
    result = ~result + 1;
  end
  done <=1;
end
```

우선 result 의 sign bit 는 각 operand 의 sign bit 의 xor 를 통해 구해주었습니다.

Fixed point 를 곱한 후 만약 48bit 이상의 값이 0 이 아니면 fixed point 의 표현 범위를 초과한 것이므로 flag 를 set 해주었습니다.

그리고 16bit 가 set 되어있으면 rounding up 을 해주었습니다.

이 후 negative 의 operand 는 addition 과 마찬가지로 2's complement 를 해준 후 계산해 주었습니다.

Selector 가 3'b010 인 경우는 fixed point -> floating point 로 변환하는 연산입니다. 우선 float point 의 표현 범위가 fixed point 보다 훨씬 크기 때문에 flag 가 발생할 일은 없습니다. 또한 생각할 점은 0 을 제외한 모든 fixed point 는 float point 의 normal number 라는 점입니다.

```
3'b010 : begin
  sign = operand1[31:31];
  flag = 0;
  if(operand1[30:0] == 31'b0) flo1 = 0; // 0 -> 0
```

0 을 제외한 숫자는 exponent 와 frac part 를 구하는 2 가지 과정을 거치게 하였습니다.

```
else begin
  if(sign ==0) begin
  fix1[14:0] = operand1[30:16];
  fix2[15:0] = operand1[15:0];
  end
  else begin
  fix1 = ~operand1 + 1;
  fix2[15:0] = fix1[15:0];
  fix1[14:0] = fix1[30:16];
  fix1[30:16] = 15'b0;
  end
  if(fix1) begin
    n = 14;
    while(fix1[14:14]!=1) begin
      n = n-1;
      fix1 = fix1 << 1;
    end
  end
  else begin
    n = -1;
    while(fix2[15:15] != 1) begin
      n = n -1;
      fix2 = fix2 << 1;
    end
  end
```

이 부분은 exponent 를 구하는 과정입니다. 먼저 operand1 이 정수부분을 가지고 있는지 확인하고 있으면 정수부분 중 최대 bit 위치를 확인하고 n 에 exp 를 저장합니다.

0 의 경우는 이미 위에서 따로 해주었기 때문에 만약 정수부분이 없으면 소수부분에 반드시 최소 1 개의 bit 가 1 이 되어있을 것입니다. while 문을 통해 가장 큰 bit 의 위치를 찾고 n 에 exp 를 저장합니다.

```
    flo1[30:23] = 8'b01111111;
    if(n >= 0) begin
      for(i=0; i<n; i= i+1) flo1[30:23]= flo1[30:23] +1;
      if(sign==0) flo2[30:0] = operand1[30:0];
      else flo2[30:0] = ~operand1 + 1;
      while(flo2[30:30] != 1) flo2 = flo2 << 1;
      flo2 = flo2 << 1;
      flo1[22:0] = flo2[30:8];
    end
    else begin
      for(i=0; i>n; i= i-1) flo1[30:23]= flo1[30:23] -1;
      if(sign==0) flo2[30:0] = operand1[30:0];
      else flo2[30:0] = ~operand1 + 1;
      while(flo2[30:30] != 1) flo2 = flo2 << 1;
      flo2 = flo2 << 1;
      flo1[22:0] = flo2[30:8];
    end
  end
  result[31:31] = sign;
  result[30:0] = flo1[30:0];
  done <=1;
end
```

Exponent 는 0 인 8'b01111111 을 기준으로 위에서 구한 n 만큼을 더하여 구할 수 있습니다.

그 후 frac part 는 Operand1 의 sign bit 를 제외한 31bit 를 가져와 1.xxxx * 2^exp 중 1 에 해당하는 가장 큰 bit 를 while 문을 통해 구합니다.

이후 소수부분만 float 에 저장해 주어야 하기 때문에 left shift 를 1 번 더 해주고 frac part 를 저장합니다.

Selector 가 3'b011 인 경우는 단순히 크기 비교만 해주면 됩니다.

