MIPS32: Pipelined CPU with FPU

Seungwan Noh

Pusan National University

Department of Electronics Engineering

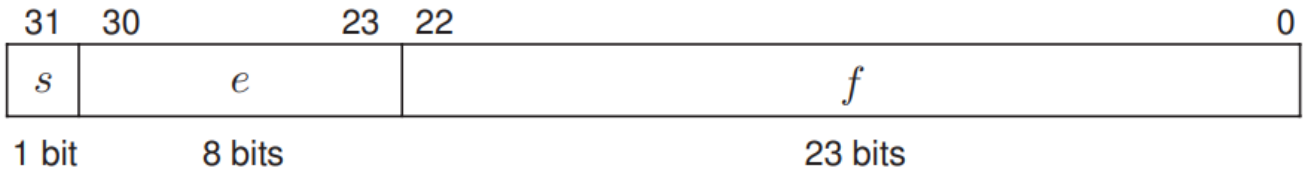
Contents

- IEEE 754 Floating-Point Data Formats
- FPU
  - ✓FADD
  - ✓FMUL
  - ✓FDIV
  - ✓FSQRT
- MIPS32 Pipelined CPU with FPU in Verilog HDL
- Simulation
- References

IEEE 754 Floating-Point Data Formats

- The IEEE 754 Standards defines ***float*** and ***double***.
- An IEEE 754 single-precision floating-pint number consists of three parts: ***s, e, f***
  - ✓1-bit **s** indicates the sign of number
  - ✓8-bit **e** is the biased exponent with the bias of 127
  - ✓23-bit **f** is the significand

- The value of a single-precision floating-point number.
  - ✓Normalized: If 0 < e < 255, V = (-1)^s x 1.f x 2^(e - 127)
  - ✓Denormalized: If e = 0 and f ~= 0, V = (-1)^s x 0.f x 2^(-126)
  - ✓If e = 0 and f = 0, V = (-1)^s x 0
  - ✓If e = 255 and f = 0, V = (-1)^s∞
  - ✓If e = 255 and f ~= 0, V = NaN



IEEE 754 Floating-Point Data Formats

- For normalized number, the 1 of 1.f is not stored in the 32-bit data; it is called a **hidden bit**.

- Normalized numbers ($V = (-1)^s \times 1.f \times 2^{e-127}$):

  $1\_01111110\_10000000000000000000000 = -1.1 \times 2^{126-127} = -0.75$
  $0\_00000001\_00000000000000000000000 = +1.0 \times 2^{1-127} = +2^{-126}$
  $0\_11111110\_00000000000000000000000 = +1.0 \times 2^{254-127} = +2^{127}$
  $0\_11111110\_11111111111111111111111 = +(2 - 2^{-23}) \times 2^{254-127}$

- Zero, infinity, and NaN:

  $0\_00000000\_00000000000000000000000 = +0$
  $1\_00000000\_00000000000000000000000 = -0$
  $0\_11111111\_00000000000000000000000 = +\infty$
  $1\_11111111\_00000000000000000000000 = -\infty$
  $0\_11111111\_10000000000000000000000 = \text{NaN}$
  $1\_11111111\_00000100001100000000000 = \text{NaN}$

- Denormalized numbers ($V = (-1)^s \times 0.f \times 2^{-126}$):

  $0\_00000000\_10000000000000000000000 = +0.1 \times 2^{-126} = +2^{-127}$
  $0\_00000000\_00000000000000000000001 = +2^{-23} \times 2^{-126} = +2^{-149}$

Converting Floating-Point Number to Integer

- The range of a 32-bit 2's complement integer *d*
  - ✓-2^31 <= *d* <= 2^32 + 1
- 32-bit single-precision floating-point number has a wider range.
- Many floating-point numbers can't be converted to integer.
- Floating-number to integer algorithm
  - ✓Attach an 8-bit 0 to 1f (the point of 1.f is shifted to the right by 32 bits).
  - ✓Shift It to the right by 127 + 31 – e bits.
  - ✓If s = 1, invert and add 1.

Converting an Integer to a Floating Number

- Any integer can be converted to a floating number.
- But integers have more significant bits than float numbers.
- The precision may be lost after the conversion.

$$
\begin{aligned}
d &= 00011111111111111111111111111111._2 && \text{(to shift “.” to left by 31 bits)} \\
&= 0.00111111111111111111111111111111_2 \times 2^{31} && \text{(to } 1.f_a \text{ format)} \\
&= 1.11111111111111111111111111111000_2 \times 2^{31-3} && \text{(3 bits shifted)} \\
&\approx 1.11111111111111111111111_2 \times 2^{(127+31-3)-127} && \text{(IEEE float format)} \\
&= (-1)^s \times 1.f \times 2^{e-127} && \text{(IEEE float format)}
\end{aligned}
$$

FADD

- Floating-point addition algorithm
- Alignment
  - ✓Make sure *a* and *b* have the same exponent to perform calculation.
  - ✓Shifted bits are treated as the IEEE 754 standard.
    - Guard bit, the leftmost bit
    - Round bit, the next bit
    - Sticky bit, the rest of the bits
- Calculation
  - ✓The result must be larger than to (but <4), we use an additional bit to the most significant bit position during the calculation.
- Normalization
  - ✓The calculated result of the significand must be converted to *1.f* format.
  - ✓Find the first 1 from the left to determine the shift amount.

FADD

- Special case
- If *a* or *b* is and *NaN*, the result *s* will be *NaN.*
- If the two numbers are infinities, the result are below

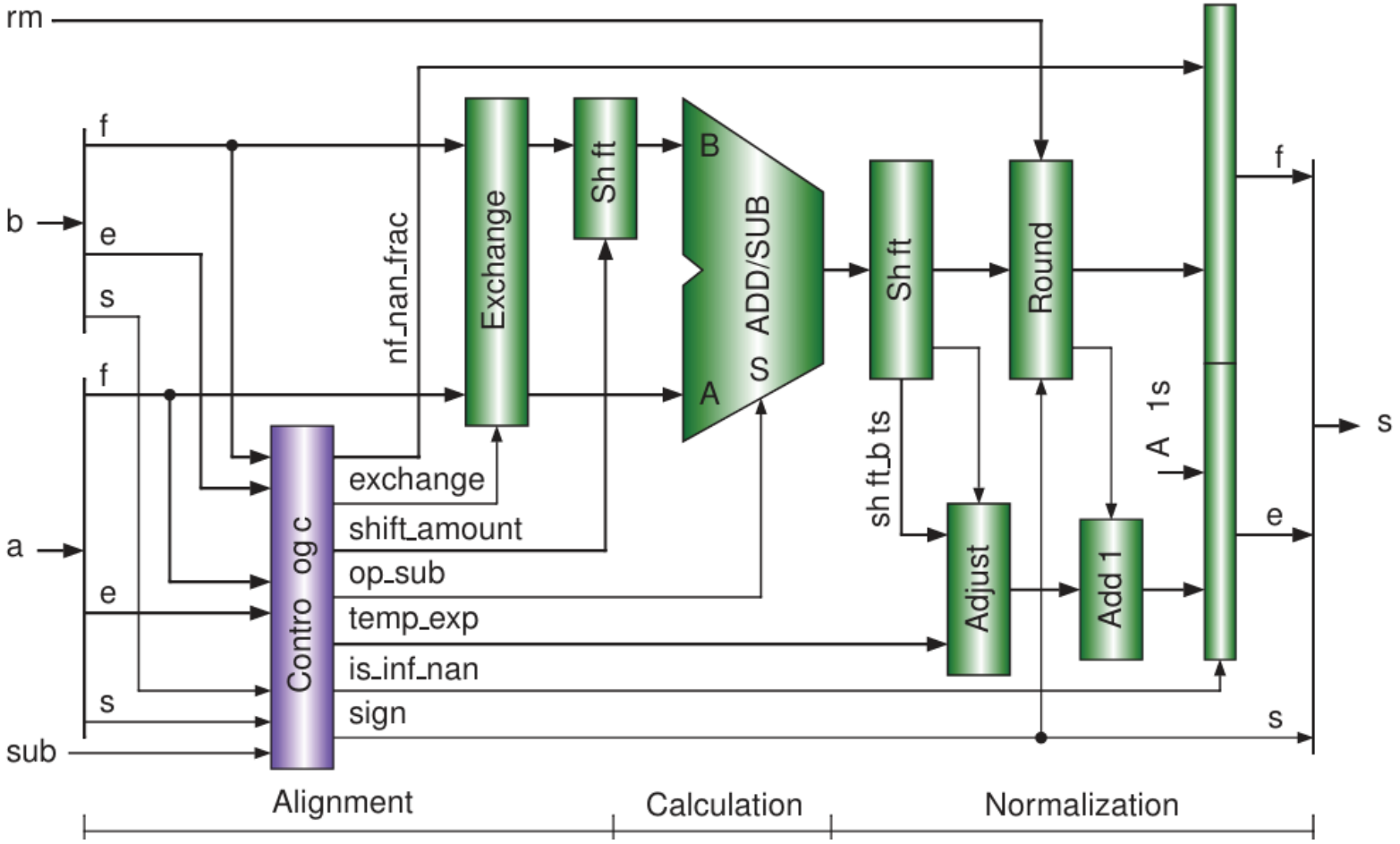
sub	*a*	*b*	*s*	Comment
0	$+\infty$	$+\infty$	$+\infty$	$(+\infty)+(+\infty)$
0	$-\infty$	$-\infty$	$-\infty$	$(-\infty)+(-\infty)$
1	$+\infty$	$-\infty$	$+\infty$	$(+\infty)-(-\infty)$
1	$-\infty$	$+\infty$	$-\infty$	$(-\infty)-(+\infty)$
0	$+\infty$	$-\infty$	NaN	$(+\infty)+(-\infty)$
0	$-\infty$	$+\infty$	NaN	$(-\infty)+(+\infty)$
1	$+\infty$	$+\infty$	NaN	$(+\infty)-(+\infty)$
1	$-\infty$	$-\infty$	NaN	$(-\infty)-(-\infty)$

FADD

- Three additional bits ***grs*** are used for significand calculation.
- The result of the float add/sub may be overflow.
- The rules of the IEEE 754 when overflow occurred:
  - ✓Round to nearest
  - ✓Round toward zero
  - ✓Round toward –infinite
  - ✓Round toward +infinite
- The result will be either infinity or maximum.

