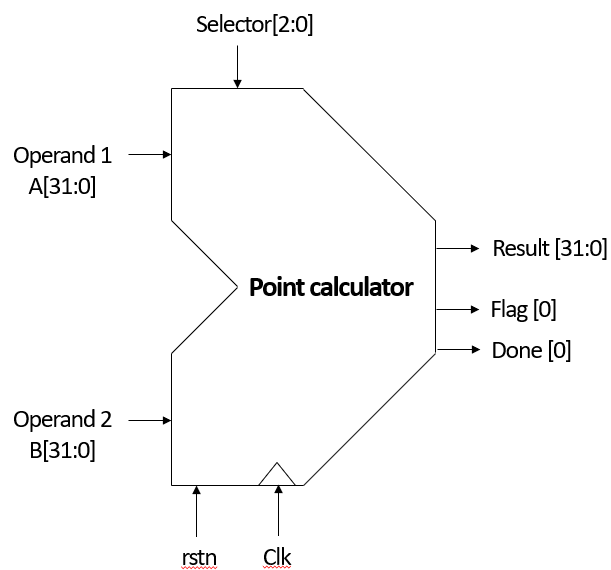
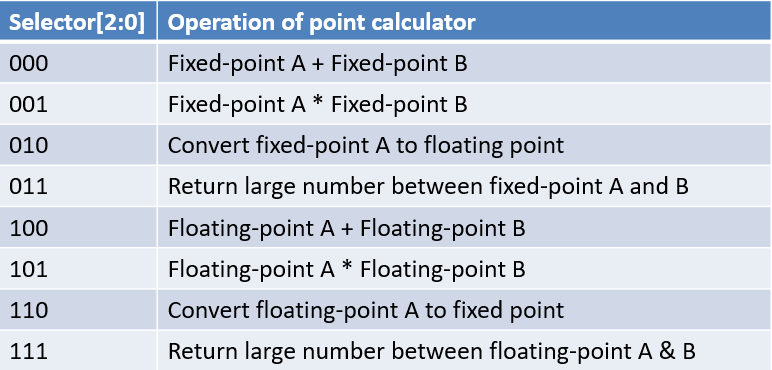
**디지털시스템설계 LAB3**

2016310936 우승민

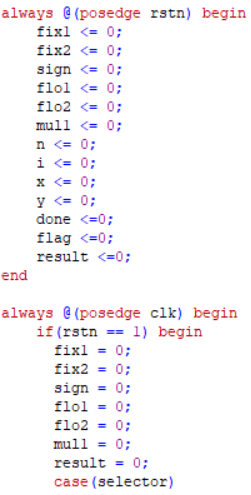
LAB3의 목표는 아래의 **Point calculator**를 구현하는 것입니다.



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



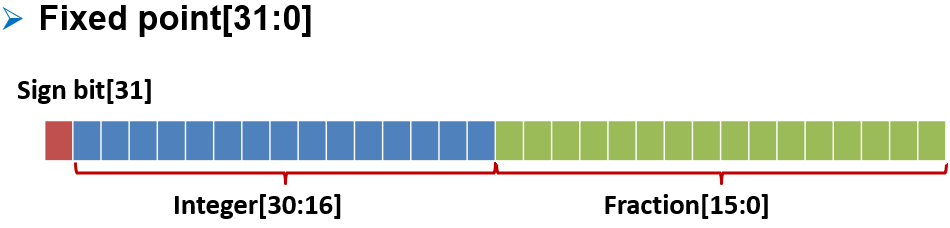
Selector로 선택한 operation의 operand1과 operand2의 올바른 결과를 출력하는 것이 이번 과제 목표입니다. 만약 계산 결과에서 overflow / underflow가 일어날 경우에는 결과값은 상관하지 않고 flag bit를 set해 줍니다.



우선 rstn값이 set이 되어야 동작을 하기 때문에 rstn값이 처음 set되면 사용하는 모든 register 값들을 초기화 해주고, 그 이후 clock이 positive edge일 때 값들을 다시 초기화 한 후 selector의 값에 따라 case문을 실행하게 해주었습니다.

