산학협력 프로젝트 : FP_ADDSUB.v 분석 보고서

2020253113 김대엽
2021245099 김용호
2022245147 하지혁

0. 순서

1. 실행 환경

Quartus Prime :
하드웨어 기술 언어(HDL) 설계의 분석과 합성에 사용되는 소프트웨어 도구
Altera에서 배포된다.
코드 작성 및 테스트 :
Verilog 파일로 코드를 작성했으며, 작성된 코드의 검증을 위해 임의의 테스트 세트를 생성 후 University Program VWF 파일을 생성해 코드를 검증했다.

2. 코드 분석

<전체 코드>

```
module fp_addsub(in_a, in_b, sub, out);
        input [31:0] in_a, in_b;      // 정규화된 입력으로 가정
        input sub;
        output [31:0] out;

        // 입력을 부호부(1bit), 지수부(8bit), 가수부(23bit)로 분리
        wire sign_a, sign_b;
        wire [7:0] exp_a, exp_b;
        wire [22:0] frac_a, frac_b;
        assign {sign_a, exp_a, frac_a} = in_a;
        assign {sign_b, exp_b, frac_b} = in_b;

        // 절댓값 비교 (|a| > |b| 이면 1)
        wire magnitude_a_gt_b = (exp_a > exp_b) || ((exp_a == exp_b) && (frac_a > frac_b));

        // 입력 a,b의 0 여부 확인
        wire is_zero_a = (exp_a==0 && frac_a==0);
        wire is_zero_b = (exp_b==0 && frac_b==0);

        // 가수부에 hidden bit 추가 (단, 0인 경우 hidden bit 없이 0으로 설정)
        wire [23:0] sig_a = is_zero_a ? 24'b0 : {1'b1, frac_a};
        wire [23:0] sig_b = is_zero_b ? 24'b0 : {1'b1, frac_b};

        // 절댓값 비교 결과에 따른 입력 선택
        wire [7:0] exp_ge = magnitude_a_gt_b ? exp_a : exp_b;
        wire [7:0] exp_lt = magnitude_a_gt_b ? exp_b : exp_a;
        wire [23:0] sig_ge = magnitude_a_gt_b ? sig_a : sig_b;
```

```
        wire [23:0] sig_lt = magnitude_a_gt_b ? sig_b : sig_a;

        // 지수 차이 계산 및 작은 측 지수를 shift
        wire [7:0] exp_diff = exp_ge - exp_lt;
        wire [23:0] sig_lt_shifted = sig_lt >> exp_diff;

        // 실제 연산이 덧셈인지 뺄셈인지 결정
        wire is_add = (~sub && ~(sign_a ^ sign_b)) | (sub & (sign_a ^ sign_b));

        // 오버플로우(carry 발생)를 고려해 1bit 여유를 두고 연산 수행
        wire [24:0] raw_mantissa = is_add ?
                ({1'b0, sig_ge} + {1'b0, sig_lt_shifted}) :
                ({1'b0, sig_ge} - {1'b0, sig_lt_shifted});

        // 최종 지수부 결정 : 절댓값이 큰 입력의 지수부를 사용
        wire [7:0] raw_exp = exp_ge;

        // 최종 부호부 결정 : 절댓값이 큰 입력의 "실질적 부호"를 최종 지수부로 사용
        wire raw_sign = (raw_mantissa==0) ?
                1'b0 :
                (magnitude_a_gt_b ? sign_a : (sign_b ^ sub));

        // 정규화
        wire [7:0] exp_out;
        wire [22:0] frac_out;
        fp_norm u_norm (raw_exp, raw_mantissa, exp_out, frac_out);

        // 최종 출력 결과 조립 (단, 두 입력이 모두 0이면 최종 출력 0 처리)
        assign out = (is_zero_a & is_zero_b) ?
                32'b0 :
                {raw_sign, exp_out, frac_out};
endmodule

module fp_norm(exp_in, mantissa_in, exp_out, mantissa_out);
        localparam M_WIDTH = 25;
        input signed [7:0] exp_in;
        input [M_WIDTH-1:0] mantissa_in;
        output signed [7:0] exp_out;
        output [22:0] mantissa_out;

        // 덧셈 오버플로우 확인
        wire add_overflow = mantissa_in[M_WIDTH-1];

        /* Case 1 : 덧셈 오버플로우 시 정규화 (오른쪽 shift) */

        // 지수부 1 증가
        wire signed [7:0] rshift_exp_out = exp_in + 1;

        // 1만큼 오른쪽 shift
        wire [M_WIDTH-1:0] rshift_mantissa_norm = mantissa_in >> 1;
        // 오른쪽 shift 결과 부분만 추출
        wire [22:0] rshift_mantissa_out = rshift_mantissa_norm[22:0];


        /* Case 2 : 그 외의 경우 정규화 (왼쪽 shift) */
```

```
        localparam EXT_WIDTH = 32;

        // 입력 bit 확장(25bit -> 32bit)
        wire [EXT_WIDTH-1:0] mantissa_extended = {{(EXT_WIDTH - M_WIDTH){1'b0}}, mantissa_in};

        // lod 모듈을 호출하여 선행 1의 존재 여부와 선행 0의 개수 확인
        wire [$clog2(EXT_WIDTH)-1:0] lz_count_extended;
        wire valid;
        lod #(EXT_WIDTH) u_lod (mantissa_extended, lz_count_extended, valid);

        // bit 확장 이전 선행 0의 개수 확인
        wire signed [$clog2(M_WIDTH):0] lz_count_actual = lz_count_extended - (EXT_WIDTH - M_WIDTH);

        // 선행 0의 개수를 바탕으로 필요한 shift 개수 계산
        wire signed [$clog2(M_WIDTH):0] shift_left = lz_count_actual - 1;

        // shift 개수만큼 지수부 감소
        wire signed [7:0] lshift_exp_out = exp_in - shift_left;

        // shift 개수만큼 왼쪽 shift
        wire [M_WIDTH-1:0] lshift_mantissa_norm = mantissa_in << shift_left;

        // 왼쪽 shift 결과 부분만 추출
        wire [22:0] lshift_mantissa_out = lshift_mantissa_norm[22:0];

        // 오버플로우 및 선행 1 존재 여부에 따라 최종 정규화 결과 선택
        assign exp_out = ~valid ? 8'b0 :
                (add_overflow ? rshift_exp_out : lshift_exp_out);
        assign mantissa_out = ~valid ? 23'b0 :
                (add_overflow ? rshift_mantissa_out : lshift_mantissa_out);
endmodule

module lod(in, p, v);
        parameter N=32;
        localparam W = $clog2(N);
        input [N-1:0] in;
        output [W-1:0] p;
        output v;

        generate
                // 입력이 2bit 일 경우 단순 비교
                if (N==2) begin
                        assign v = in[1] | in[0];      // 1의 존재 여부를 반환
                        assign p[0] = ~in[1] & in[0];           // 선행 0의 개수를 반환
                end
                // 입력이 2bit 이상일 경우 재귀적으로 나누어 처리
                else begin
                        wire [W-2:0] pL, pH;
                        wire vL, vH;
                        lod #(N>>1) u1 (in[(N>>1)-1:0], pL, vL);        // 우측 N/2 bit 재귀
                        lod #(N>>1) u2 (in[N-1:(N>>1)], pH, vH);        // 좌측 N/2 bit 재귀
                        assign v = vH | vL;          // 1의 존재 여부를 반환
                        assign p = vH ? {1'b0, pH} : {vL, pL};          // 선행 0의 개수를 반환
                end
```

```
        endgenerate
endmodule
```