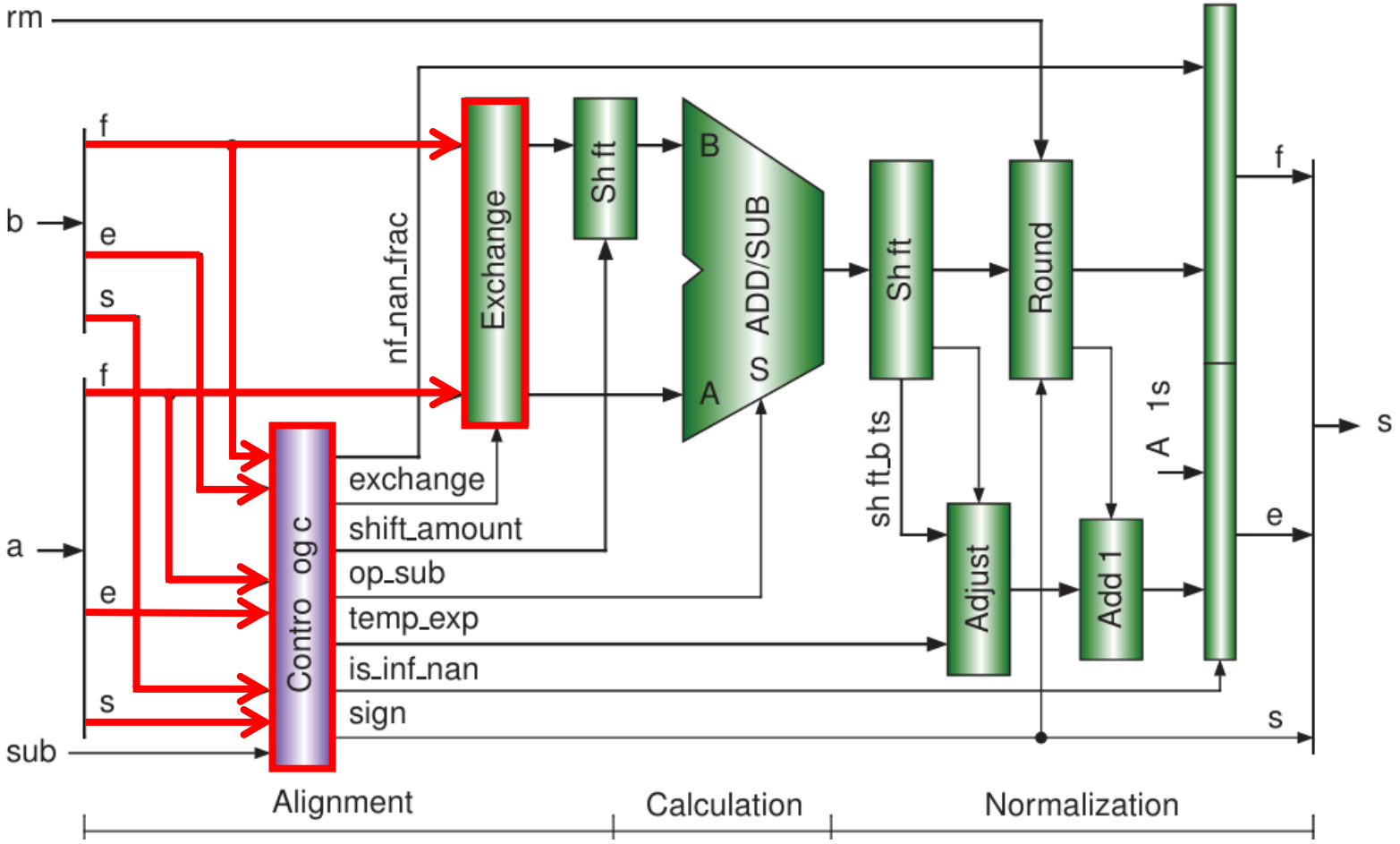
FADD

- Block diagram of a floating-point adder



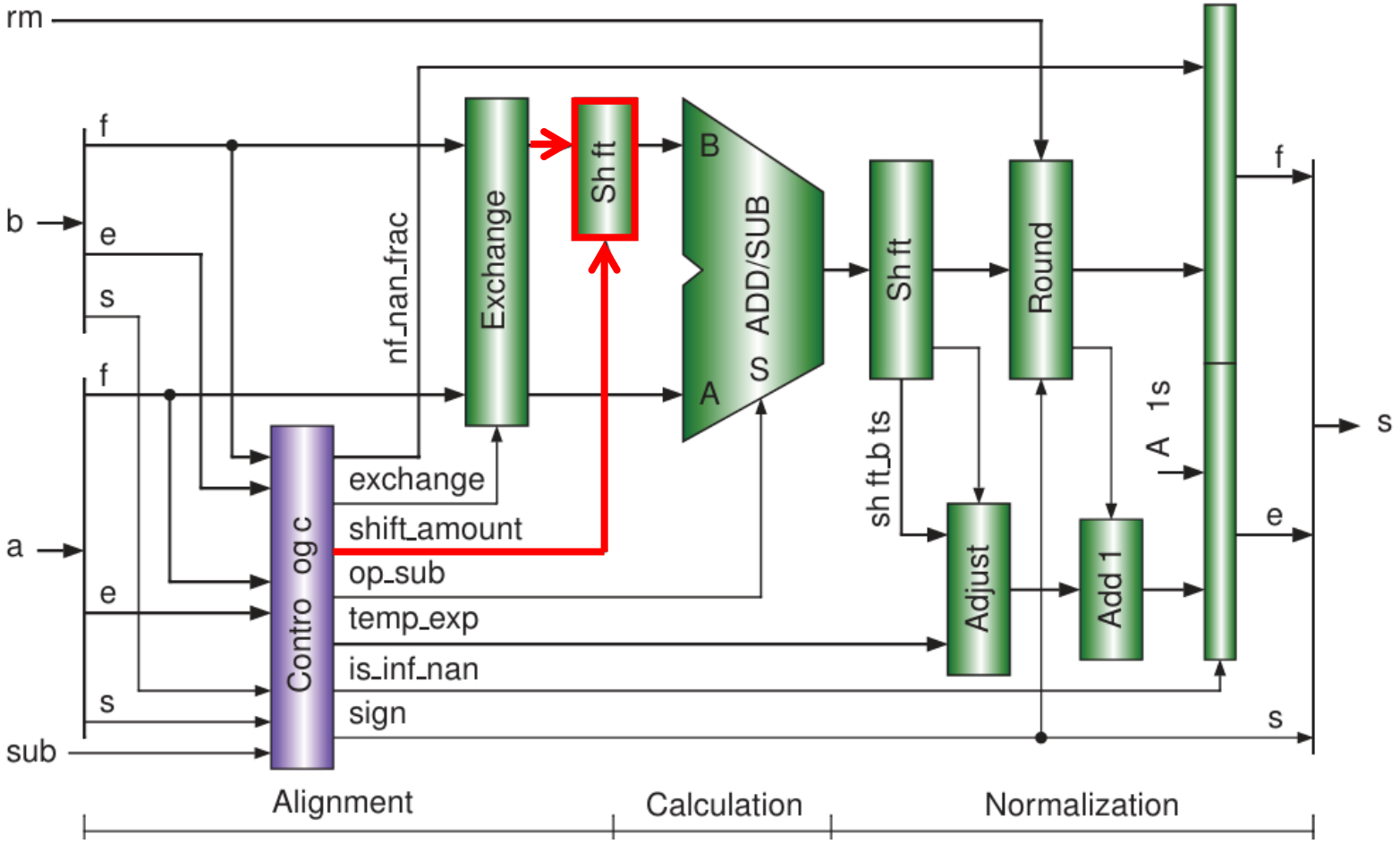
FADD

- Block diagram of a floating-point adder



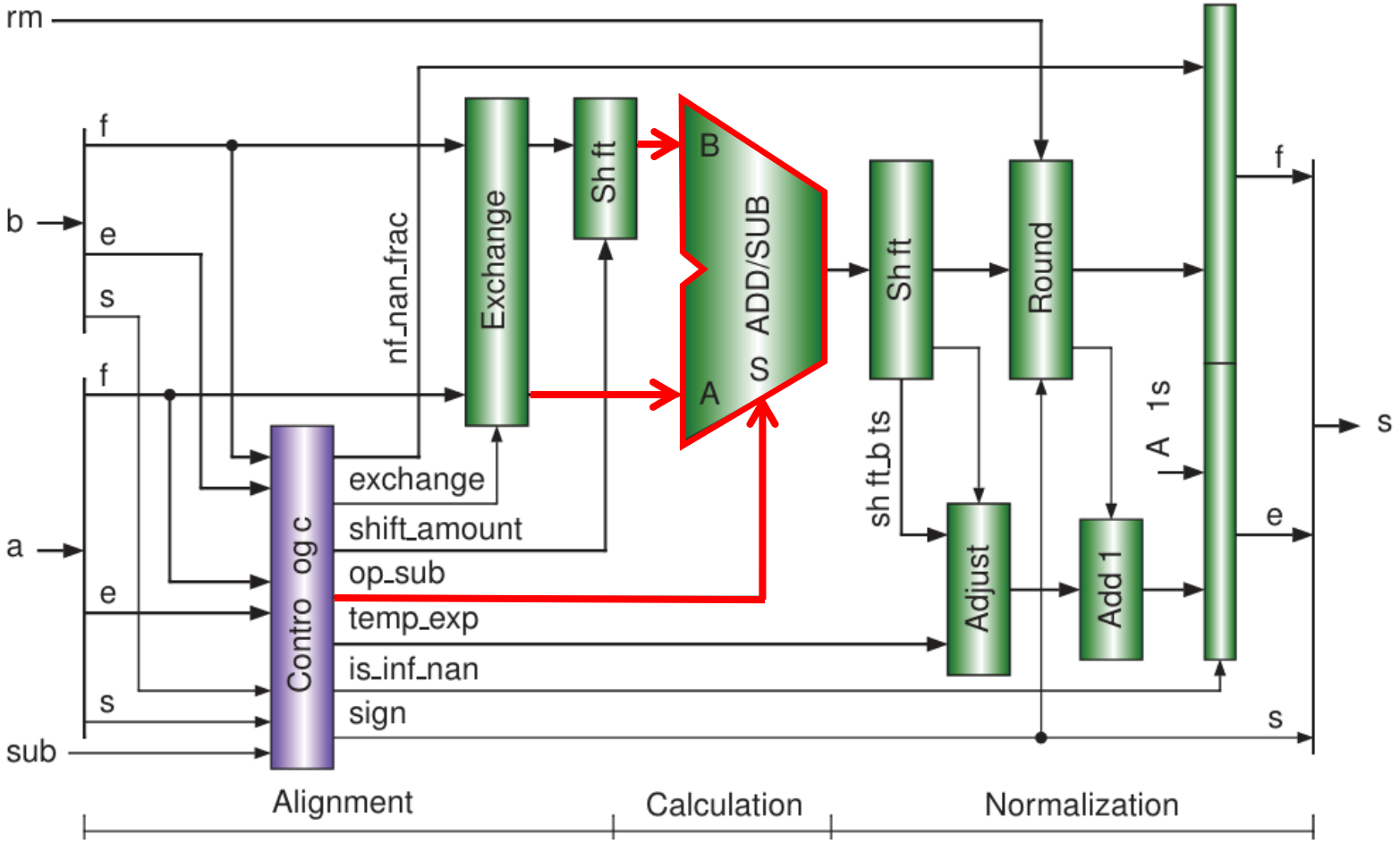
FADD

- Block diagram of a floating-point adder



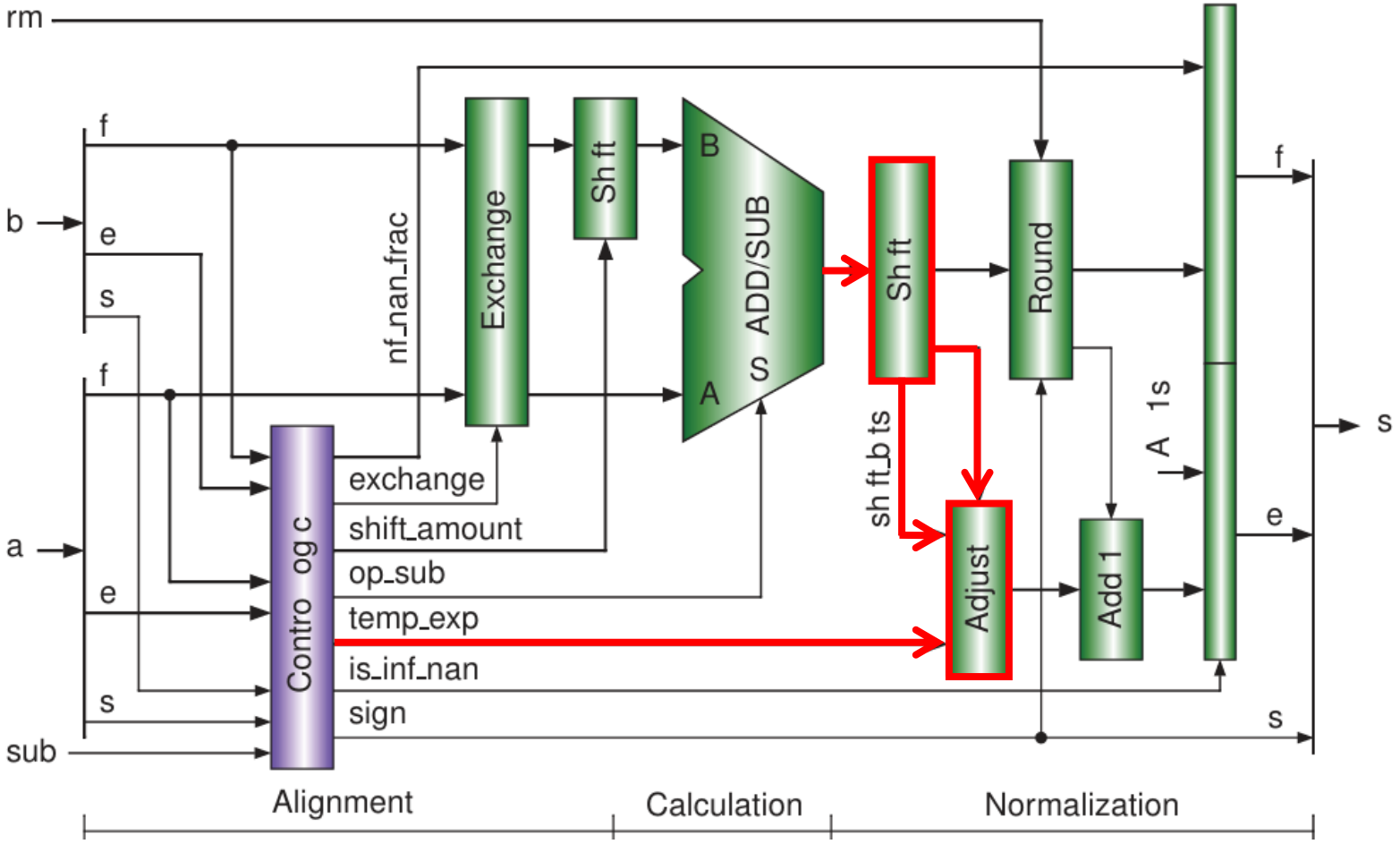
FADD

- Block diagram of a floating-point adder



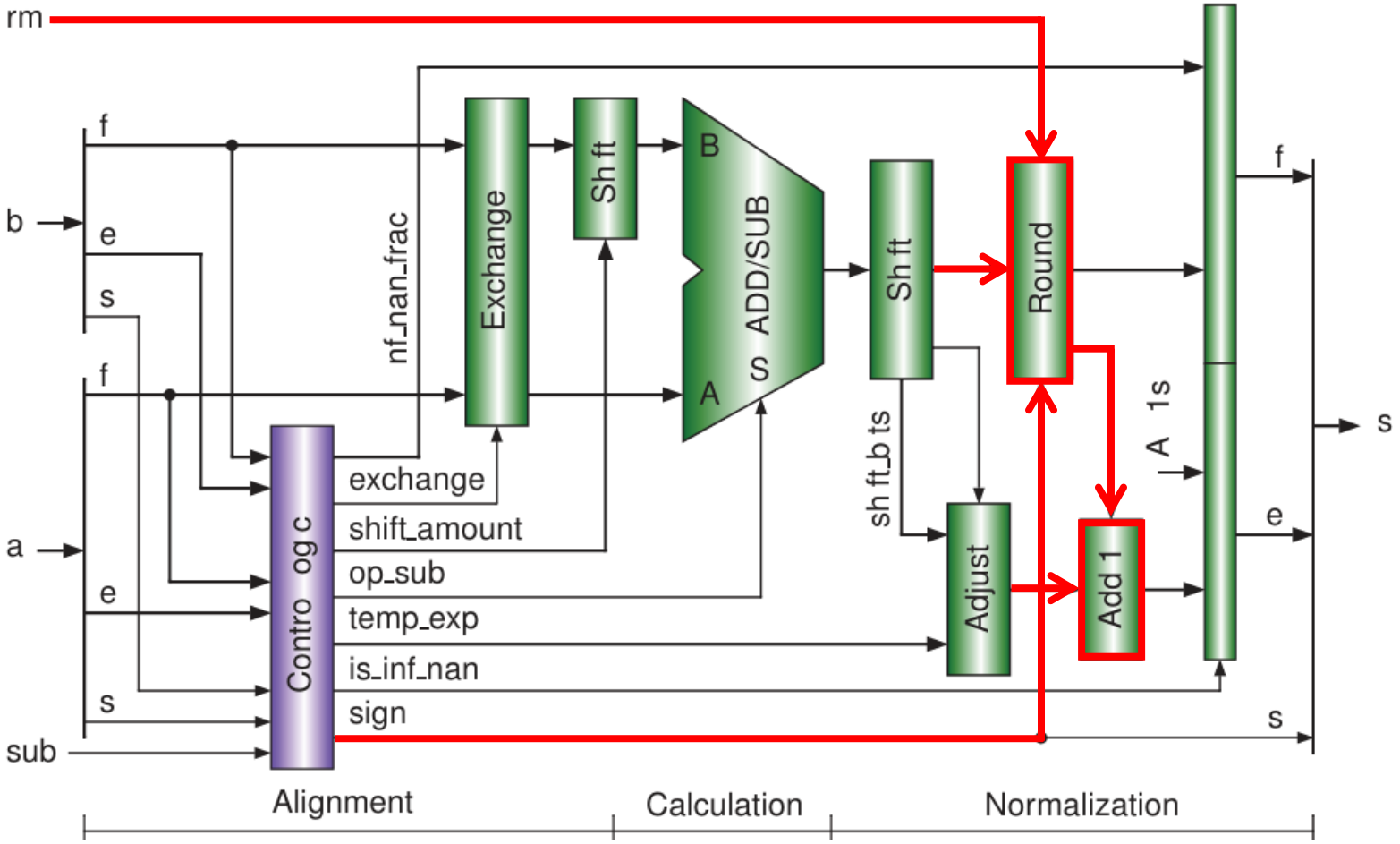
FADD

- Block diagram of a floating-point adder