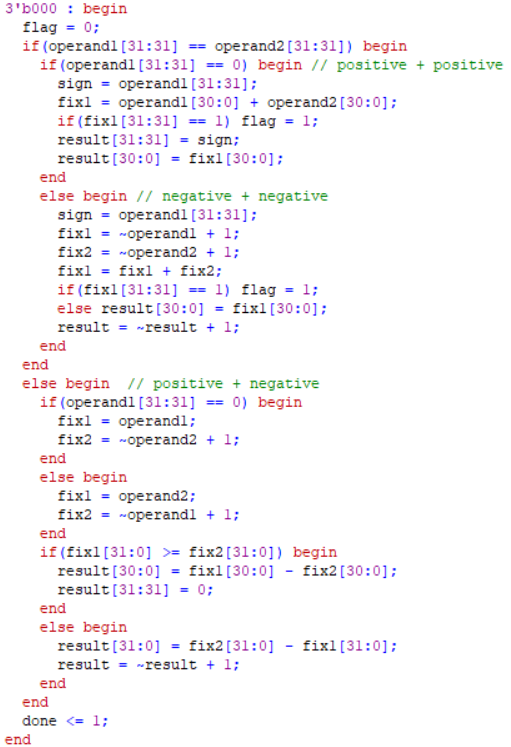
이제 selector의 값이 000 ~ 011일 때 계산해주는 fixed point를 보면 최상위 비트는 sign bit를 뜻하고 하위 16비트는 소수점을 나타내고 나머지 16부터 30까지 15bit는 정수부분을 표현합니다.

여기서 fixed point는 2’s complement입니다. 따라서 범위는 (-2^15 ~ 2^15-2^(-16))입니다.



Selector가 3’b000인 addition의 경우부터 설명하겠습니다. Fixed point는 2’s complement이기 때문에 operand1, 2의 sign bit값에 따라 3가지 경우로 나누었습니다.

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

positive + positive일 경우에는 operand1, 2의 [30:0] bit를 단순히 더해주었습니다. 만약 31개의 bit 2개를 더했는데 32bit가 set되면 범위를 벗어난 것이므로 flag를 set해주었습니다.

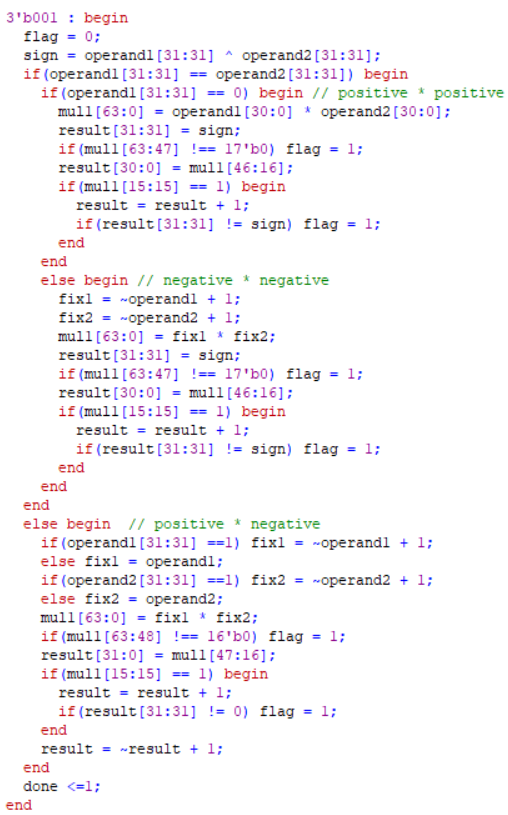
Negative + negative 경우는 operand1, 2를 2’s complement의 방식으로 양수로 바꾼 후 같은 방식으로 해준 후 마지막에 다시 음수로 바꾸어 주었습니다.

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

Selector가 3’b001인 multiplication의 경우도 마찬가지로 operand1, 2의 sign bit값에 따라 3가지 경우로 나누었습니다.

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

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

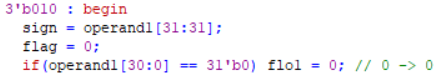
우선 result의 sign bit는 각 operand의 sign bit의 xor를 통해 구해주었습니다.

Fixed point를 곱한 후 만약 48bit 이상의 값이 0이 아니면 fixed point의 표현 범위를 초과한 것이므로 flag를 set해주었습니다.

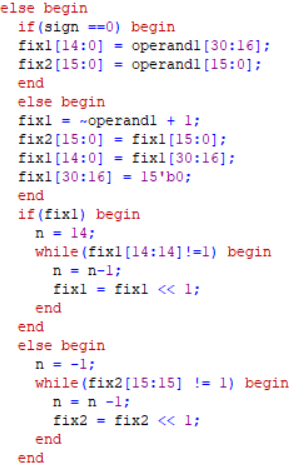
그리고 16bit가 set되어있으면 rounding up을 해주었습니다.