```
3'b011 : begin
  if(operand1[31:31] == operand2[31:31]) begin
  result <= (operand1 >= operand2)? operand1 : operand2;
  end
  if((operand1[31:31] == 0) && (operand2[31:31] ==1)) begin
  result <= operand1;
  end
  if((operand1[31:31] == 1) && (operand2[31:31] ==0)) begin
  result <= operand2;
  end
  flag <= 0;
  done <=1;
end
```

다음으로 floating point 의 경우를 설명하겠습니다.

➢ Floating point[31:0] : IEEE 754 single precision

Sign bit S[31]

Exponent E[30:23] **Fraction F[22:0]**

Exponent	Fraction	Object
0	0	0
0	Nonzero	Denormal number
1~254	Anything	Normal number
255	0	+/- ∞
255	Nonzero	NaN

➢ Normal interpretation : $(-1)^S x(1+0.F)x2^{E-127}$

➢ Denormal interpretation : $(-1)^S x(0.F)x2^{-126}$

Floating point 는 위의 표현처럼 지수부분(exponent)와 소수부분(fraction)으로 구성되어 있기에 두 부분을 따로 구해주어야 합니다. 또한 표처럼 5 가지 경우에 따라 형식이 다르기 때문에 주의해 주어야합니다.

Selector 가 3'b101 인 floating point 의 addition 은 exponent 크기에 따라 나누었습니다.

나누는 기준은 exponent1 과 exponent2 의 크기 비교를 한 후 denormal 과 normal 인 경우에 따라 나누어주었습니다.

```
3'b100 : begin
  if(operand1[31:31] == operand2[31:31]) begin // addition
    if(operand1[30:23] >= operand2[30:23]) begin // operand1 exp >= operand2 exp
      sign = operand1[31:31];
      if(operand1[30:23] == 8'b11111111) flo2[30:0] = operand1[30:0];
      if(operand2[30:23] == 8'b0) begin
        if(operand2[30:0] == 31'b0) flo2[30:0] = operand1[30:0];
```

여기서 operand2 가 0 이 아니면 denormal 이고, operand1 과 더하는 과정은 아래와 같습니다.

```
else begin
  if(operand1[30:23] == 8'b0) begin // denormal + denormal
    flo1[23:0] = operand1[22:0] + operand2[22:0];
    flo2[22:0] = flo1[22:0];
    if(flo1[23:23] == 1) flo2[23:23] = 1;
  end
  else begin // normal + denormal
    n = operand1[30:23] - operand2[30:23];
    flo1[22:0] = operand2[22:0];
    for(i=0; i<n-1; i= i+1) flo1 = flo1 >> 1;
    if(flo1[0:0] == 1) flo1 = flo1+1;
    flo1 = flo1 >> 1;
    flo2[23:0] = operand1[22:0] + flo1[22:0];
    if(flo2[23:23] == 1) begin
      flo2[23:23] = 0;
      if(flo2[0:0] == 1) flo2 = flo2 + 1;
      flo2 = flo2 >> 1;
      flo2[31:23] = operand1[31:23] +1;
    end
    else flo2[31:23] = operand1[31:23];
  end
end
```

operand1 도 denormal 이면 fraction 을 그대로 더해주고 만약 0.xxx + 0.xxx 에서 1.xxx 가 되면 normal 형태로 바꾸기 위해 exponent 값을 1 로 해주었습니다.

Operand1 이 normal 이면 exponent 의 차이를 구하고 그 만큼 operand2 의 fraction 을 right shift 한 후 더해주었습니다. 주의할 것은 마지막에 rounding off 를 해주기 위해 for 문을 한번 덜 하고 rounding 후 다시 한번만 shift right 하는 것입니다.

그 후 fraction part 를 더하고 1.xxxx + 0.xxxx 에서 10.xxxx 가 될 경우에는 exponent 값을 1 더해주고 fraction 을 shift right 합니다.