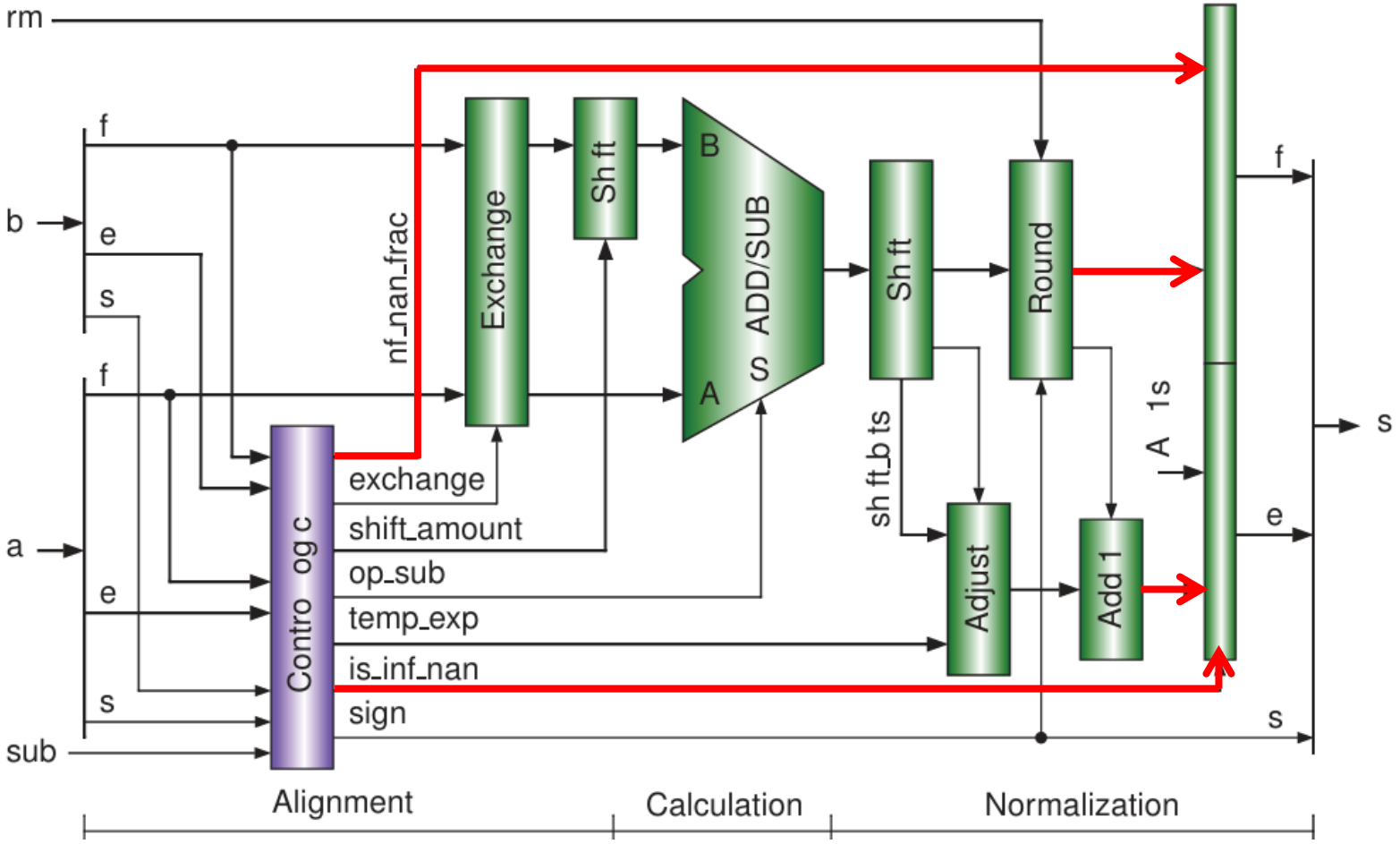
FADD

- Block diagram of a floating-point adder



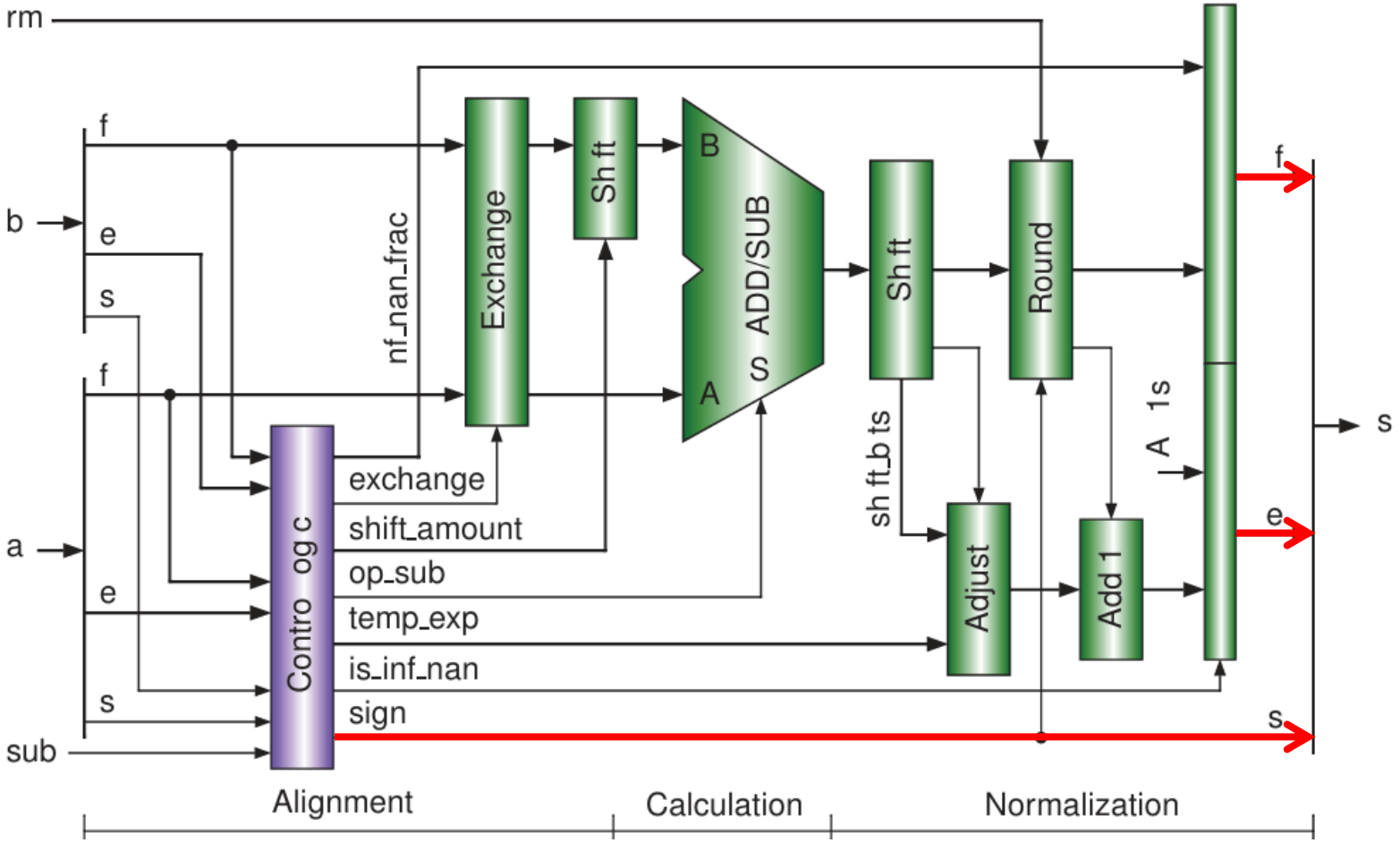
FADD

- Block diagram of a floating-point adder



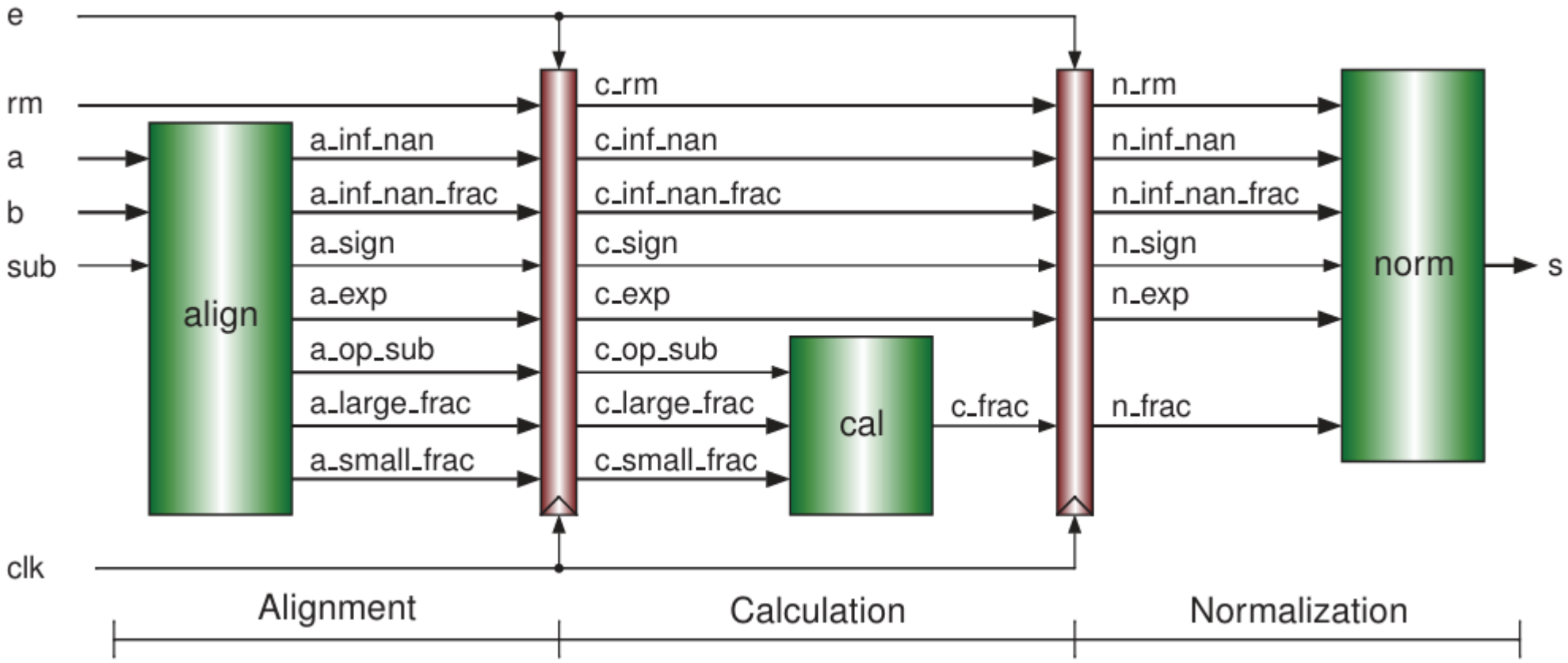
FADD

- Block diagram of a floating-point adder



FADD

- Pipelined floating point adder Verilog implementation.
- Alignment stage, calculation stage, and normalization stage with two pipelined registers.