이 후 negative의 operand는 addition과 마찬가지로 2’s complement를 해준 후 계산해 주었습니다.

Selector가 3’b010인 경우는 fixed point -> floating point로 변환하는 연산입니다. 우선 float point의 표현 범위가 fixed point보다 훨씬 크기 때문에 flag가 발생할 일은 없습니다. 또한 생각할 점은 0을 제외한 모든 fixed point는 float point의 normal number라는 점입니다.

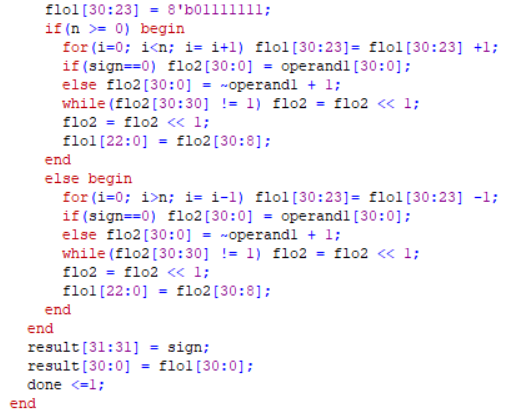


0을 제외한 숫자는 exponent와frac part를 구하는 2가지 과정을 거치게 하였습니다.



이 부분은 exponent를 구하는 과정입니다. 먼저 operand1이 정수부분을 가지고 있는지 확인하고 있으면 정수부분 중 최대 bit 위치를 확인하고 n에 exp를 저장합니다.

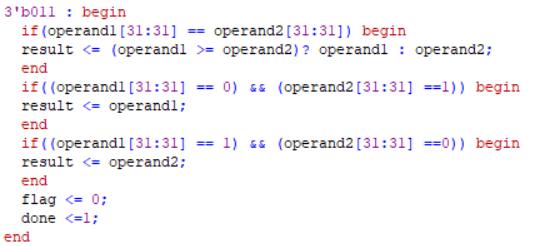
0의 경우는 이미 위에서 따로 해주었기 때문에 만약 정수부분이 없으면 소수부분에 반드시 최소 1개의 bit가 1이 되어있을 것입니다. while문을 통해 가장 큰 bit의 위치를 찾고 n에 exp를 저장합니다.

Exponent는 0인 8’b01111111을 기준으로 위에서 구한 n만큼을 더하여 구할 수 있습니다.

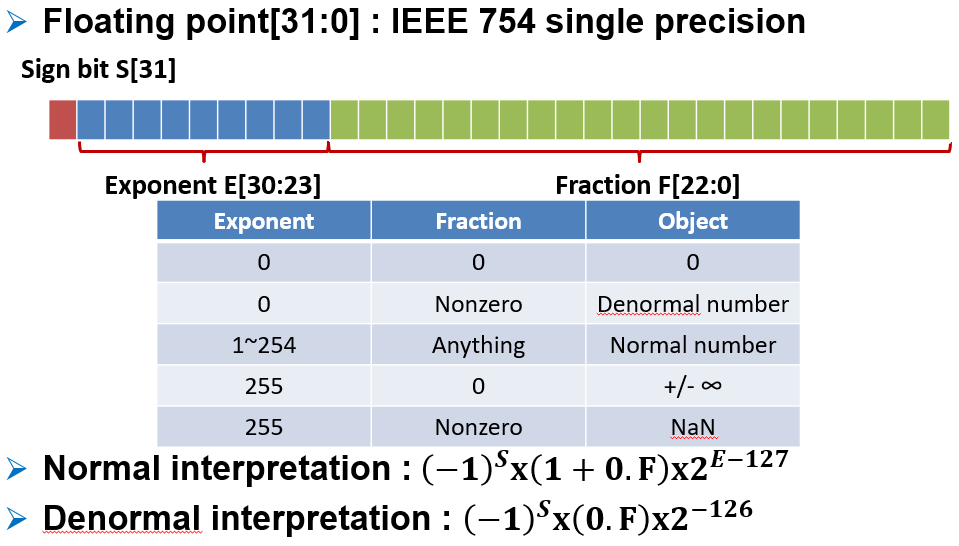
그 후 frac part는 Operand1의 sign bit를 제외한 31bit를 가져와 1.xxxx \* 2^exp중 1에 해당하는 가장 큰 bit를 while문을 통해 구합니다.