```
  else begin  // normal + normal
    n = operand1[30:23] - operand2[30:23];
    flo1[22:0] = operand2[22:0];
    if(n!=0) begin
      flo1[23:23] = 1;
      for(i=0; i<n-1; i= i+1) flo1 = flo1 >> 1;
      if(flo1[0:0] == 1) flo1 = flo1+1;
      flo1 = flo1 >> 1;
      flo2[23:0] = operand1[22:0] + flo1[22:0];
      if(flo2[23:23] == 1) begin
        flo2[23:23] = 0;
        if(flo2[0:0] == 1) flo2 = flo2 + 1;
        flo2 = flo2 >> 1;
        flo2[31:23] = operand1[31:23] +1;
      end
      else flo2[31:23] = operand1[31:23];
    end
    else begin
      flo1[23:23] = 1;
      flo2[22:0] = operand1[22:0];
      flo2[23:23] = 1;
      flo2[24:0] = flo1[23:0] + flo2[23:0];
      flo2 = flo2 >> 1;
      flo2[31:23] = operand1[31:23] + 1;
    end
  end
end
```

다음으로 operand2 가 normal 인 경우에는 현재 exponent1 을 exponent2 의 이상으로 두었기 때문에 operand1 도 normal 입니다.

Code 의 대부분은 normal + denormal 과 비슷하지만 여기서는 exponent 값이 같을 때와 다를 때 나누어주었습니다.

또한 denormal 과 달리 operand2 의 fraction 이 1.xxxx 이기 때문에 24bit 에 1 을 넣어준 후 계산하였습니다.

Exponent2 가 exponent1 보다 큰 경우에는 반대로만 해주고 똑같기 때문에 생략하겠습니다.

```
  if(flo2[30:23] == 8'b11111111) flag = 1;
  else flag = 0;
  result[31:31] = sign;
  result[30:0] = flo2[30:0];
end
```

이후 result 에 값을 넣을 때 만약 floating point 의 범위를 넘어서 exponent 가 8'b11111111 이 되면 flag 값을 set 해 주었습니다.

다음은 둘의 sign bit 가 다를 때의 addition 을 설명하겠습니다(subtraction).

우선 INF – INF 인 경우는 NAN 이 나와야 하므로 그 경우는 따로 추가해주었습니다.

```
else begin //subtraction
  flag = 0;
  if(operand1[30:23] > operand2[30:23]) begin // exp1 > exp2
    sign = operand1[31:31];
    if(operand1[30:23] == 8'b11111111) begin
      if(operand2[30:23] == 8'b11111111) flo2[31:0] = 32'hFFFFFFFF;
      else flo2[30:0] = operand1[30:0];
    end
    else begin
      if(operand2[30:23] == 8'b0) begin // normal - denormal
        if(operand2[30:0] == 31'b0) flo2[30:0] = operand1[30:0];
        else begin
          n = operand1[30:23] - operand2[30:23];
          flo1[22:0] = operand2[22:0];
          for(i=0; i<n-1; i= i+1) flo1 = flo1 >> 1;
          if(flo1[0:0] == 1) flo1 = flo1+1;
          flo1 = flo1 >> 1;
          if(operand1[22:0] >= flo1[22:0]) begin
            flo2[22:0] = operand1[22:0] - flo1[22:0];
            flo2[31:23] = operand1[31:23];
          end
          else begin
            flo2[23:23] = 1;
            flo2[22:0] = operand1[22:0];
            flo2[23:0] = flo2[23:0] - flo1[22:0];
            flo1[31:23] = operand1[31:23];
            while((flo2[23:23] != 1) && (flo1[31:23] != 8'b0)) begin
              flo2 = flo2 << 1;
              flo1[31:23] = flo1[31:23] - 1;
            end
            flo2[31:23] = flo1[31:23];
          end
        end
      end
```

Subtraction 은 sign 값을 절댓값이 큰 값으로 해주는 것과 fraction 을 더하지 않고 뺀다는 점만 다르고 그 외에는 기존의 addition 과 거의 동일합니다. 주의 할 것은 1.xxxx – 0.xxxx 가 0.xxxx 가 나올 수 있다는 것입니다. 이 경우에는 정수 부분에 1 이 올 수 있도록 normalized 해주어야 합니다.