<모듈별 기능 분석>

fp_addsub :

32비트 in_a, in_b를 각각 1비트 부호, 8비트 지수, 23비트 가수로 분해한다. 분해된 각 값은 sign_a/b, exp_a/b, frac_a/b에 할당되며 각각 계산된다.

is_zero_a/b를 통해 입력받은 a, b가 0인지를 확인한다. 이 결과에 따라 추가되는 hidden bit가 달라진다. 입력값이 0인 경우에는 0을, 그렇지 않은 경우에는 1을 추가해 1.xxx형태로 정규화해 두 입력 a, b의 23비트 가수부를 각각 24비트 수인 sig_a/b로 확장한다.

magnitude_a_gt_b를 이용해 절댓값이 큰 수를 고른다. 이때 우선 지수를 기준으로 판단하고, 지수가 동일한 경우 가수를 기준으로 판단한다. 두 수 중 절댓값이 큰 수의 지수와 24비트로 확장된 가수는 각각 exp/sig_ge에, 작은 수의 지수와 동일하게 24비트로 확장된 가수는 각각 exp/sig_lt에 할당된다.

두 수의 지수 차이를 계산 후 작은 수의 지수를 시프트해 큰 수의 지수와 통일한다.
sub, 입력 a, b의 부호를 종합적으로 고려해 is_add를 결정한다.
is_add의 결과에 따라 가수부끼리의 덧셈/뺄셈을 진행한다. 이때 각 가수에 0을 덧대 25비트로 변형해 연산 후 발생할 수 있는 오버플로우를 예방한다.