이후 소수부분만 float에 저장해 주어야 하기 때문에 left shift를 1번 더 해주고 frac part를 저장합니다.

Selector가 3’b011인 경우는 단순히 크기 비교만 해주면 됩니다.



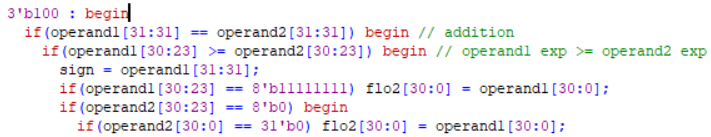
다음으로 floating point의 경우를 설명하겠습니다.



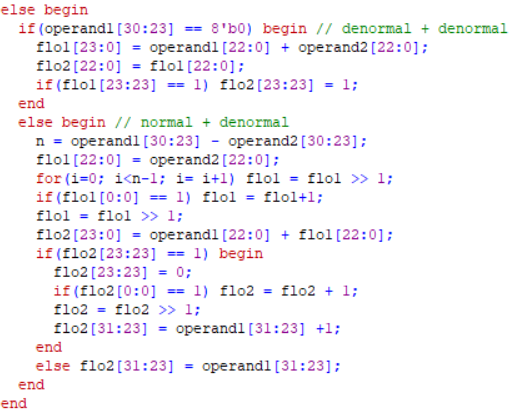
Floating point는 위의 표현처럼 지수부분(exponent)와 소수부분(fraction)으로 구성되어 있기에 두 부분을 따로 구해주어야 합니다. 또한 표처럼 5가지 경우에 따라 형식이 다르기 때문에 주의해 주어야합니다.

Selector가 3’b101인 floating point의 addition은 exponent 크기에 따라 나누었습니다.

나누는 기준은 exponent1 과 exponent2의 크기 비교를 한 후 denormal과 normal인 경우에 따라 나누어주었습니다.



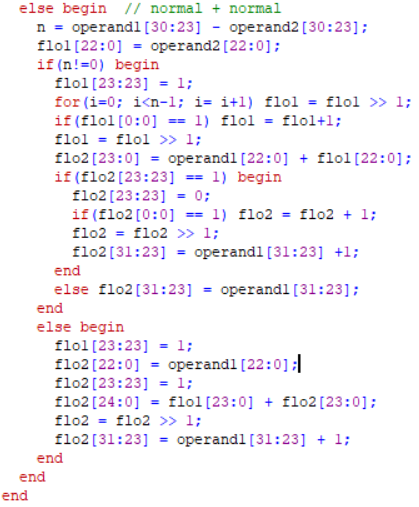
여기서 operand2가 0이 아니면 denormal이고, operand1과 더하는 과정은 아래와 같습니다.



operand1도 denormal이면 fraction을 그대로 더해주고 만약 0.xxx + 0.xxx에서 1.xxx가 되면 normal형태로 바꾸기 위해 exponent값을 1로 해주었습니다.

Operand1이 normal이면 exponent의 차이를 구하고 그 만큼 operand2의 fraction을 right shift한 후 더해주었습니다. 주의할 것은 마지막에 rounding off를 해주기 위해 for문을 한번 덜 하고 rounding 후 다시 한번만 shift right 하는 것입니다.

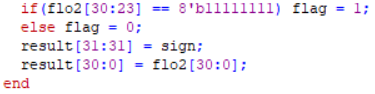
그 후 fraction part를 더하고 1.xxxx + 0.xxxx 에서 10.xxxx가 될 경우에는 exponent값을 1 더해주고 fraction을 shift right합니다.

다음으로 operand2가 normal인 경우에는 현재 exponent1을 exponent2의 이상으로 두었기 때문에 operand1도 normal입니다.

Code의 대부분은 normal + denormal과 비슷하지만 여기서는 exponent값이 같을 때와 다를 때 나누어주었습니다.

또한 denormal과 달리 operand2의 fraction이 1.xxxx이기 때문에 24bit에 1을 넣어준 후 계산하였습니다.