FADD

- **Alignment**
- exchange is set when b is larger than a.
- Assign *fp_large* and *fp_small* based on exchange signal.
- Hidden bit is set if exponent is not zero.
- Adds 1 hidden bit to et 24-bit fraction.

```
13      wire            exchange    = (b[30:0] > a[30:0]);
14      wire    [31:0]  fp_large    = exchange ? b : a;
15      wire    [31:0]  fp_small    = exchange ? a : b;
16      wire            fp_large_hidden_bit = |fp_large[30:23];
17      wire            fp_small_hidden_bit = |fp_small[30:23];
18      wire    [23:0]  large_frac24    = {fp_large_hidden_bit, fp_large[22:0]};
19      wire    [23:0]  small_frac24    = (fp_small_hidden_bit, fp_small[22:0]);
```

FADD

- **Alignment**
- The output exponent is taken from the larger operand.
- The output sign is set to sign of b if operation is not a sub or a.
- Decide if the operation is sub in fraction level.

```
21      assign          temp_exp    = fp_large[30:23];
22      assign          sign        = exchange ? sub ^ b[31] : a[31];
23      assign          op_sub      = sub ^ fp_large[31] ^ fp_small[31];
```

FADD

- **Alignment**
- Check if exponents is all 1s and if fraction is all zeros.
- Identifies if each operand is inf or *NaN*.
- Result is inf either input is inf.
- Result is *NaN* if either input is *NaN*, or both are inf with opposite signs and substraction.

```
25    wire          fp_large_expo_is_ff = &fp_large[30:23]; // exp == 0xff
26    wire          fp_small_expo_is_ff = &fp_small[30:23];
27    wire          fp_large_frac_is_00 = ~|fp_large[22:0]; // frac == 0x00
28    wire          fp_small_frac_is_00 = ~|fp_small[22:0];
29    wire          fp_large_is_inf = fp_large_expo_is_ff & fp_large_frac_is_00;
30    wire          fp_small_is_inf = fp_small_expo_is_ff & fp_small_frac_is_00;
31    wire          fp_large_is_nan = fp_large_expo_is_ff & ~fp_large_frac_is_00;
32    wire          fp_small_is_nan = fp_small_expo_is_ff & ~fp_small_frac_is_00;
33    assign        s_is_inf    = fp_large_is_inf | fp_small_is_inf;
34    wire          s_is_nan    = fp_large_is_nan | fp_small_is_nan |
35                                ((sub ^ fp_small[31] ^ fp_large[31]) &
36                                fp_large_is_inf & fp_small_is_inf);
```

FADD

- **Alignment**
- Set *NaN* fraction and ensure it's not zero with a leading 1.
- If the result is *NaN*, assign the *nan_frac* as output otherwise 0.
- Check *fp_large* is normalized and *fp_small* is denormalized.
- Denormalized numbers don't have a hidden bit.
- If *fp_small* is denormal, reduce the shift by 1.

```
38    wire    [22:0]  nan_frac    = (a[21:0] > b[21:0]) ? {1'b1,a[21:0]} : {1'b1,b[21:0]};
39    assign          inf_nan_frac   = s_is_nan ? nan_frac : 23'h0;
40    wire    [7:0]   exp_diff    = fp_large[30:23] - fp_small[30:23];
41    wire            small_den_only = (fp_large[30:23] != 0) & (fp_small[30:23] == 0);
42    wire    [7:0]   shift_amount = small_den_only ? exp_diff - 8'h1 : exp_diff;
43    wire    [49:0]  small_frac50 = (shift_amount >= 26) ?
44                                   {26'h0,small_frac24} :
45                                   {small_frac24,26'h0} >> shift_amount;
46    assign          small_frac27 = {small_frac50[49:24],|small_frac50[23:0]};
```

FADD

- **Alignment**
- Fraction alignment step.
- *small_frac24* is the 24-bit significand (including hidden bit if normalized).
- *small_frac50[49:24]* → Top 26 bits (G + R + 24-bit significand).
- |*small_frac50[23:0]* → Sticky bit from all lower discarded bits.

```
38      wire    [22:0]  nan_frac    = (a[21:0] > b[21:0]) ? {1'b1,a[21:0]} : {1'b1,b[21:0]};
39      assign          inf_nan_frac   = s_is_nan ? nan_frac : 23'h0;
40      wire    [7:0]   exp_diff    = fp_large[30:23] - fp_small[30:23];
41      wire            small_den_only = (fp_large[30:23] != 0) & (fp_small[30:23] == 0);
42      wire    [7:0]   shift_amount = small_den_only ? exp_diff - 8'h1 : exp_diff;
43      wire    [49:0]  small_frac50 = (shift_amount >= 26) ?
44                                     {26'h0,small_frac24} :
45                                     {small_frac24,26'h0} >> shift_amount;
46      assign          small_frac27 = {small_frac50[49:24],|small_frac50[23:0]};
```

FADD

- **Calculation**
- Add 1 zero MSB to ensure no overflow when subtracting.
- Add 3 zero LSBs to match the precision of *small_frac27*.
- If *op_sub* is set do subtraction or do addition.

```
1   module fadd_cal (
2       input   [23:0]  large_frac24,
3       input           op_sub,
4       input   [26:0]  small_frac27,
5       output  [27:0]  cal_frac
6   );
7       wire    [27:0]  aligned_large_frac  = {1'b0,large_frac24,3'b000};
8       wire    [27:0]  aligned_small_frac  = {1'b0,small_frac27};
9       assign          cal_frac            = op_sub ? aligned_large_frac - aligned_small_frac :
10                                                     aligned_large_frac + aligned_small_frac;
11  endmodule
```

FADD

- **Normalization**
- Determine how many leading zeros the fraction has.
- Detect how many bits we need to left-shift the fraction to normalize it.

```
10      wire    [26:0]  f4, f3, f2, f1, f0;
11      wire    [4:0]   zeros;
12      assign          zeros[4] = ~|cal_frac[26:11];
13      assign          zeros[3] = ~|f4[26:19];
14      assign          zeros[2] = ~|f3[26:23];
15      assign          zeros[1] = ~|f2[26:25];
16      assign          zeros[0] = ~f1[26];
17      assign          f4 = zeros[4] ? {cal_frac[10:0],16'b0} : cal_frac[26:0];
18      assign          f3 = zeros[3] ? {f4[18:0],8'b0} : f4;
19      assign          f2 = zeros[2] ? {f3[22:0],4'b0} : f3;
20      assign          f1 = zeros[1] ? {f2[24:0],2'b0} : f2;
21      assign          f0 = zeros[0] ? {f1[25:0],1'b0} : f1;
```

FADD

- **Normalization**
- If MSB is set, overflow in significand, shift right.
- If not, normalize, shift left and decrement exponent accordingly.
- If result is subnormal or underflow, force exponent to 0 and *sll*.

```
22      reg     [26:0]  frac0;
23      reg     [7:0]   exp0;
24      always @(*) begin
25          if (cal_frac27) begin
26              frac0   = cal_frac[27:1];     // 1x.xxx xxx
27              exp0    = temp_exp + 8'h1;    //  1.xxx xxx
28          end else begin
29              if ( (temp_exp > zeros) && f0[26] ) begin // a normaized number
30                  exp0    = temp_exp - zeros;
31                  frac0   = f0;             // 01.xxx xxx
32              end else begin
33                  exp0    = 0;
34                  if (temp_exp != 0)        // (e - 127) = ((e - 1) - 126)
35                      frac0   = cal_frac[26:0] << (temp_exp - 8'h1);
36                  else
37                      frac0   = cal_frac[26:0];
38              end
39          end
40      end
```

FADD

- **Normalization**
- Looks at guard (bit 2), round (1), and sticky (0) bits.
- Based on rounding mode and sign, decides whether to add 1.
- Rounds the 23-bit significand with 1-bit leading hidden bit.
- Detects if exponent overflowed (became 0xff or more).

```
41      wire            frac_plus_1 =
42                          ~rm[1] & ~rm[0] &  frac0[2] & (frac0[1] |  frac0[0]) |
43                          ~rm[1] & ~rm[0] &  frac0[2] & ~frac0[1] & ~frac0[0]  & frac0[3] |
44                          ~rm[1] &  rm[0] & (frac0[2] |  frac0[1] |  frac0[0]) &  sign |
45                           rm[1] & ~rm[0] & (frac0[2] |  frac0[1] |  frac0[0]) & ~sign;
46      wire    [24:0]  frac_round = {1'b0,frac0[26:3]} + frac_plus_1;
47      wire    [7:0]   exponent    = frac_round[24] ? exp0 + 8'h1 : exp0;
48      wire            overflow    = &exp0 | &exponent;
```

FADD

- **Normalization**
- Normal case: build {sign, exponent, fraction}
- Specials: *NaN*, *inf*, signed zero, max value
- Overflow: depends on rounding mode and sign

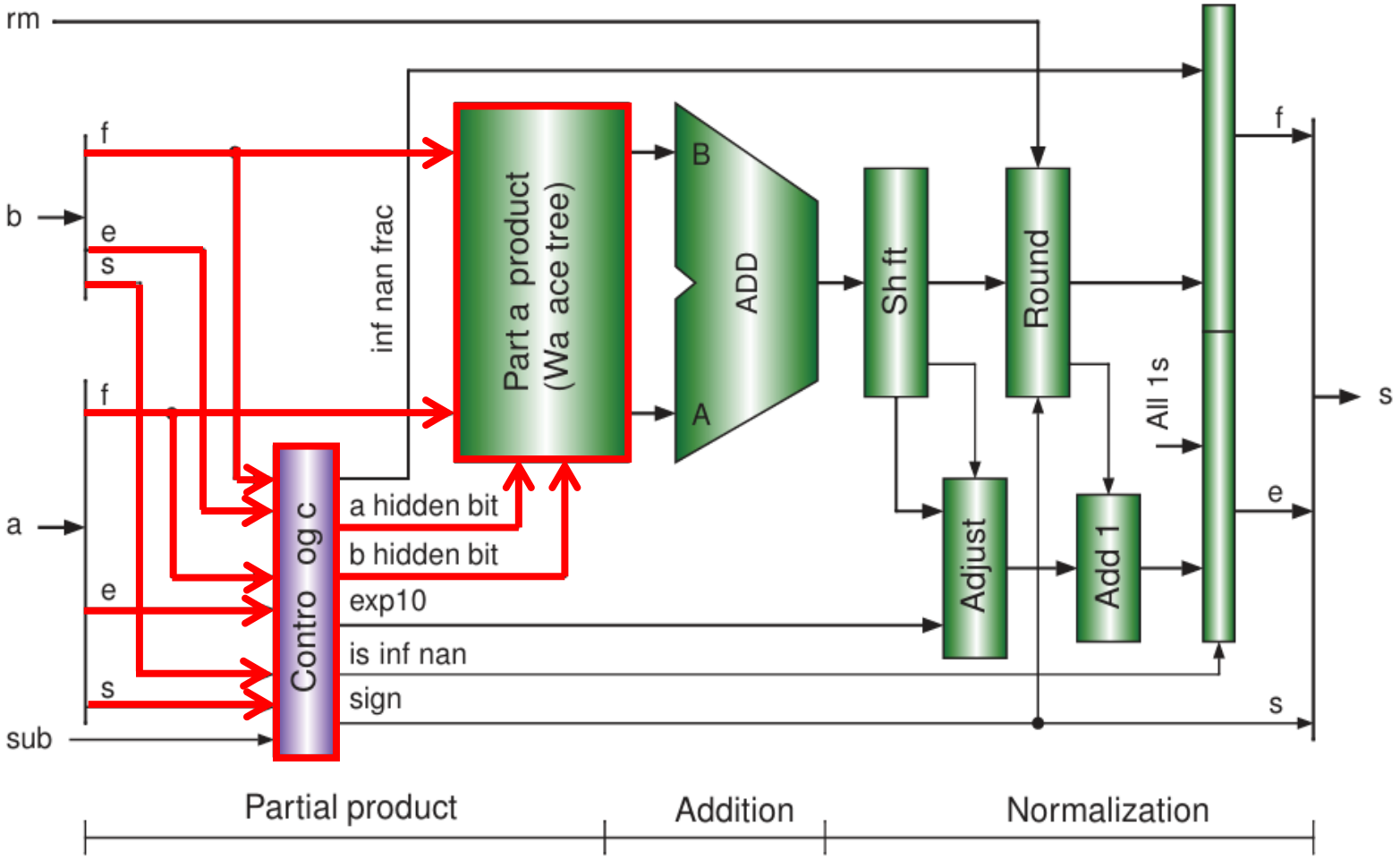
```
48    assign          s           = final_result (overflow, rm, sign, is_nan, is_inf, exponent,
49                                                frac_round[22:0], inf_nan_frac);
50    function    [31:0]  final_result;
51        input           overflow;
52        input   [1:0]   rm;
53        input           sign, is_nan, is_inf;
54        input   [7:0]   exponent;
55        input   [22:0]  fraction, inf_nan_frac;
56        casex ({overflow, rm, sign, is_nan, is_inf})
57            6'b1_00_x_0_x : final_result = {sign,8'hff,23'h000000};     // inf
58            6'b1_01_0_0_x : final_result = {sign,8'hfe,23'h7fffff};     // max
59            6'b1_01_1_0_x : final_result = {sign,8'hff,23'h000000};     // inf
60            6'b1_10_0_0_x : final_result = {sign,8'hff,23'h000000};     // inf
61            6'b1_10_1_0_x : final_result = {sign,8'hfe,23'h7fffff};     // max
62            6'b1_11_x_0_x : final_result = {sign,8'hfe,23'h7fffff};     // max
63            6'b0_xx_x_0_0 : final_result = {sign,exponent,fraction};    // nor
64            6'bx_xx_x_1_x : final_result = {1'b1,8'hff,inf_nan_frac};   // nan
65            6'bx_xx_x_0_1 : final_result = {sign,8'hff,inf_nan_frac};   // inf
66            default       : final_result = {sign,8'h00,23'h000000};     // 0
67        endcase
68    endfunction
```

FMUL

- Floating-point multiplication algorithm
- 2 normalized float numbers
  - ✓The result is a normalized number, if 1 <= *e_c* <= 254.
  - ✓The result is a denormalized number, if *e_c* < 1 and result is larger or equal to 2^(-149).
  - ✓The result is a zero otherwise.
- 1 normalized and 1 denormalized float numbers
  - ✓The largest absolute is a normalized number.
  - ✓The smallest absolute exceed the range; it represented by a denormalized number or a zero.
- 2 denormalized float numbers
  - ✓The result is smaller than minimum denormalized number.
  - ✓The result is represented by a zero.