연산 후 지수는 절댓값이 큰 지수의 것으로 고정되고, 부호는 가수의 결과에 따라 0이면 0을, 그렇지 않을 경우 절댓값이 큰 입력의 실질적 부호(연산 부호 sub을 고려한 실질적 부호)를 선택한다.

fp_addsub가 수행된 후 얻는 값은 1. 부호 2. 통일된 지수 3. 연산이 끝난 25비트 가수의 세가지이다. 이 중 지수와 가수를 fp_norm을 이용해 정규화한다. (fp_norm의 기능은 아래에서 다시 서술한다.)

부호와 정규화된 지수와 가수를 재조립해 출력으로 재구성한다.

"입력된 두 부동소수점 수를 부호/지수/가수로 나눠 계산 전 단위를 통일하고, 계산한 후 정규화해 결과를 되돌려주는 회로"이다.

fp_norm :

해당 모듈은 통일된 지수, 25비트로 확장된 가수를 입력받아 각각을 정규화해 반환한다.
입력받은 가수의 MSB를 통해 덧셈 오버플로우를 확인한다. 24번째 비트가 1인 경우 자릿수가 올라감에 따른 오버플로우가 발생한 상태라고 가정한다. 오버플로우의 발생 여부에 따라 동작은 둘로 나뉘며 상세한 내용은 다음과 같다.

1. 덧셈 오버플로우가 발생한 경우의 오른쪽 시프트 정규화를 수행하며 상세한 내용은 다음과 같다. 지수를 1 증가시킨다. 그 후 가수부를 1만큼 오른쪽으로 시프트한 후 23비트만을 추출해 출력한다.

2. 덧셈 오버플로우가 발생하지 않은 일반적인 경우 왼쪽 시프트 정규화를 수행하며, 상세한 내용은 다음과 같다. 입력받은 가수부에 0을 덧대어 32비트로확장한다. lod 모듈을 호출해 가수 내부에서 가장 앞에 있는 1의 위치를 탐색한다. (lod 모듈의 설명에서 다시 언급하지만 lod는 재귀 호출을 통해 이진 탐색을 구현하는 모듈이다.) 1앞의 0의 개수는 lz_count_extended에 저장되고, 가수가 0인지 여부는 valid에 저장된다. 최상위 1이 가장 앞으로 오도록 시프트한다. (0을 덧대는 등의 과정에서 정규화되지 않은 가수가 이 시프트 연산에 의해 1.xxx형태로 정규화된다.) 이 때 가수를 시프트하는만큼 지수를 감소시킨다.

valid가 0인 경우 (가수 전체가 0인 경우 결과를 0으로 설정한다. valid가 0이 아닌 경우 덧셈 오버플로우의 결과에 따라 오버플로우가 발생했다면 1의 동작 결과를, 그렇지 않다면 2의 동작 결과를 채택하여 반환한다.

"25비트 형태로 들어온 가수를 부호와 오버플로우 등을 고려해 정규화해 반환하는 회로"이다.

lod :

입력이 2비트이고 MSB가 1이면 0을, LSB가 1이면 1을 반환한다. (01 >> p=1, 10, 11, 00 >> p=0)

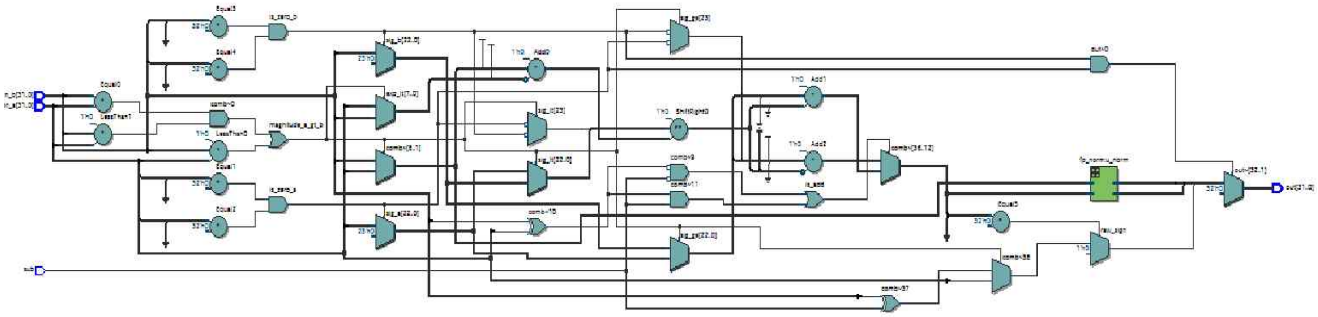
입력이 2비트가 아닌 경우 전체 비트에 대하여 하위 절반과 상위 절반에 대해 각각 lod를 실행해 재귀적으로 최상위 1의 인덱스를 탐색한다.