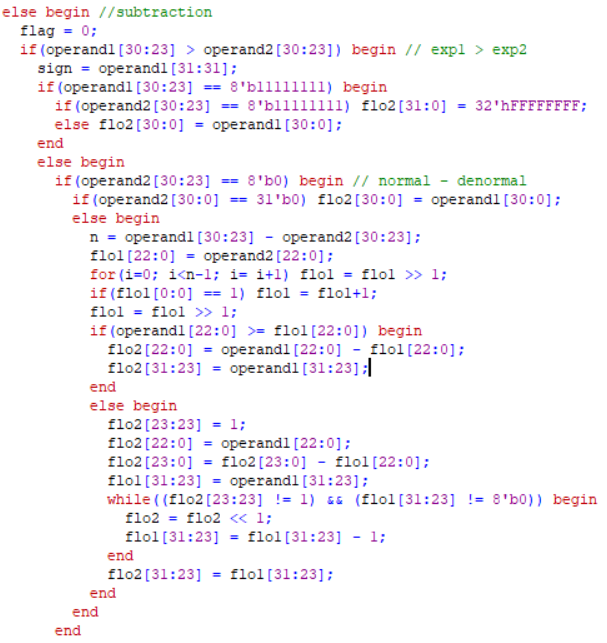
Exponent2가 exponent1보다 큰 경우에는 반대로만 해주고 똑같기 때문에 생략하겠습니다.



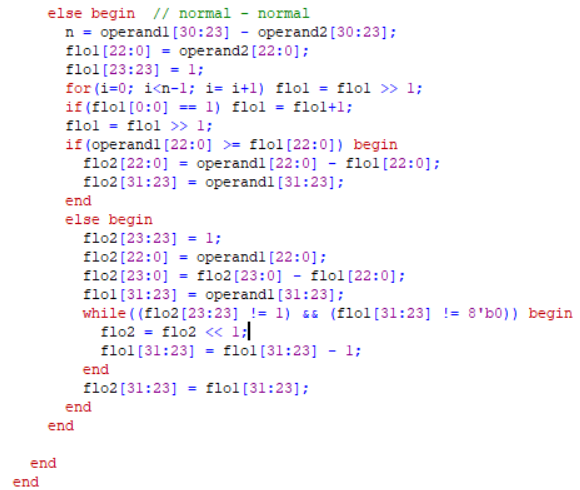
이후 result에 값을 넣을 때 만약 floating point의 범위를 넘어서 exponent가 8’b11111111이 되면 flag 값을 set해 주었습니다.

다음은 둘의 sign bit가 다를 때의 addition을 설명하겠습니다(subtraction).

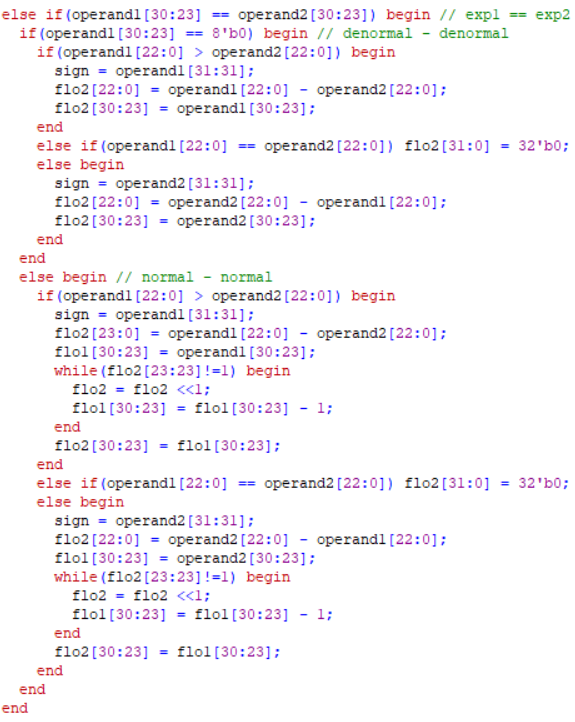
우선 INF – INF인 경우는 NAN이 나와야 하므로 그 경우는 따로 추가해주었습니다.



Subtraction은 sign 값을 절댓값이 큰 값으로 해주는 것과 fraction을 더하지 않고 뺀다는 점만 다르고 그 외에는 기존의 addition과 거의 동일합니다. 주의 할 것은 1.xxxx – 0.xxxx가 0.xxxx가 나올 수 있다는 것입니다. 이 경우에는 정수 부분에 1이 올 수 있도록 normalized 해주어야 합니다.