```
      else begin  // normal - normal
        n = operand1[30:23] - operand2[30:23];
        flo1[22:0] = operand2[22:0];
        flo1[23:23] = 1;
        for(i=0; i<n-1; i= i+1) flo1 = flo1 >> 1;
        if(flo1[0:0] == 1) flo1 = flo1+1;
        flo1 = flo1 >> 1;
        if(operand1[22:0] >= flo1[22:0]) begin
          flo2[22:0] = operand1[22:0] - flo1[22:0];
          flo2[31:23] = operand1[31:23];
        end
        else begin
          flo2[23:23] = 1;
          flo2[22:0] = operand1[22:0];
          flo2[23:0] = flo2[23:0] - flo1[22:0];
          flo1[31:23] = operand1[31:23];
          while((flo2[23:23] != 1) && (flo1[31:23] != 8'b0)) begin
            flo2 = flo2 << 1;
            flo1[31:23] = flo1[31:23] - 1;
          end
          flo2[31:23] = flo1[31:23];
        end
      end

    end
  end
```

그리고 addition 에서는 순서가 상관이 없어서 첫 if 문을 operand1[30:23] >= operand2[30:23]로 하여 exponent 가 같을 때도 묶어서 해주었지만 subtraction 에서는 exponent 가 같아도 fraction 의 크기에 따라 sign 값이 정해지기 때문에 exponent 가 같을 때는 따로 만들어주었습니다.

```
else if(operand1[30:23] == operand2[30:23]) begin // exp1 == exp2
  if(operand1[30:23] == 8'b0) begin // denormal - denormal
    if(operand1[22:0] > operand2[22:0]) begin
      sign = operand1[31:31];
      flo2[22:0] = operand1[22:0] - operand2[22:0];
      flo2[30:23] = operand1[30:23];
    end
    else if(operand1[22:0] == operand2[22:0]) flo2[31:0] = 32'b0;
    else begin
      sign = operand2[31:31];
      flo2[22:0] = operand2[22:0] - operand1[22:0];
      flo2[30:23] = operand2[30:23];
    end
  end
  else begin // normal - normal
    if(operand1[22:0] > operand2[22:0]) begin
      sign = operand1[31:31];
      flo2[23:0] = operand1[22:0] - operand2[22:0];
      flo1[30:23] = operand1[30:23];
      while(flo2[23:23]!=1) begin
        flo2 = flo2 <<1;
        flo1[30:23] = flo1[30:23] - 1;
      end
      flo2[30:23] = flo1[30:23];
    end
    else if(operand1[22:0] == operand2[22:0]) flo2[31:0] = 32'b0;
    else begin
      sign = operand2[31:31];
      flo2[22:0] = operand2[22:0] - operand1[22:0];
      flo1[30:23] = operand2[30:23];
      while(flo2[23:23]!=1) begin
        flo2 = flo2 <<1;
        flo1[30:23] = flo1[30:23] - 1;
      end
      flo2[30:23] = flo1[30:23];
    end
  end
end
```

Exponent2 가 exponent1 보다 클 때는 반대의 경우와 동일하기 때문에 생략하겠습니다.

Selector 가 3'b101 인 floating point 의 multiplication 도 exponent 크기에 따라 나누었습니다.

먼저 예외 상황인 NAN, INF, 0 인 경우를 보면 0 을 INF 와 곱하면 NAN 이 출력되고 다른 숫자와 곱하면 0 이 됩니다. INF 도 마찬가지로 0 이외의 숫자와 곱하면 INF 가 됩니다. 이 경우는 flag 를 set 해주지 않습니다.

```
3'b101 : begin
  flag = 0;
  if((operand1[30:0] == 31'b0) && (operand2[30:23] == 8'b11111111)) result = 32'hFFFFFFFF;  // 0 X INF = NAN
  else if((operand2[30:0] == 31'b0) && (operand1[30:23] == 8'b11111111)) result = 32'hFFFFFFFF;  // 0 X INF = NAN
  else if((operand1[30:0] == 31'b0) || (operand2[30:0] == 31'b0)) result = 32'b0; // 0 X ? = 0
  else if((operand1[30:23] == 8'b11111111) || (operand2[30:23] == 8'b11111111)) result = 32'h7F800000; // INF X ? = INF
```

다음으로 normal * normal 일 때는 exponent 의 합을 n 에 저장해서 floating point 의 범위를 벗어나는 경우의 값을 기준으로 flag 값을 설정해주었습니다.