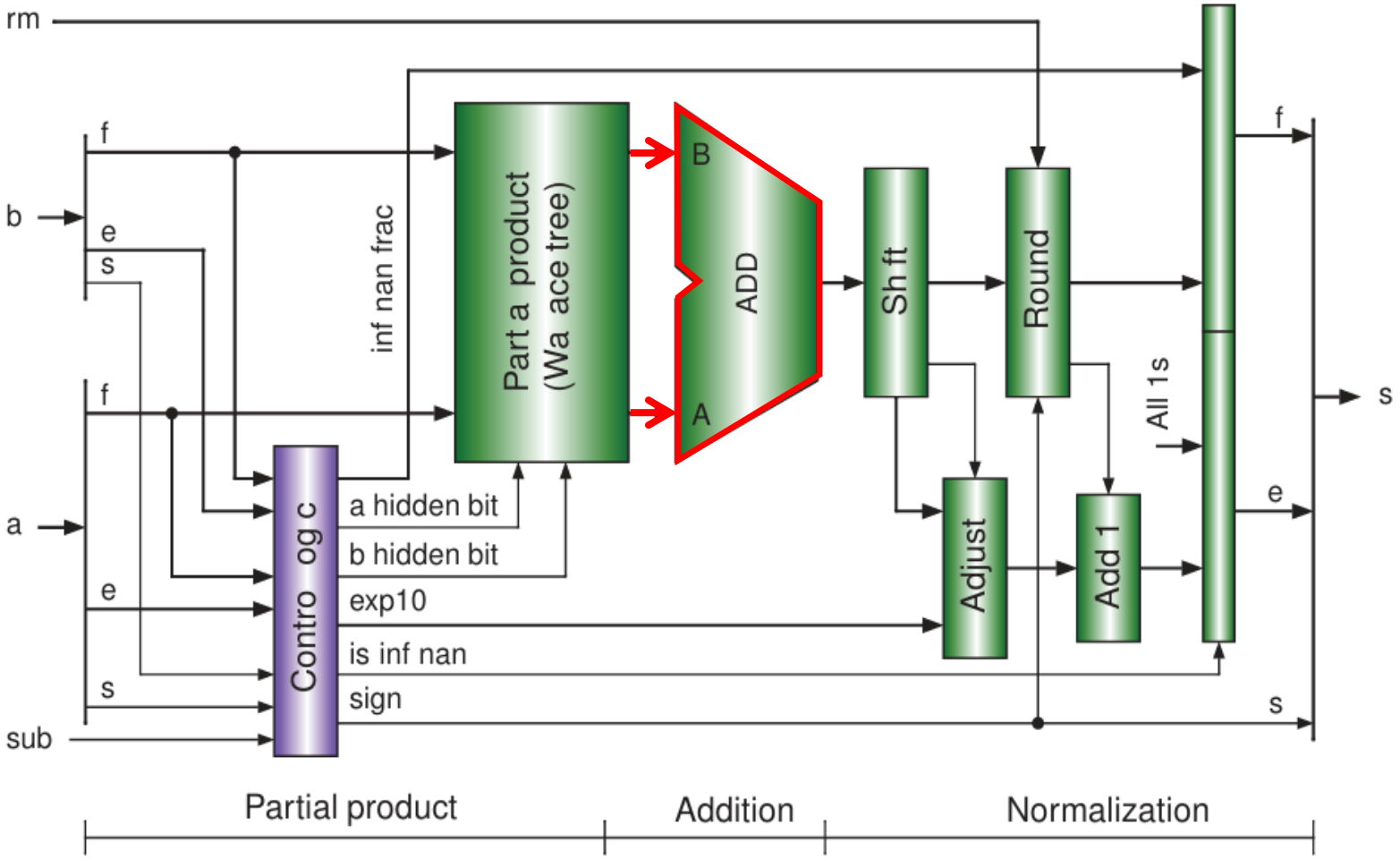
FMUL

- Block diagram of the floating-point number



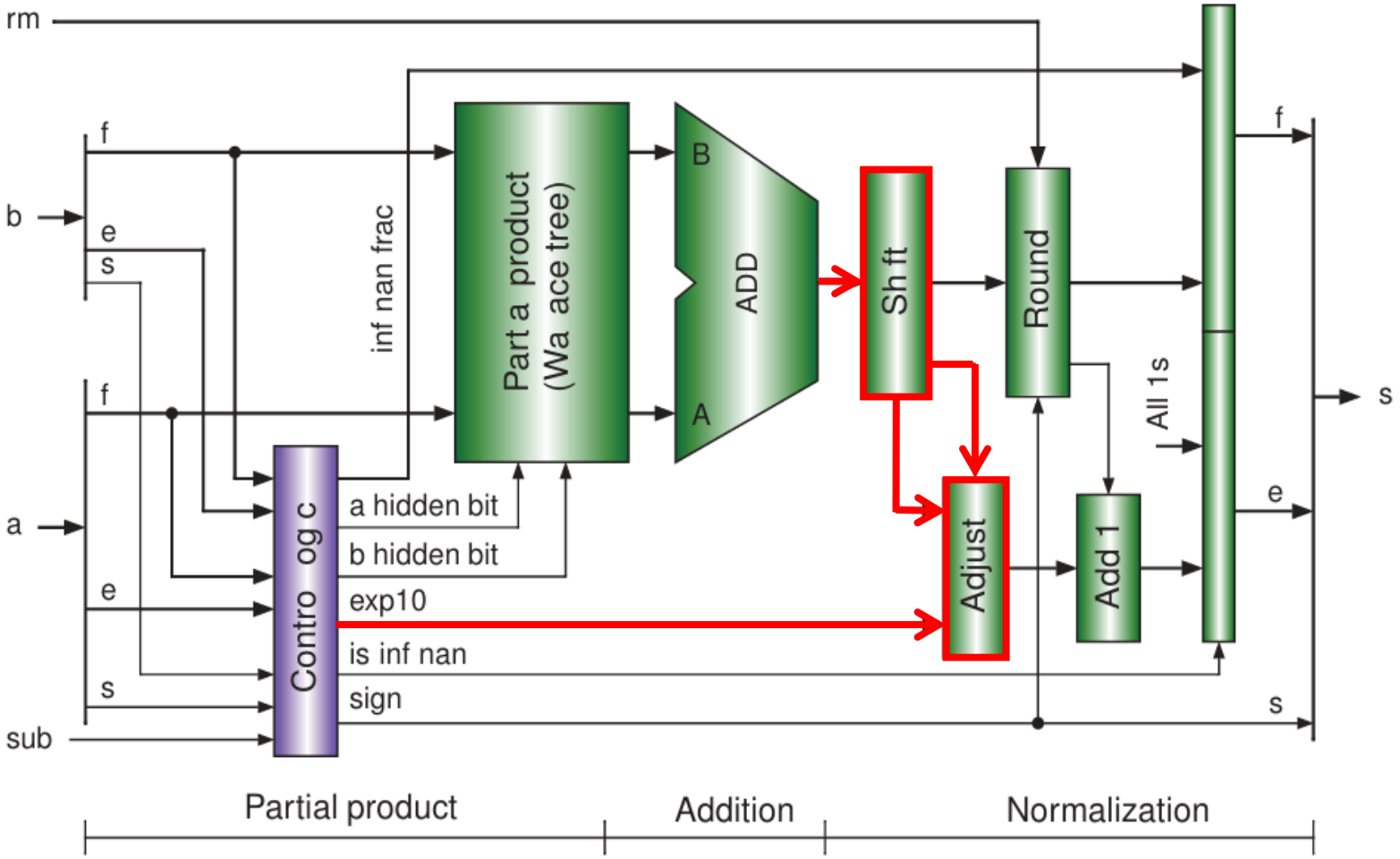
FMUL

- Block diagram of the floating-point number



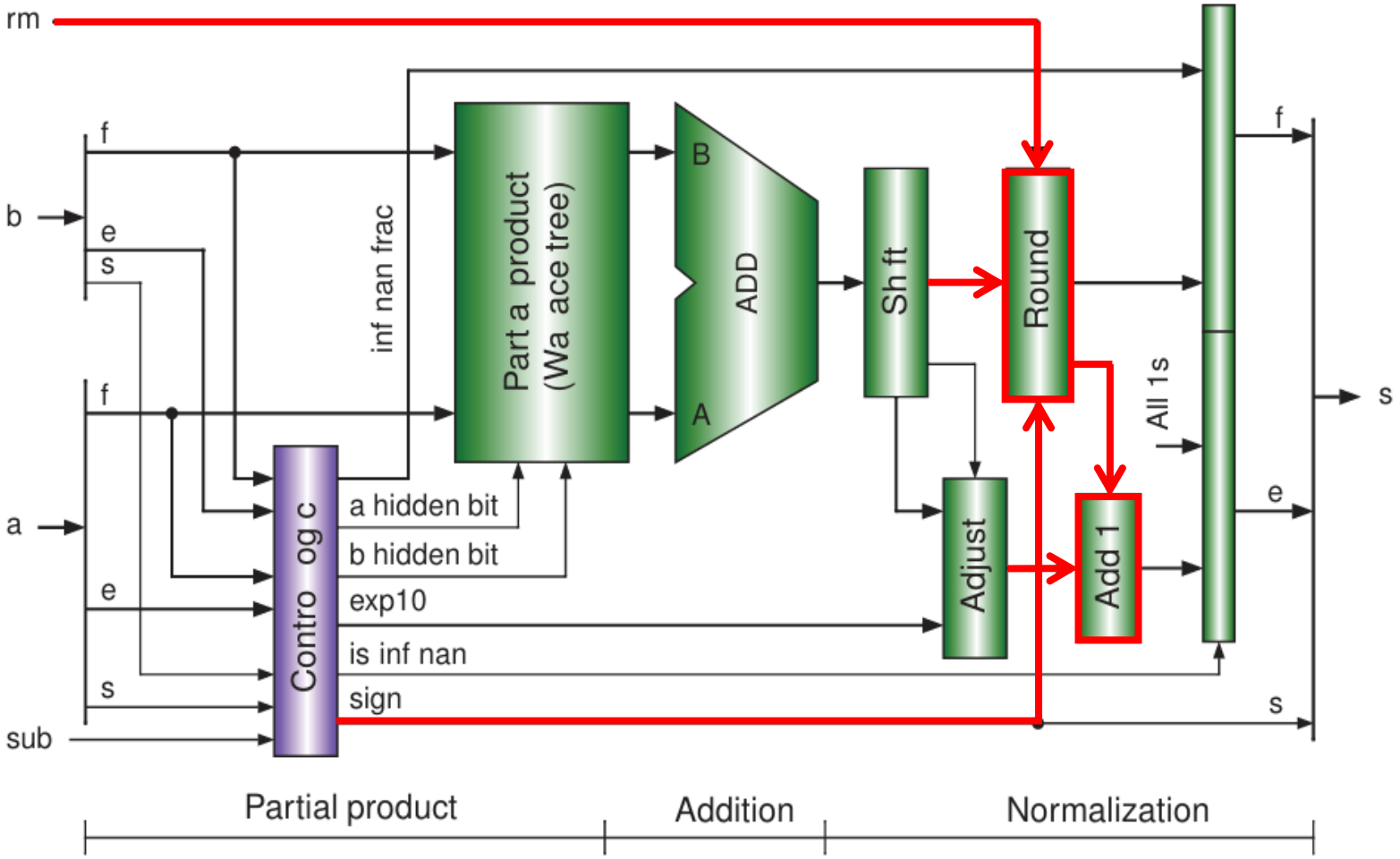
FMUL

- Block diagram of the floating-point number



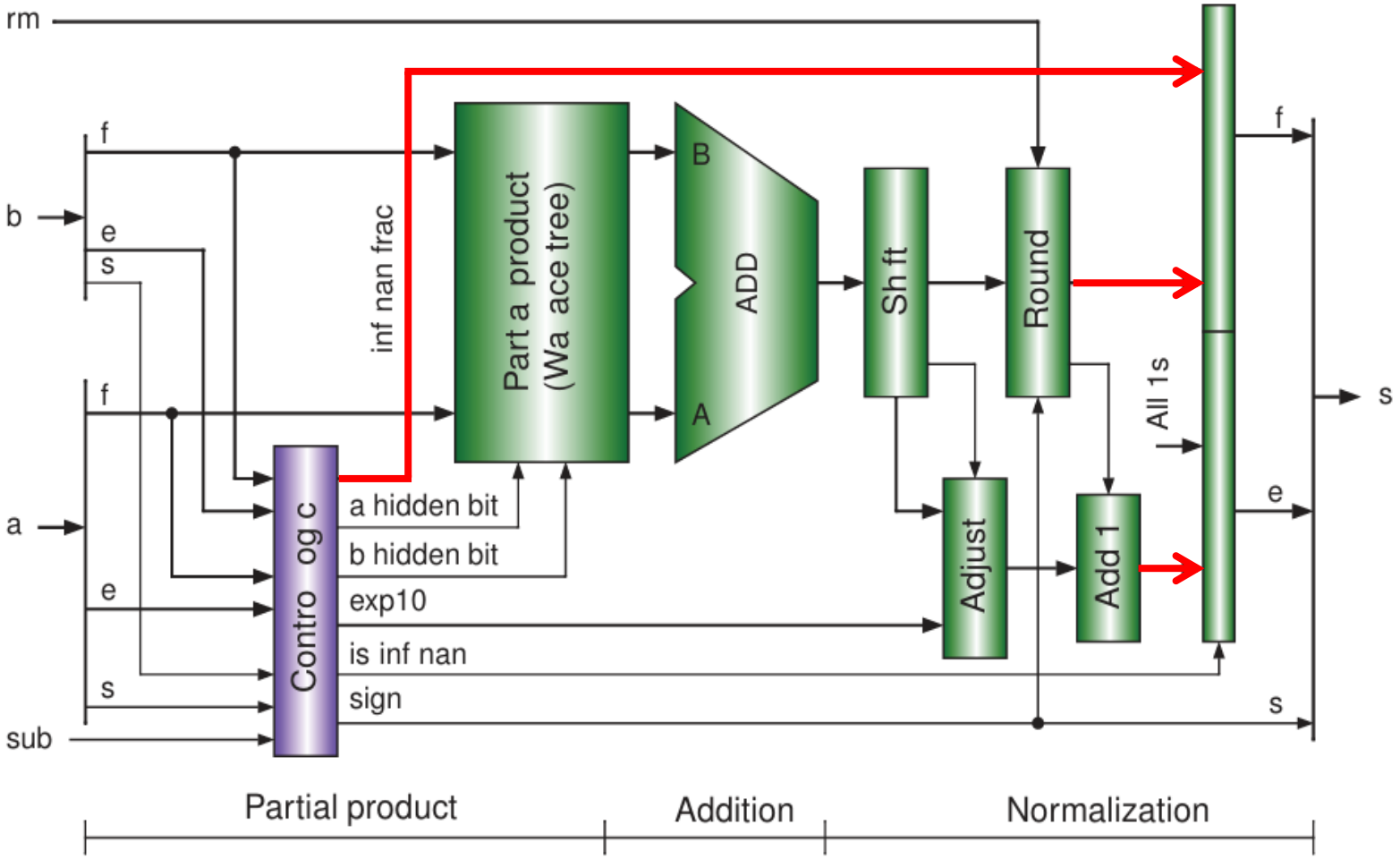
FMUL

- Block diagram of the floating-point number



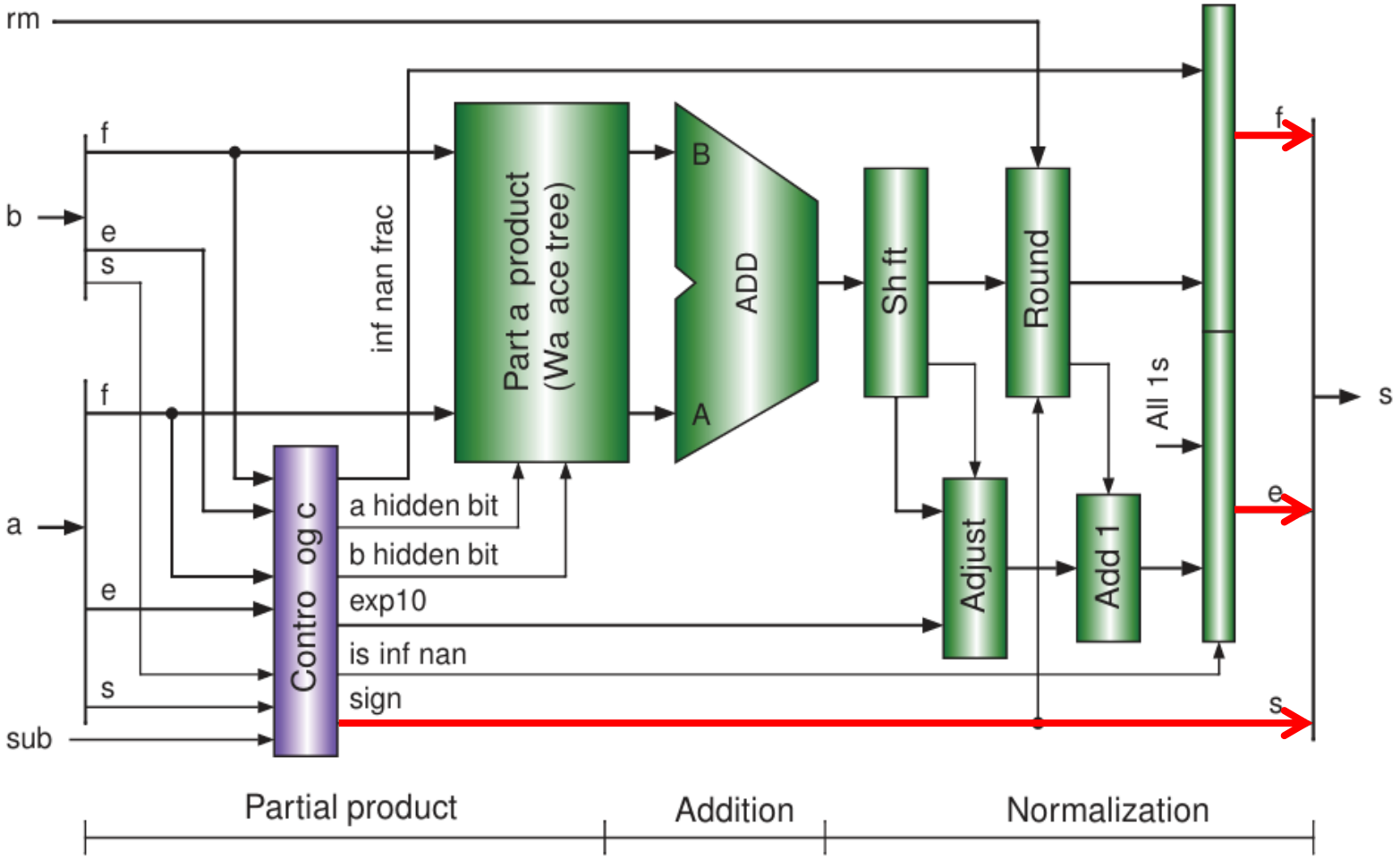
FMUL

- Block diagram of the floating-point number



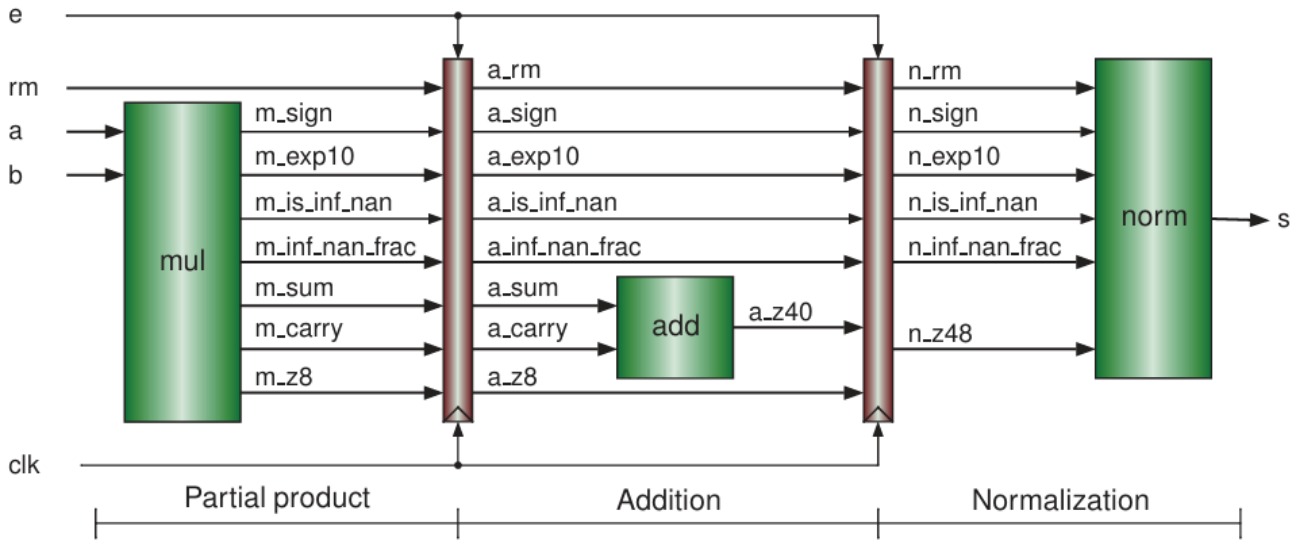
FMUL

- Block diagram of the floating-point number



FMUL

- Pipelined floating-point multiplier (Pipelined Wallace tree FMUL).
- Two clock cycles to perform the significand multiplication.
- To execute a float multiplication on every clock cycle, add a pipeline register to the Wallace tree.
- Operation is divided into three stages: partial product, addition and normalization.



FMUL

- **Partial product**
- Special case detection (*NaN*, *inf*, zero).
- Sign bit computation, exponent calculation, fraction multiplication (Wallace tree multiplier).
- Prepares intermediate values for normalization and rounding.

```
12    wire          a_expo_is_00    = ~|a[30:23];
13    wire          b_expo_is_00    = ~|b[30:23];
14    wire          a_expo_is_ff    =  &a[30:23];
15    wire          b_expo_is_ff    =  &b[30:23];
16    wire          a_frac_is_00    = ~|a[22:0];
17    wire          b_frac_is_00    = ~|b[22:0];
18    wire          a_is_inf        = a_expo_is_ff &  a_frac_is_00;
19    wire          b_is_inf        = b_expo_is_ff &  b_frac_is_00;
20    wire          a_is_nan        = a_expo_is_ff & ~a_frac_is_00;
21    wire          b_is_nan        = b_expo_is_ff & ~b_frac_is_00;
22    wire          a_is_0          = a_expo_is_00 &  a_frac_is_00;
23    wire          b_is_0          = b_expo_is_00 &  b_frac_is_00;
```

FMUL

- **Partial product**
- Inf if either is inf (and not *NaN*).
- Either operand is *NaN*, one is inf and the other is zero.
- Constructs a *NaN* payload using the larger of the two fraction.

```