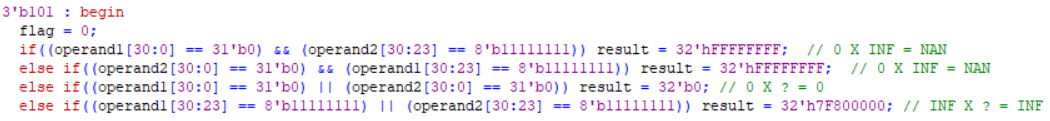
그리고 addition에서는 순서가 상관이 없어서 첫 if문을 operand1[30:23] >= operand2[30:23]로 하여exponent가 같을 때도 묶어서 해주었지만 subtraction에서는 exponent가 같아도 fraction의 크기에 따라 sign값이 정해지기 때문에 exponent가 같을 때는 따로 만들어주었습니다.



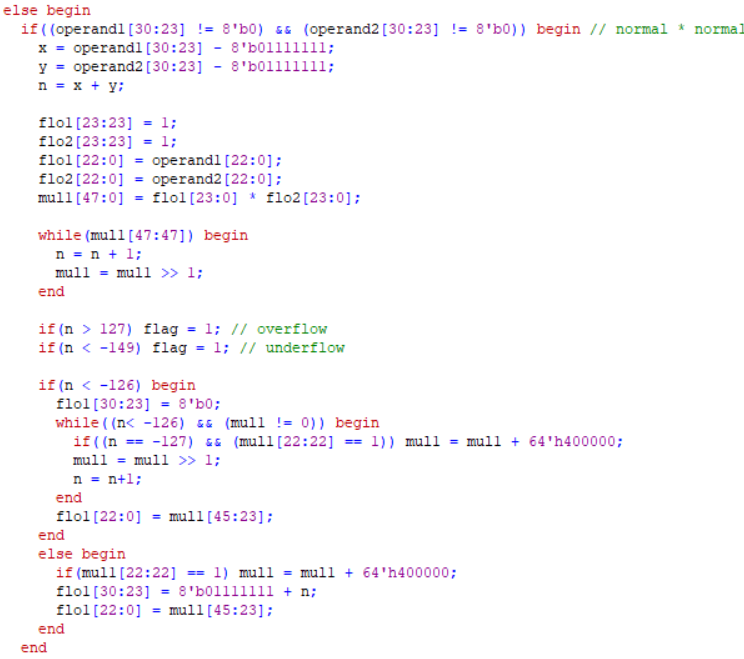
Exponent2가 exponent1보다 클 때는 반대의 경우와 동일하기 때문에 생략하겠습니다.

Selector가 3’b101인 floating point의 multiplication도 exponent 크기에 따라 나누었습니다.

먼저 예외 상황인 NAN, INF, 0인 경우를 보면 0을 INF와 곱하면 NAN이 출력되고 다른 숫자와 곱하면 0이 됩니다. INF도 마찬가지로 0이외의 숫자와 곱하면 INF가 됩니다. 이 경우는 flag를 set해주지 않습니다.



다음으로 normal \* normal 일 때는 exponent의 합을 n에 저장해서 floating point의 범위를 벗어나는 경우의 값을 기준으로 flag 값을 설정해주었습니다.



Fraction part에 각각 1을 더하고 곱해준 후 상위 23개의 bit만을 fraction으로 가져오고 exponent는 exponent의 합을 다시 floating 형식에 맞추어 입력하였습니다.

만약 n이 -126보다 작으면 denormal로 바꾸어 표현할 수 있기에 exponent를 0으로 만들고 while문을 사용하여 fraction part를 표현 가능한 부분까지 shift right 해주었습니다.