```
else begin
  if((operand1[30:23] != 8'b0) && (operand2[30:23] != 8'b0)) begin // normal * normal
    x = operand1[30:23] - 8'b01111111;
    y = operand2[30:23] - 8'b01111111;
    n = x + y;

    flo1[23:23] = 1;
    flo2[23:23] = 1;
    flo1[22:0] = operand1[22:0];
    flo2[22:0] = operand2[22:0];
    mul1[47:0] = flo1[23:0] * flo2[23:0];

    while(mul1[47:47]) begin
      n = n + 1;
      mul1 = mul1 >> 1;
    end

    if(n > 127) flag = 1; // overflow
    if(n < -149) flag = 1; // underflow

    if(n < -126) begin
      flo1[30:23] = 8'b0;
      while((n< -126) && (mul1 != 0)) begin
        if((n == -127) && (mul1[22:22] == 1)) mul1 = mul1 + 64'h400000;
        mul1 = mul1 >> 1;
        n = n+1;
      end
      flo1[22:0] = mul1[45:23];
    end
    else begin
      if(mul1[22:22] == 1) mul1 = mul1 + 64'h400000;
      flo1[30:23] = 8'b01111111 + n;
      flo1[22:0] = mul1[45:23];
    end
  end
```

Fraction part 에 각각 1 을 더하고 곱해준 후 상위 23 개의 bit 만을 fraction 으로 가져오고 exponent 는 exponent 의 합을 다시 floating 형식에 맞추어 입력하였습니다.

만약 n 이 -126 보다 작으면 denormal 로 바꾸어 표현할 수 있기에 exponent 를 0 으로 만들고 while 문을 사용하여 fraction part 를 표현 가능한 부분까지 shift right 해주었습니다.

마지막 shift 에서는 rounding off 를 해주었습니다. (mul1 = mul1 + 64'h400000;)

denormal * denormal 의 경우는 반드시 underflow 가 일어나기 때문에 flag bit 를 set 해주었습니다.

```
else if((operand1[30:23] == 8'b0) && (operand2[30:23] == 8'b0)) flag = 1; // denormal * denormal
```

normal * denormal 경우는 normal * normal 과 달리 exponent 의 합에서 1 을 더해주었습니다. (denormal 의 지수가 -127 이 아닌 -126 이기 때문에)

이 경우는 1.xxxx * 0.xxxx 이기 때문에 fraction bit 를 1.xxxx 형태로 맞추기 위해 while 문을 사용하였습니다. 이외에는 normal * normal 과 동일합니다.

```
    else begin // normal * denormal
      x = operand1[30:23] - 8'b01111111;
      y = operand2[30:23] - 8'b01111111;
      n = x + y + 1;

      if(n < -149) flag = 1; // underflow

      if(operand1[30:23] != 8'b0) flo1[23:23] = 1;
      if(operand2[30:23] != 8'b0)  flo2[23:23] = 1;
      flo1[22:0] = operand1[22:0];
      flo2[22:0] = operand2[22:0];
      mul1[47:0] = flo1[23:0] * flo2[23:0];

      while(mul1[46:46] != 1) begin
        mul1 = mul1 << 1;
        n = n - 1;
      end

      if(n < -126) begin
        flo1[30:23] = 8'b0;
        while((n< -126) && (mul1 != 0)) begin
          if((n == -127) && (mul1[22:22] == 1)) mul1 = mul1 + 64'h400000;
          mul1 = mul1 >> 1;
          n = n+1;
        end
        if(n!=-126) flag = 1;
        flo1[22:0] = mul1[45:23];
      end
      else begin
        if(mul1[22:22] == 1) mul1 = mul1 + 64'h400000;
        flo1[30:23] = 8'b01111111 + n;
        flo1[22:0] = mul1[45:23];
      end

    end

    sign = operand1[31:31] ^ operand2[31:31];
    result[31:31] = sign;
    result[30:0] = flo1[30:0];

  end
  done <= 1;
end
```

Selector 가 3'b110 인 floating point 을 fixed point 로 변환하는 연산은 우선 표현 가능한 범위가 크게 차이 나기 때문에 overflow 나 underflow 가 발생하는 경우가 많습니다. 그렇기 때문에 exponent 로 flag bit 부터 확인해주었습니다.