24      assign          s_is_inf        = a_is_inf | b_is_inf;
25      assign          s_is_nan        = a_is_nan | (a_is_inf & b_is_0) |
26                                        b_is_nan | (b_is_inf & a_is_0);
27      wire    [22:0]  nan_frac        = (a[21:0] > b[21:0]) ? {1'b1,a[21:0]} : {1'b1,b[21:0]};
28      assign          inf_nan_frac    = s_is_nan ? nan_frac : 23'h0;
```

FMUL

- **Partial product**
- Sign calculation by XOR of input signs.
- In IEEE 754, exponent is stored with a bias of 127.
- True exponent is ea + eb – 127.
- But if one operand is denormalized (expo = 0), its exponent is treated as -126.
- To fix the underflow from using a false exponent of 0, add 1 for each denorm.

```
29    assign          sign          = a[31] ^ b[31];
30    assign          exp10         = {2'h0,a[30:23]} + {2'h0,b[30:23]} - 10'h7f +
31                                    a_expo_is_00 + b_expo_is_00 // -126
32    wire    [23:0]  a_frac24      = {~a_expo_is_00,a[22:0]};
33    wire    [23:0]  b_frac24      = {~b_expo_is_00,b[22:0]};
```

FMUL

- **Partial product**
- IEEE 754 uses an implicit leading 1 for normalized numbers.
- For normalized input: frac = 1.xxx…
- For denormals: frac = 0.xxx…
- So adds the implicit 1 (or not) to create a full 24-bit fraction.

```
29      assign          sign        = a[31] ^ b[31];
30      assign          exp10       = {2'h0,a[30:23]} + {2'h0,b[30:23]} - 10'h7f +
31                                      a_expo_is_00 + b_expo_is_00 // -126
32      wire    [23:0]  a_frac24    = {~a_expo_is_00,a[22:0]};
33      wire    [23:0]  b_frac24    = {~b_expo_is_00,b[22:0]};
```

FMUL

- **Addition**
- *z_sum* and *z_carry* are the outputs from the Wallace tree.
- These two vectors are not the final binary result; they must be added using a normal binary adder.
- Adds the two 40-bit vectors to get the final 48-bit product (only upper 40 bits are kept[47:8]).

```
1  module fmul_add (
2      input   [39:0]  z_sum,
3      input   [39:0]  z_carry,
4      output  [47:8]  z
5  );
6      assign          z   = z_sum + z_carry;
7  endmodule
```