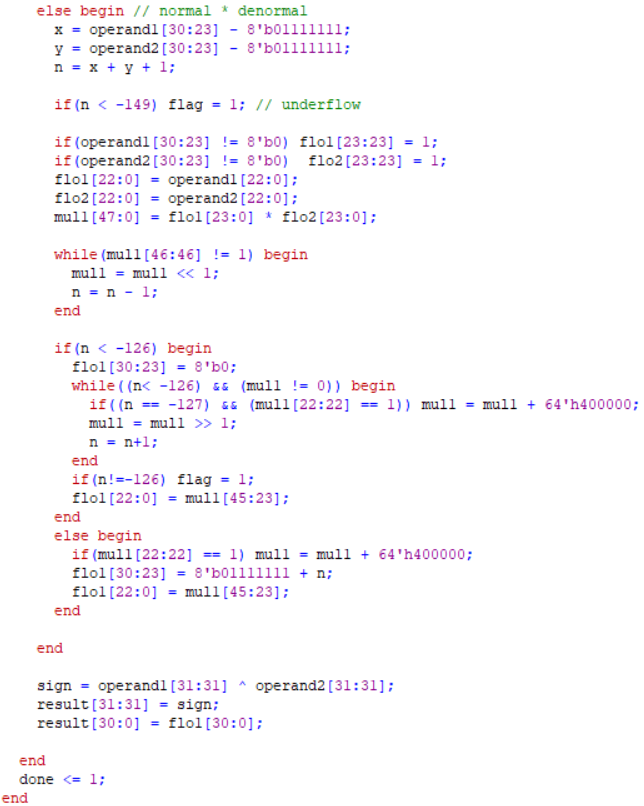
마지막 shift에서는 rounding off를 해주었습니다. (mul1 = mul1 + 64’h400000;)

denormal \* denormal의 경우는 반드시 underflow가 일어나기 때문에 flag bit를 set해주었습니다.

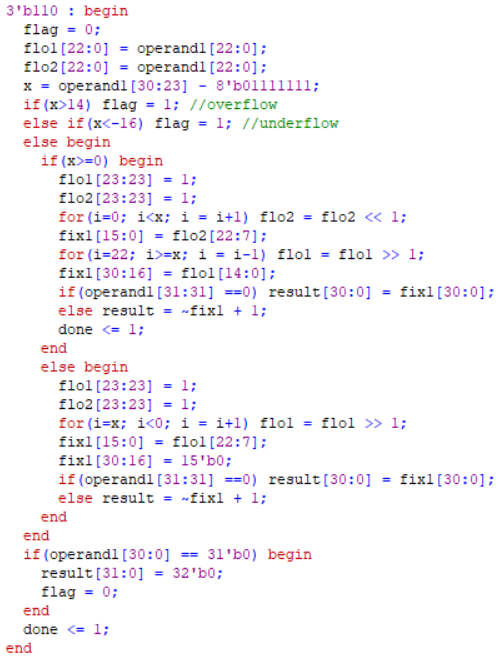


normal \* denormal 경우는 normal \* normal과 달리 exponent의 합에서 1을 더해주었습니다. (denormal의 지수가 -127이 아닌 -126이기 때문에)

이 경우는 1.xxxx \* 0.xxxx 이기 때문에 fraction bit를 1.xxxx 형태로 맞추기 위해 while문을 사용하였습니다. 이외에는 normal \* normal과 동일합니다.

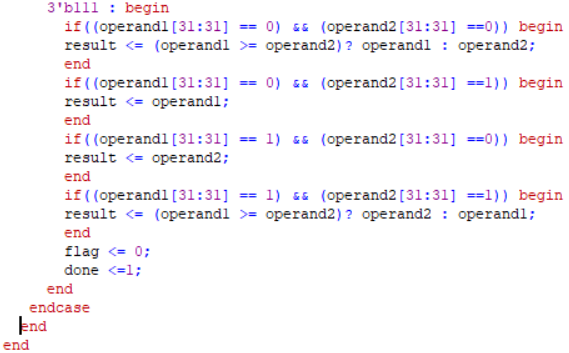


Selector가 3’b110인 floating point을 fixed point로 변환하는 연산은 우선 표현 가능한 범위가 크게 차이 나기 때문에 overflow나 underflow가 발생하는 경우가 많습니다. 그렇기 때문에 exponent로 flag bit부터 확인해주었습니다.



Flag가 발생하지 않으면 fraction bit를 따와 1.xxxx 형태로 만들어준 후 exponent에 맞추어 shift right / left를 통해 정수부분과 소수부분을 찾아주었습니다. 그 후 sign bit가 1이면 2’s complement 해주었습니다.

마지막으로 Selector가 3’b111인 floating point의 크기 비교는 sign bit를 우선으로 비교해준 후 sign bit가 같으면 [30:0]의 31bit로 비교해주었습니다.



코드 설명은 이상으로 마치겠습니다. 마지막으로 각 형식의 장단점을 설명하겠습니다.

Fixed point의 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 point의 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 fraction의 값에 따라 exponent값의 수정이 필요하고, exponent의 값에 따라 normal, denormal 의 형태로 모양이 다르기 때문에 그것 또한 고려해 주어야합니다.

아래는 저의 test 코드를 실행하였을 때 사진입니다.