```
3'b110 : begin
  flag = 0;
  flo1[22:0] = operand1[22:0];
  flo2[22:0] = operand1[22:0];
  x = operand1[30:23] - 8'b01111111;
  if(x>14) flag = 1; //overflow
  else if(x<-16) flag = 1; //underflow
  else begin
    if(x>=0) begin
      flo1[23:23] = 1;
      flo2[23:23] = 1;
      for(i=0; i<x; i = i+1) flo2 = flo2 << 1;
      fix1[15:0] = flo2[22:7];
      for(i=22; i>=x; i = i-1) flo1 = flo1 >> 1;
      fix1[30:16] = flo1[14:0];
      if(operand1[31:31] ==0) result[30:0] = fix1[30:0];
      else result = ~fix1 + 1;
      done <= 1;
    end
    else begin
      flo1[23:23] = 1;
      flo2[23:23] = 1;
      for(i=x; i<0; i = i+1) flo1 = flo1 >> 1;
      fix1[15:0] = flo1[22:7];
      fix1[30:16] = 15'b0;
      if(operand1[31:31] ==0) result[30:0] = fix1[30:0];
      else result = ~fix1 + 1;
    end
  end
  if(operand1[30:0] == 31'b0) begin
    result[31:0] = 32'b0;
    flag = 0;
  end
  done <= 1;
end
```

Flag 가 발생하지 않으면 fraction bit 를 따와 1.xxxx 형태로 만들어준 후 exponent 에 맞추어 shift right / left 를 통해 정수부분과 소수부분을 찾아주었습니다. 그 후 sign bit 가 1 이면 2's complement 해주었습니다.

마지막으로 Selector 가 3'b111 인 floating point 의 크기 비교는 sign bit 를 우선으로 비교해준 후 sign bit 가 같으면 [30:0]의 31bit 로 비교해주었습니다.

```
      3'b111 : begin
        if((operand1[31:31] == 0) && (operand2[31:31] ==0)) begin
        result <= (operand1 >= operand2)? operand1 : operand2;
        end
        if((operand1[31:31] == 0) && (operand2[31:31] ==1)) begin
        result <= operand1;
        end
        if((operand1[31:31] == 1) && (operand2[31:31] ==0)) begin
        result <= operand2;
        end
        if((operand1[31:31] == 1) && (operand2[31:31] ==1)) begin
        result <= (operand1 >= operand2)? operand2 : operand1;
        end
        flag <= 0;
        done <=1;
      end
    endcase
  end
end
```

코드 설명은 이상으로 마치겠습니다. 마지막으로 각 형식의 장단점을 설명하겠습니다.

Fixed point 의 장점은 연산 과정이 쉽다는 것입니다. 더하는 것과 곱하는 것 모두 특별한 변환없이 바로 가능합니다. 다만 그만큼 단점이 너무 큽니다. 표현 가능한 범위가 너무 적어 원하는 계산을 해낼 가능성이 매우 적습니다.

Floating point 의 장점은 표현 가능한 범위가 매우 넓다는 것입니다. 절댓값으로 2^(-149) ~ 2^128- 2^105 까지의 범위를 표현이 가능합니다. 하지만 계산과정이 많이 복잡합니다. Exponent 값과 fraction 부분을 따로 나누어 계산해야 하고 fraction 의 값에 따라 exponent 값의 수정이 필요하고, exponent 의 값에 따라 normal, denormal 의 형태로 모양이 다르기 때문에 그것 또한 고려해 주어야합니다.

아래는 저의 test 코드를 실행하였을 때 사진입니다.