재귀적으로 최상위 1의 인덱스를 탐색하는 과정에서 p는 최상위 1의 인덱스(=1에 앞서는 0의 개수)를 표현하는 배열이 된다.

“입력받은 전체 비트에 대하여 최상위 1의 인덱스를 찾는 이진탐색 회로”이다.

3. 시각화

<회로도>
작성된 코드를 RTL 뷰어를 통해 확인한 모습은 다음과 같다.

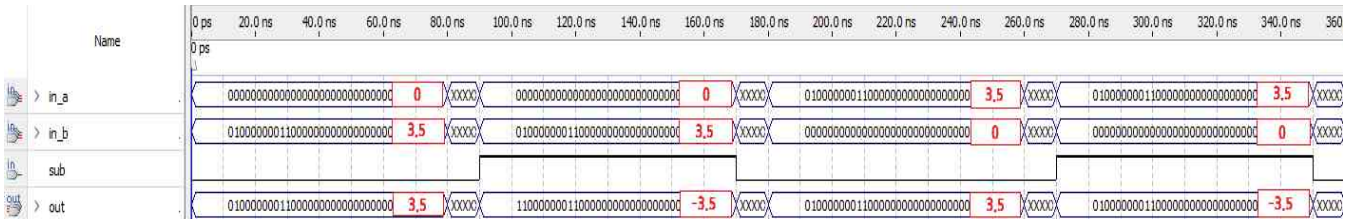


4. 실행 결과

(각 테스트 케이스에 대한 작성 기준 및 실행 결과를 사진 및 글로 작성했음)

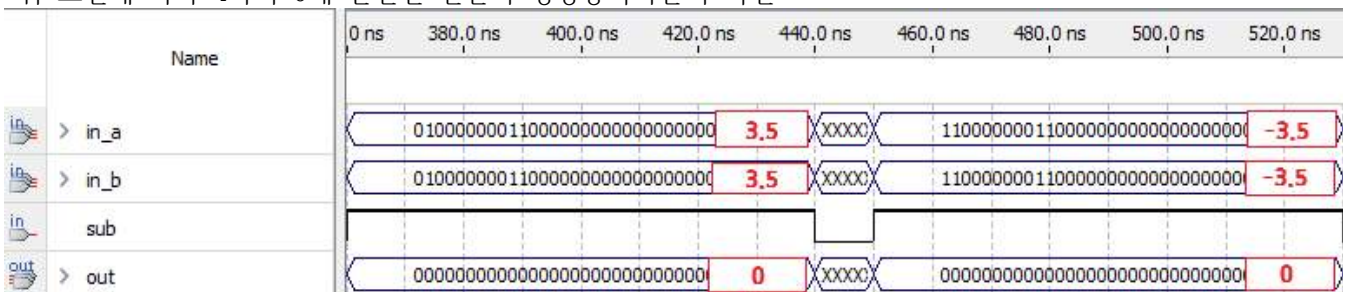
1. 기본적인 연산 확인 16가지(사진1~4)
-입력 a의 양/음수 여부
-입력 b의 양/음수 여부
-입력 sub의 부호 여부
-절댓값이 큰 입력의 선택 여부

위 4가지 조건에 따라 16가지 기본 연산이 정상동작하는지 확인



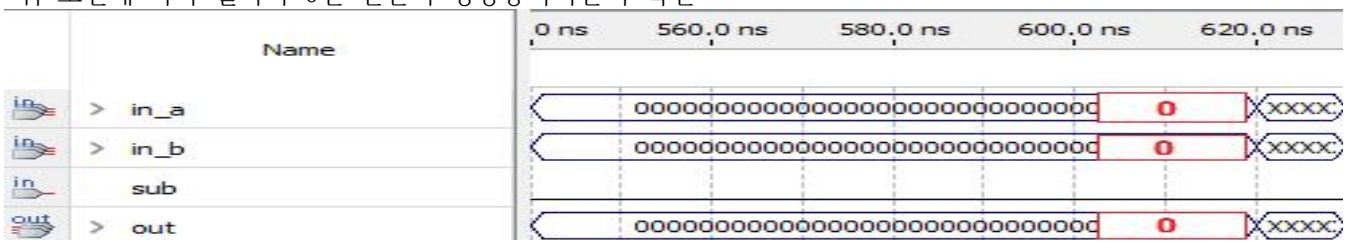
2. 입력 0 처리 확인 4가지(사진5)
-입력 a의 0 여부
-입력 b의 0 여부