FMUL

- **Normalization**
- Counts leading zeros to figure out how much to shift the fraction left so that if becomes normalized (starts with 1.x in binary).
- *z0* = normalized fraction (1.xxx…)
- *zeros* = how many shifts

```
11      wire    [46:0]  z5, z4, z3, z2, z1, z0;
12      wire    [5:0]   zeros;
13      assign          zeros[5]    = ~|z[46:15];
14      assign          z5          = zeros[5] ? {z[14:0],32'b0} : z[46:0];
15      assign          zeros[4]    = ~|z[46:31];
16      assign          z4          = zeros[4] ? {z5[30:0],16'b0} : z5;
17      assign          zeros[3]    = ~|z[46:39];
18      assign          z3          = zeros[3] ? {z4[38:0], 8'b0} : z4;
19      assign          zeros[2]    = ~|z[46:43];
20      assign          z2          = zeros[2] ? {z3[42:0], 4'b0} : z3;
21      assign          zeros[1]    = ~|z[46:45];
22      assign          z1          = zeros[1] ? {z2[44:0], 2'b0} : z2;
23      assign          zeros[0]    = ~z1[46];
24      assign          z0          = zeros[0] ? {z1[45:0], 1'b0} : z1;
```

FMUL

- **Normalization**
- Adjusts the exponent and fraction based on whether normalization is needed.

```
26      reg     [46:0]  frac0;
27      reg     [9:0]   exp0;
28      always @(*) begin
29          if (z[47]) begin                // 1x.xxx...
30              exp0    = exp10 + 10'h1;
31              frac0   = z[47:1];          //  1.xxx...
32          end else begin
33              if (!exp10[9] && (exp10[8:0] > zeros) && z0[46]) begin
34                  exp0    = exp10 - zeros;
35                  frac0   = z0;           //  1.xxx...
36              end else begin              // is a denormalized number or 0
37                  exp0    = 0;
38                  if (!exp10[9] && (exp10 != 0))              // e > 0
39                      frac0   = z[46:0] << (exp10 - 10'h1);   // e-127 -> -126
40                  else
41                      frac0   = z[46:0] >> (10'h1 - exp10);   // e = 0 or neg
42              end
43          end
44      end
```

FMUL

- **Normalization**
- Keep top 23 bits (plus 3: guard, round sticky), total 27 bits.
- Compute whether to round up by grs bits and rounding modes.

```
45      wire    [26:0]  frac     = {frac0[46:21],|frac0[20:0]};  // x.xx...xx grs
46      wire            frac_plus_1 = ~rm[1] & ~rm[0] &  frac0[2] & (frac0[1] |  frac0[0]) |
47                                    ~rm[1] & ~rm[0] &  frac0[2] & ~frac0[1] & ~frac0[0]  & frac0[3] |
48                                    ~rm[1] &  rm[0] & (frac0[2] |  frac0[1] |  frac0[0]) &  sign |
49                                     rm[1] & ~rm[0] & (frac0[2] |  frac0[1] |  frac0[0]) & ~sign;
50      wire    [24:0]  frac_round  = {1'b0,frac[26:3]} + frac_plus_1;
51      wire    [9:0]   exp1        = frac_round[24] ? exp0 + 10'h1 : exp0;
52      wire            overflow    = (exp0 >= 10'h0ff) | (exp1 >= 10'h0ff);
```

FDIV

- Floating-point division algorithm
- 2 normalized float numbers
  - ✓The result is normalized number, if 1 <= e <= 254.
  - ✓The result is an infinity, if e >254.
  - ✓The result can be a denormalized number or zero, if e < 1
- Dividing a normalized float # by a denormalized float #
  - ✓The result is larger than maximum, it represented with an infinity.
- Dividing a denormalized float # by a normalized float #
  - ✓The largest absolute is a normalized number.
  - ✓The smallest absolute is represented with a zero
  - ✓It can be represented by a normalized #, denormalized #, and a zero.
- 2 denormalized float numbers
  - ✓The result is a normalized number.

FDIV

- Special cases
  - ✓ *a/inf = 0*
  - ✓ *a/0 = inf*
  - ✓ *0/0 = NaN*
  - ✓ *inf/inf = NaN*
  - ✓ *Nan/b = NaN*

FDIV

- Newton-Raphson algorithm to perform the significand division.
- To compute *N / D*, we can first compute *1 / D* then multiply *N.*
- ***x_{n+1} = x_n + x_n * (1 – x_n * D) = x_n * (2 – x_n * D)***
- Multiplication is fast and parallelizable, while division is slow.
- Newton-Raphson algorithm converts division into multiplications.

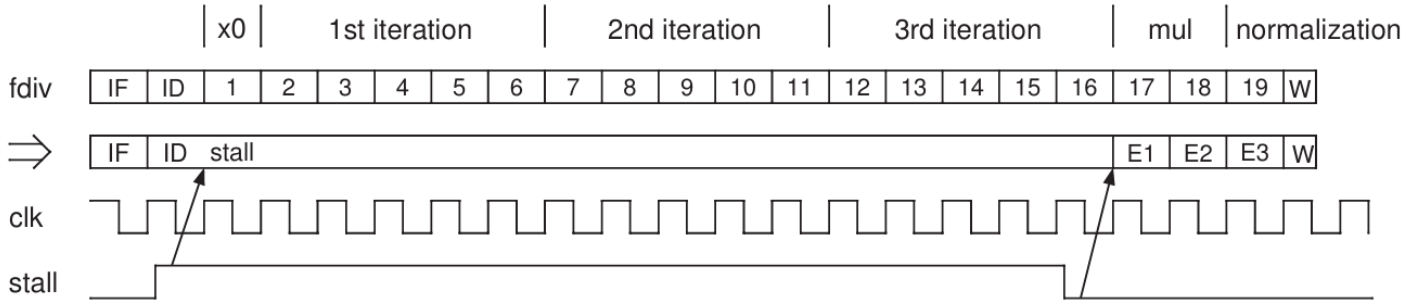
FDIV

- The Newton-Raphson division algorithm consists of two parts.
  - ✓iterative calculation, $x_{n+1} = x_n * (2 - x_n * D)$
  - ✓quotient calculation, $q = a * x_n$

- The operation of the float division takes the following steps.
  - ✓Newton iteration, which calculates x_n
  - ✓partial product, which calculates carry and sum with Wallace CSA array
  - ✓addition, which calculates the product by adding the carry and sum
  - ✓normalization, which generates the final result in the IEEE float format

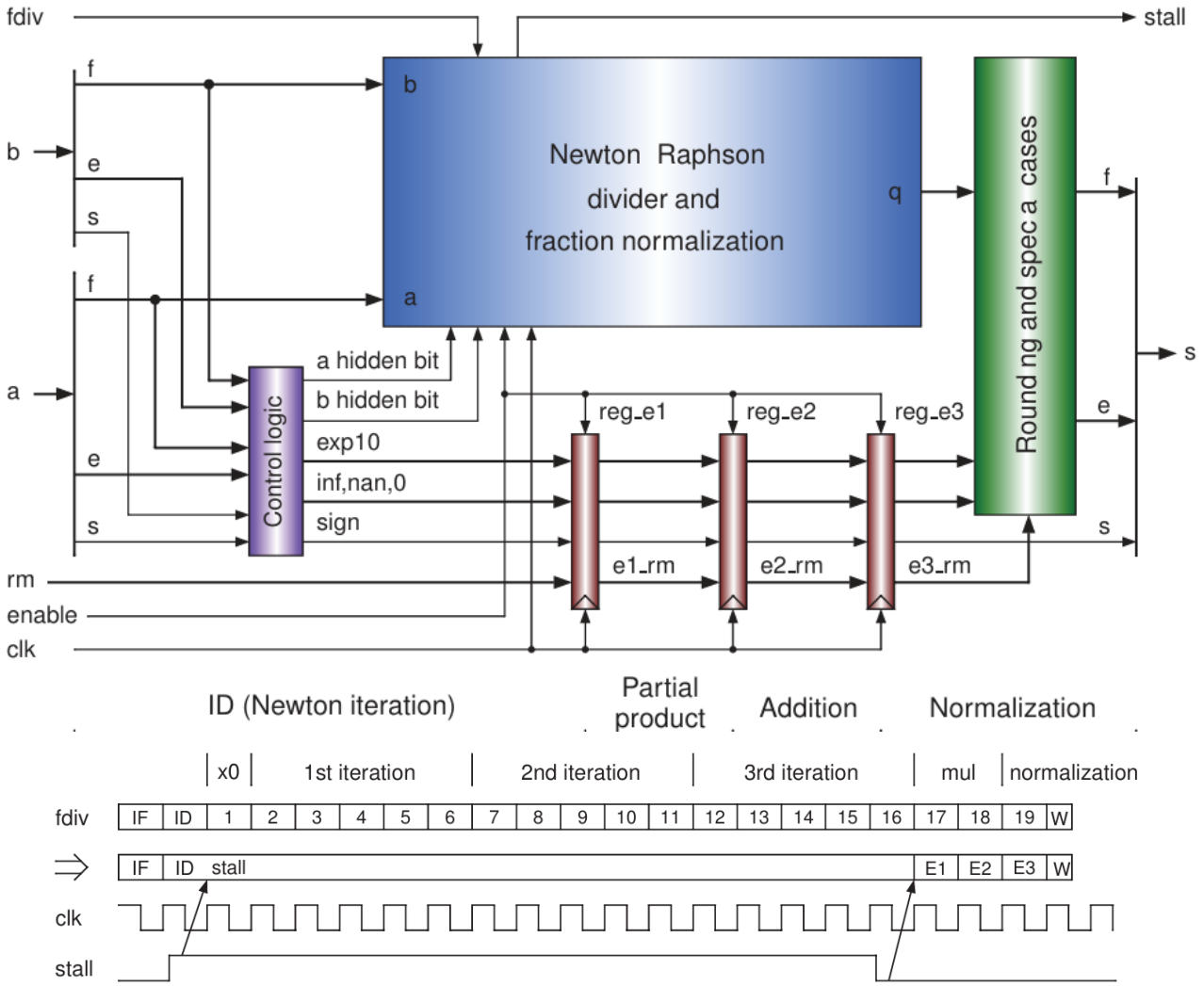
FDIV

- Step 1 is done in an iterative manner.
- Steps 2-4 are done in the pipeline manner.
- The pipeline is stalled during Newton iteration in ID stage.
- Each iteration takes 5 clock cycle.
  - ✓2 multiplication and 1 subtraction.
- The multiplication is done with Wallace tree (2 cycles).
- The subtraction takes 1 cycle.



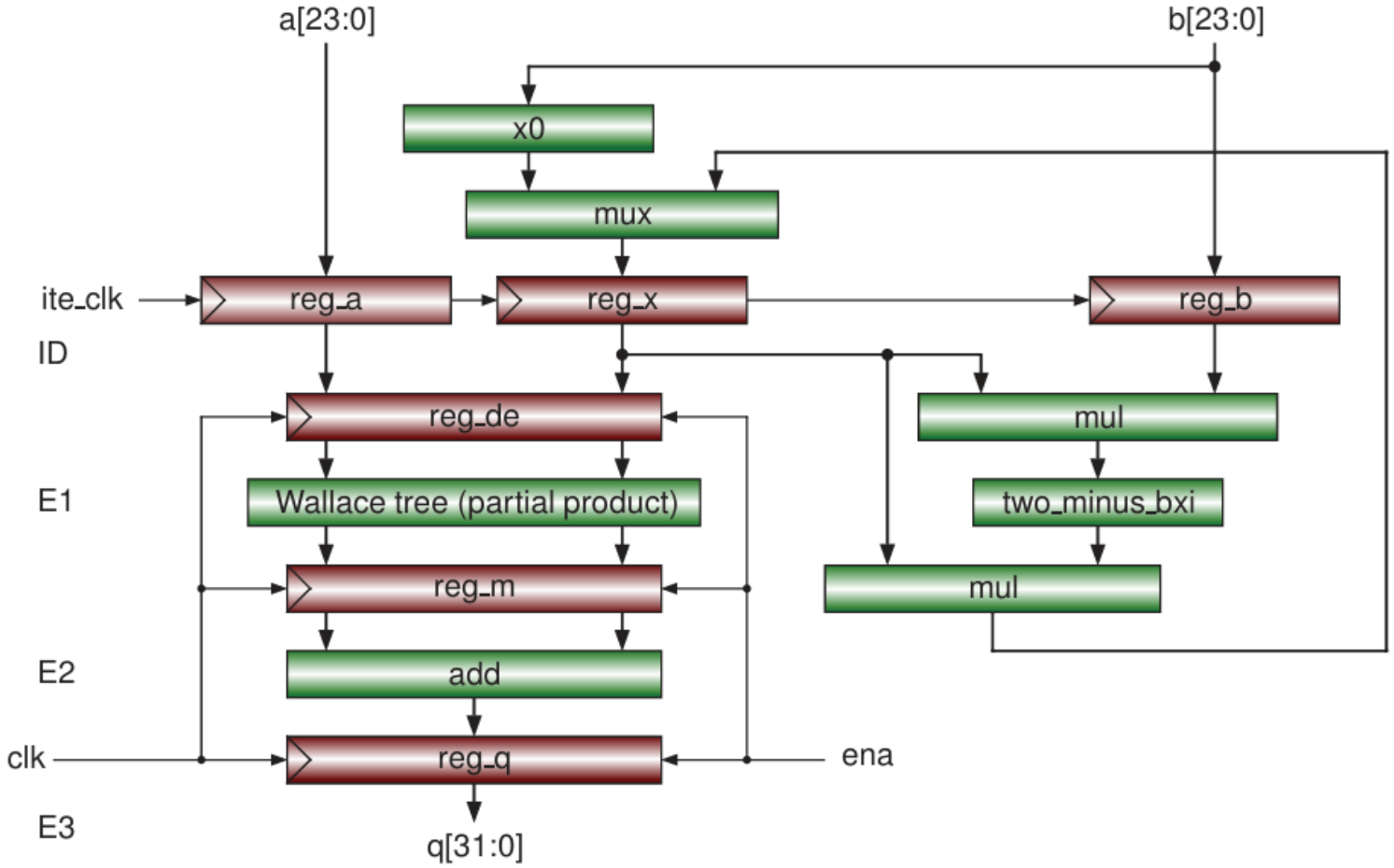
FDIV

- Block diagram of the floating-point divider



FDIV

- Schematic diagram of the 24-bit Newton-Raphson divider



FDIV

- **Newton-Raphson divider**

```
1  module newton24 (
2      input           [23:0]  a,
3      input           [23:0]  b,
4      input                   fdiv,
5      input                   clk, clrn,
6      input                   ena,
7      output          [31:0]  q,
8      output  reg             busy,
9      output          [4:0]   count,
10     output          [25:0]  reg_x,
11     output                  stall
12 );
```

a,b : 24-bit normalized dividend and divisor

q : 32-bit final fractional result (A * (1/B))
busy : busy performing calculation
count : 5-bit counter tracking iterations
reg_x : current approx. of the reciprocal of b