```
VSIM 58> run -all
# selector : 000 opernad1 : 00000000 operand2 : 00000000 ans : 00000000 your ans : 00000000 <right>
# selector : 000 opernad1 : 7fffffff operand2 : 7fffffff your flag bit is 1 <right>
# selector : 111 opernad1 : 00000000 operand2 : fffffffe ans : 00000000 your ans : 00000000 <right>
# selector : 001 opernad1 : 00000000 operand2 : 7fffffff ans : 00000000 your ans : 00000000 <right>
# selector : 000 opernad1 : ffffffff operand2 : 7fffffff ans : 7ffffffe your ans : 7ffffffe <right>
# selector : 010 opernad1 : 7fffffff operand2 : 00000000 ans : 46ffffff your ans : 46ffffff <right>
# selector : 001 opernad1 : 7fffffff operand2 : 0001000f your flag bit is 1 <right>
# selector : 001 opernad1 : 7fffffff operand2 : 0000000f ans : 00078000 your ans : 00078000 <right>
# selector : 000 opernad1 : 00001111 operand2 : 00000001 ans : 00001112 your ans : 00001112 <right>
# selector : 000 opernad1 : 00001111 operand2 : 00000001 ans : 00001112 your ans : 00001112 <right>
# selector : 000 opernad1 : 00011111 operand2 : 00000010 ans : 00011121 your ans : 00011121 <right>
# selector : 000 opernad1 : 00011111 operand2 : 00000001 ans : 00011112 your ans : 00011112 <right>
# selector : 010 opernad1 : 00081111 operand2 : 00000000 ans : 41011110 your ans : 41011110 <right>
# selector : 010 opernad1 : 00010000 operand2 : 00000000 ans : 3f800000 your ans : 3f800000 <right>
# selector : 000 opernad1 : 00001111 operand2 : 00000001 ans : 00001112 your ans : 00001112 <right>
# selector : 000 opernad1 : 00001111 operand2 : 00000001 ans : 00001112 your ans : 00001112 <right>
# selector : 000 opernad1 : 00001111 operand2 : 00000001 ans : 00001112 your ans : 00001112 <right>
# selector : 000 opernad1 : 00001111 operand2 : 00000001 ans : 00001112 your ans : 00001112 <right>
# selector : 010 opernad1 : 00011111 operand2 : 00000000 ans : 3f888880 your ans : 3f888880 <right>
# selector : 010 opernad1 : 00010000 operand2 : 00000000 ans : 3f800000 your ans : 3f800000 <right>
# selector : 100 opernad1 : 44eb0000 operand2 : 43d40000 ans : 45100000 your ans : 45100000 <right>
# selector : 100 opernad1 : 44eb0000 operand2 : c3d40000 ans : 44b60000 your ans : 44b60000 <right>
# selector : 100 opernad1 : c4eb0000 operand2 : 43d40000 ans : c4b60000 your ans : c4b60000 <right>
# selector : 100 opernad1 : 44eb0000 operand2 : c3d40000 ans : 44b60000 your ans : 44b60000 <right>
# selector : 101 opernad1 : 44c00000 operand2 : 43800000 ans : 48c00000 your ans : 48c00000 <right>
# selector : 110 opernad1 : 43f88888 operand2 : 00000000 ans : 01f11110 your ans : 01f11110 <right>
# selector : 000 opernad1 : 80000001 operand2 : 0000000f ans : 80000010 your ans : 80000010 <right>
# selector : 001 opernad1 : ffffffff operand2 : ffffffff ans : 00000000 your ans : 00000000 <right>
# selector : 001 opernad1 : 00008000 operand2 : 00002000 ans : 00001000 your ans : 00001000 <right>
# selector : 001 opernad1 : 00000000 operand2 : 40002000 ans : 00000000 your ans : 00000000 <right>
# selector : 001 opernad1 : e000203f operand2 : 00000004 ans : ffff8001 your ans : ffff8001 <right>
# selector : 001 opernad1 : 40000000 operand2 : fffe0000 ans : 80000000 your ans : 80000000 <right>
# selector : 001 opernad1 : e0ff0104 operand2 : 3df21f5e your flag bit is 1 <right>
# selector : 001 opernad1 : 0193f2e3 operand2 : fedc102f your flag bit is 1 <right>
# selector : 001 opernad1 : 00400000 operand2 : 00001000 ans : 00040000 your ans : 00040000 <right>
# selector : 011 opernad1 : 00000005 operand2 : 00000003 ans : 00000005 your ans : 00000005 <right>
# selector : 011 opernad1 : ffffffff operand2 : 00000002 ans : 00000002 your ans : 00000002 <right>
# selector : 011 opernad1 : fffffffe operand2 : fffffffc ans : fffffffe your ans : fffffffe <right>
# selector : 100 opernad1 : f7000001 operand2 : 71800600 ans : f6ffe000 your ans : f6ffe000 <right>
# selector : 100 opernad1 : 71800600 operand2 : 00000001 ans : 71800600 your ans : 71800600 <right>
# selector : 100 opernad1 : 71800600 operand2 : 80000001 ans : 71800600 your ans : 71800600 <right>
# selector : 100 opernad1 : 71800600 operand2 : f1800001 ans : 6b3fe000 your ans : 6b3fe000 <right>
# selector : 101 opernad1 : 71800600 operand2 : f1800001 your flag bit is 1 <right>
# selector : 100 opernad1 : 71800600 operand2 : 40000001 ans : 71800600 your ans : 71800600 <right>
```