위 조건에 따라 4가지 0에 관련된 연산이 정상동작하는지 확인



3. 같은 수 뺄셈 연산 확인(사진6)
-양수/음수 동일값 뺄셈 연산 여부

위 조건에 따라 결과가 0인 연산이 정상동작하는지 확인



4. 입력이 모두 0인 경우(사진)
-입력이 모두 0인 경우

5. 분석

fp_addsub.v는 두 개의 32비트 단일 정밀도 부동 소수점 수에 대한 덧셈/뺄셈을 수행한다. 32비트 형식으로 입력받은 수를 각각 부호/지수/가수로 나눠 각각 처리하며 이 과정에서 처리 결과에 대한 표현 및 비트에 대한 정규화를 수행한다.
코드의 검증을 위해 부동소수점 계산 검증을 위해 준비한 임의의 숫자와, 실수를 32bit 수로 변환하는 사이트를 준비하고, numerical하게 계산된 결과와 analytic하게 계산한 결과를 바탕으로 작성된 fp_addsub.v의 동작을 검증했다. 처음 배포받은 fp_addsub.v에서 확인할 수 있었던 문제는 다음과 같다.

1. 절댓값이 큰 연산자의 부호만을 판단했기 때문에 “큰 음수 - 작은 음수”, “작은 양수 - 큰 양수” , “0관련 연산” 등의 연산이 진행될 때 부호 처리가 적절하지 않았다.
2. 일의 자리에서 십의 자리로 숫자가 넘어가는 경우, 이에 대한 비트 시프트가 구현되지 않아 계산 결과가 의도한 결과와 다르게 출력되었다.
3. 입력이 모두 0인 경우 연산 과정에서 오류가 발생했다. 기존 코드는 입력이 0인 경우에도 정규화를 위해 hidden bit 1을 추가하고, 이 과정에서 0과 관련된 연산에 오류가 발생했음을 확인했다.
4. 가수부가 0이 되는 경우 연산 과정에서 오류가 발생했다. fp_norm을 통한 정규화 과정에서 가장 앞에 있는 1을 찾는 도중 가수부가 0인 연산에 관련된 연산 오류가 발생했음을 확인했다.

위와 같이 관측된 문제를 해결할 수 있을 것이라고 생각했다. 협업을 통해 확인된 오류를 수정했으며, 수정된 사항에 대한 구체적인 내용은 아래와 같다.

1. 가수부가 0일 때 부호를 0으로 선택하는 코드를 추가하고, b가 더 커 연산 결과의 부호를 뒤집어야 하는 상황을 반영할 수 있도록 수정했다.
2. 자릿수가 넘어가 오버플로우가 발생하는 경우 fp_norm에서 오버플로우 비트를 감지하도록 설계를 변경 후 이에 맞춰 지수와 가수를 재조정하도록 수정했다.
3. 입력이 0인 경우 hidden bit 없이 가수부를 0으로 처리하도록 수정하고, 입력이 모두 0인 경우 최종 출력 또한 0이 되도록 수정했다.
4. 가수부가 0인지 여부를 판단할 수 있도록 valid 변수를 추가하고, 이를 통해 가수부가 0인지를 확인하고 이에 대한 예외처리를 수행하도록 구현해 연산 오류를 해결했다.

6. 평가 및 미래과제

fp_addsub.v를 분석하고, 코드를 검증하는 과정에서 확인할 수 있었던 문제점을 정의 후, 해당 부분에 대한 수정을 진행했다. 검증 과정에서 확인한 문제들에 대해서는 적절한 처리가 이루어져 적정 수준으로 동작하는 코드로 개선되었다고 평가할 수 있다.
개선된 fp_addsub.v를 기반으로 다음과 같은 발전 과제를 생각해볼 수 있다.

1. IEEE 754 기반으로 설계된 가감산기에는 소수점 절사에 대한 구체적인 처리가 부족하다. IEEE 754에서 정의되는 소수점 절사에 대한 방법론을 공부 후, 이를 적용해 소수점을 보다 정밀하게 처리할 수 있도록 개선할 수 있을 것이다.
2. 개선한 코드가 하드웨어 컴포넌트, 전력 사용 등의 측면에서 완벽하게 최적화되어있다고 판단할 수 없다. 이에 따라 향후 전체적인 코드 최적화를 시도해볼 수 있다.
3. 향후 64bit 의 가감산기로 확장하거나, 곱셈 로직의 병합, 다양한 오버/언더 플로우 상황에 대한 대응 등 추가적인 기능의 확장을 시도해볼 수 있을 것이다.