```
# selector : 101 opernad1 : 71800600 operand2 : f1800001 your flag bit is 1 <right>
# selector : 100 opernad1 : 71800600 operand2 : 40000001 ans : 71800600 your ans : 71800600 <right>
# selector : 101 opernad1 : 00000000 operand2 : ffffffff ans : ffffffff your ans : ffffffff <right>
# selector : 101 opernad1 : 70000000 operand2 : 7f7fffff your flag bit is 1 <right>
# selector : 100 opernad1 : 70000000 operand2 : 7f7fffff ans : 7f7fffff your ans : 7f7fffff <right>
# selector : 100 opernad1 : ff000001 operand2 : 71800600 ans : ff000001 your ans : ff000001 <right>
# selector : 101 opernad1 : 00000000 operand2 : 7f7fffff ans : 00000000 your ans : 00000000 <right>
# selector : 101 opernad1 : 31800600 operand2 : ff000001 ans : f1000601 your ans : f1000601 <right>
# selector : 101 opernad1 : 71800600 operand2 : 00000001 ans : 27000600 your ans : 27000600 <right>
# selector : 101 opernad1 : 71800600 operand2 : 60540001 your flag bit is 1 <right>
# selector : 101 opernad1 : 31800600 operand2 : 00000001 your flag bit is 1 <right>
# selector : 101 opernad1 : 71800600 operand2 : 4d540001 ans : 7f5409f1 your ans : 7f5409f1 <right>
# selector : 000 opernad1 : 80000000 operand2 : ffffffff your flag bit is 1 <right>
# selector : 010 opernad1 : ffffffff operand2 : 00000000 ans : b7800000 your ans : b7800000 <right>
# selector : 010 opernad1 : fffffffe operand2 : 00000000 ans : b8000000 your ans : b8000000 <right>
# selector : 010 opernad1 : ffff0000 operand2 : 00000000 ans : bf800000 your ans : bf800000 <right>
# selector : 110 opernad1 : b7800000 operand2 : 00000000 ans : ffffffff your ans : ffffffff <right>
# selector : 110 opernad1 : b8000000 operand2 : 00000000 ans : fffffffe your ans : fffffffe <right>
# selector : 110 opernad1 : bf800000 operand2 : 00000000 ans : ffff0000 your ans : ffff0000 <right>
# selector : 100 opernad1 : ff0b0000 operand2 : ff040000 your flag bit is 1 <right>
# selector : 100 opernad1 : 7f0b0000 operand2 : ff040000 ans : 7ce00000 your ans : 7ce00000 <right>
# selector : 100 opernad1 : 7f0b0000 operand2 : 7f040000 your flag bit is 1 <right>
# selector : 100 opernad1 : 6f0b0000 operand2 : 6f040000 ans : 6f878000 your ans : 6f878000 <right>
# selector : 100 opernad1 : 6f0b0000 operand2 : 6f040000 ans : 6f878000 your ans : 6f878000 <right>
# selector : 101 opernad1 : 3b002000 operand2 : 00840000 ans : 00004210 your ans : 00004210 <right>
# selector : 101 opernad1 : 3f002000 operand2 : 00840000 ans : 00421080 your ans : 00421080 <right>
# ** Note: $finish    : C:/classes/digital/LAB3/top.v(74)
#    Time: 136300 ps  Iteration: 0  Instance: /lab3_top
# 1
# Break in Module lab3_top at C:/classes/digital/LAB3/top.v line 74
```