FDIV

- **Newton-Raphson divider**

```
13      reg             [31:0]  q;
14      reg             [25:0]  reg_x;
15      reg             [23:0]  reg_a;
16      reg             [23:0]  reg_b;
17      reg             [4:0]   count;
18      wire            [49:0]  bxi;
19      wire            [51:0]  x52;
20      wire            [49:0]  d_x;
21      wire            [31:0]  e2p;
```

reg_x : stores current reciprocal approximation
reg_a, reg_b : store the input fraction
count : store the iteration count

bxi : hold the product B*X_i (24 + 26 – 1)
x52 : hold the product X_i * (2 – B*X_i) (26 + 26)
d_x : intermediate value for the final multiplication (A * X_i)
e2p : the final fraction output q before assignment

FDIV

- **Newton-Raphson divider**
- This line calculates the initial approximation for the reciprocal X.
- rom is a function that acts as a LUT.
- b[22:9] extract 14 bits from the mantissa b, but rom takes only 4.
- Only b[12:9] are being used as the address for rom.

```
22      wire                [7:0]   x0          = rom(b[22:9]);
```

```
86      function    [7:0]   rom;
87          input   [3:0]   b;
88          case (b)
89              4'h0: rom = 8'hff;          4'h1: rom = 8'hdf;
90              4'h2: rom = 8'hc3;          4'h3: rom = 8'haa;
91              4'h4: rom = 8'h93;          4'h5: rom = 8'h7f;
92              4'h6: rom = 8'h6d;          4'h7: rom = 8'h5c;
93              4'h8: rom = 8'h4d;          4'h9: rom = 8'h3f;
94              4'ha: rom = 8'h33;          4'hb: rom = 8'h27;
95              4'hc: rom = 8'h1c;          4'hd: rom = 8'h12;
96              4'he: rom = 8'h08;          4'hf: rom = 8'h00;
97          endcase
98      endfunction
```

FDIV

- **Newton-Raphson divider**

```
23      always @(posedge clk or negedge clrn) begin
24          if (!clrn) begin
25              busy    <= 0;
26              count   <= 0;
27              reg_x   <= 0;
28          end else begin
29              if (fdiv & (count == 0)) begin
30                  count   <= 5'b1;
31                  busy    <= 1'b1;
32              end else begin
33                  if (count == 5'h01) begin
34                      reg_a   <= a;
35                      reg_b   <= b;
36                      reg_x   <= {2'b1,x0,16'b0};
37                  end
38                  if (count != 0)     count   <= count + 5'b1;
39                  if (count == 5'h0f) busy    <= 0;
40                  if (count == 5'h10) count   <= 5'b0;
41                  if ( (count == 5'h06) ||
42                       (count == 5'h0b) ||
43                       (count == 5'h10) )
44                       reg_x  <= x52[50:25];
45              end
46          end
47      end
```

Newton-Raphson iteration result are fed back into reg_x for the next iteration from x52 at specific iterations (count 6, 11, and 16).

The [50:25] bit slice selects the relevant 26 bits from the x52 product.

FDIV

- **Newton-Raphson divider**
- Stalls if *fdiv* comes in while the module is idle (count == 0).
- Stalls if the module is already busy with an operation.
- wallace_26x24_product calculate B*X_i
- wallace_26x26_product calculate the next reciprocal approximation X_{i+1} = x_i * (2 – B*X_i).

```
48      assign          stall   = fdiv & (count == 0) | busy;
49      wallace_26x24_product   bxxi (
50          .a(reg_b),
51          .b(reg_x),
52          .z(bxi)
53      );
54      wire    [25:0]  b26     = ~bxi[48:23] + 1'b1;
55      wallace_26x26_product   xip1 (
56          .a(reg_x),
57          .b(b26),
58          .z(x52)
59      );
```

FDIV

- **Newton-Raphson divider**
- wallace_24x26 is for the final multiplication A * X_final.

```
60      reg     [25:0]  reg_de_x;
61      reg     [23:0]  reg_de_a;
62      wire    [49:0]  m_s;
63      wire    [49:8]  m_c;
64      wallace_24x26           wt (
65          .a(reg_de_a),
66          .b(reg_de_x),
67          .x(m_s[49:8]),
68          .y(m_c),
69          .z(m_s[7:0])
70      );
```

FDIV

- **Newton-Raphson divider**
- reg *a_s* and reg *a_c* hold the sum and carry output of wt.
- Aligning the sum and carry bit before the final addition.
- Truncating *d_x* to 31 bits and calculates the sticky bit.

```
71      reg     [49:0]  a_s;
72      reg     [49:8]  a_c;
73      assign          d_x = {1'b0,a_s} + {a_c,8'b0};
74      assign          e2p = {d_x[48:18],|d_x[17:0]};
```

FDIV

- **Newton-Raphson divider**
- Managing pipeline registers for the final multiplication.
- If *ena* is set
  - ✓The most refined reciprocal X is latched into *reg_de_x* for the final mult.
  - ✓The dividend fraction *reg_a* is latched to *reg_de_a*.
  - ✓The sum and carry vectors from *wt* multiplier are latched.
  - ✓The result *e2p* is latched into the output *q*.

```
75      always @(posedge clk or negedge clrn) begin
76          if (!clrn) begin
77              reg_de_x    <= 0;           reg_de_a    <= 0;
78              a_s         <= 0;           a_c         <= 0;
79              q           <= 0;
80          end else if (ena) begin
81              reg_de_x    <= x52[50:25];  reg_de_a    <= reg_a;
82              a_s         <= m_s;         a_c         <= m_c;
83              q           <= e2p;
84          end
85      end
```

FDIV

- **Newton-Raphson divider**
- Defined a function rom.
- LUT provides an initial approximation of 1/B.

```
86      function    [7:0]   rom;
87          input   [3:0]   b;
88          case (b)
89              4'h0: rom = 8'hff;          4'h1: rom = 8'hdf;
90              4'h2: rom = 8'hc3;          4'h3: rom = 8'haa;
91              4'h4: rom = 8'h93;          4'h5: rom = 8'h7f;
92              4'h6: rom = 8'h6d;          4'h7: rom = 8'h5c;
93              4'h8: rom = 8'h4d;          4'h9: rom = 8'h3f;
94              4'ha: rom = 8'h33;          4'hb: rom = 8'h27;
95              4'hc: rom = 8'h1c;          4'hd: rom = 8'h12;
96              4'he: rom = 8'h08;          4'hf: rom = 8'h00;
97          endcase
98      endfunction
```

FDIV

- **Input & Pre-processing**
- Identify the characteristics of the input operands (zero, denormalized, *inf*, or *NaN*).
- Sign calculation
- Perform the initial exponent subtraction and add the bias 127

```
18      wire            a_expo_is_00    = ~|a[30:23];
19      wire            b_expo_is_00    = ~|b[30:23];
20      wire            a_expo_is_ff    =  &a[30:23];
21      wire            b_expo_is_ff    =  &b[30:23];
22      wire            a_frac_is_00    = ~|a[22:00];
23      wire            b_frac_is_00    = ~|b[22:00];
24      wire            sign            = a[31] ^ b[31];
25      wire    [9:0]   exp_10          = {2'h0,a[30:23]} - {2'h0,b[30:23]} + 10'h7f;
```

FDIV

- **Input & Pre-processing**
- For normalized numbers, the implicit ‘1’ is prepended.
- For denormalized numbers, 1’b0 appended.
- External modules prepare the fraction for the actual division by ensuring they are in a consistent ‘1.f’ format.
- The initial exponent *exp_10* is adjusted by the shift amount.

```
26      wire    [23:0]  a_temp24        = a_expo_is_00 ? {a[22:0],1'b0} : {1'b1,a[22:0]};
27      wire    [23:0]  b_temp24        = b_expo_is_00 ? {b[22:0],1'b0} : {1'b1,b[22:0]};
28      wire    [23:0]  a_frac24, b_frac24;
29      wire    [4:0]   shamt_a, shamt_b;
30      shift_to_msb_equ_1  shift_a (a_tmp24, a_frac24, shamt_a);
31      shift_to_msb_equ_1  shift_b (b_tmp24, b_frac24, shamt_b);
32      wire    [9:0]   exp10   = exp_10 - shamt_a + shamt_b;
```

FDIV

- **Pipeline registers**

```
34      reg             e1_sign, e1_ae00, e1_aeff, e1_af00, e1_be00, e1_beff, e1_bf00;
35      reg             e2_sign, e2_ae00, e2_aeff, e2_af00, e2_be00, e2_beff, e2_bf00;
36      reg             e3_sign, e3_ae00, e3_aeff, e3_af00, e3_be00, e3_beff, e3_bf00;
37      reg     [1:0]   e1_rm, e2_rm, e3_rm;
38      reg     [9:0]   e1_exp10, e2_exp10, e3_exp10;
39      always @ (negedge clrn or posedge clk) begin
40          if (!clrn) begin   // 3 pipeline registers: reg_e1, reg_e2, and reg_e3
41              // reg_e1                 // reg_e2             // reg_e3
42              e1_sign <= 0;             e2_sign <= 0;         e3_sign <= 0;
43              e1_rm   <= 0;             e2_rm   <= 0;         e3_rm   <= 0;
44              e1_exp10<= 0;             e2_exp10<= 0;         e3_exp10<= 0;
45              e1_ae00 <= 0;             e2_ae00 <= 0;         e3_ae00 <= 0;
46              e1_aeff <= 0;             e2_aeff <= 0;         e3_aeff <= 0;
47              e1_af00 <= 0;             e2_af00 <= 0;         e3_af00 <= 0;
48              e1_be00 <= 0;             e2_be00 <= 0;         e3_be00 <= 0;
49              e1_beff <= 0;             e2_beff <= 0;         e3_beff <= 0;
50              e1_bf00 <= 0;             e2_bf00 <= 0;         e3_bf00 <= 0;
51          end else if (ena) begin
52              e1_sign <= sign;          e2_sign <= e1_sign;   e3_sign <= e2_sign;
53              e1_rm   <= rm;            e2_rm   <= e1_rm;     e3_rm   <= e2_rm;
54              e1_exp10<= exp10;         e2_exp10<= e1_exp10;  e3_exp10<= e2_exp10;
55              e1_ae00 <= a_expo_is_00;  e2_ae00 <= e1_ae00;   e3_ae00 <= e2_ae00;
56              e1_aeff <= a_expo_is_ff;  e2_aeff <= e1_aeff;   e3_aeff <= e2_aeff;
57              e1_af00 <= a_frac_is_00;  e2_af00 <= e1_af00;   e3_af00 <= e2_af00;
58              e1_be00 <= b_expo_is_00;  e2_be00 <= e1_be00;   e3_be00 <= e2_be00;
59              e1_beff <= b_expo_is_ff;  e2_beff <= e1_beff;   e3_beff <= e2_beff;
60              e1_bf00 <= b_frac_is_00;  e2_bf00 <= e1_bf00;   e3_bf00 <= e2_bf00;
61          end
62      end
```

FDIV

- **Newton-Raphson divider**

```
64      wire  [31:0]  q;
65      newton24  frac_newton (
66          .a(a_frac24),
67          .b(b_frac24),
68          .fdiv(fdiv),
69          .clk(clk),
70          .clrn(clrn),
71          .ena(ena),
72
73          .q(q),
74          .busy(busy),
75          .count(count),
76          .reg_x(reg_x),
77          .stall(stall)
78      );
```

FDIV

- If the MSB of *q (q[31])* is ‘1’, q is already in the form 1.xxx…x.
- If *q[31]* is ‘0’, it needs to be shifted left to bring the leading ‘1’.
- The exponent is adjusted based on the normalization of *q*.
- If *q* was shifted left, the exponent needs to be decremented.

```
79      wire  [31:0]  z0          = q[31] ? q : {q[30:0],1'b0};   // 1.xxx...x
80      wire  [9:0]   expo_adj    = q[31] ? e3_exp10 : e3_exp10 - 10'b1;
```

FDIV

- Check the exponent and the fraction before final rounding.

```
83      always @(*) begin
84          if (exp_adj[9]) begin
85              exp0    = 0;
86              if (z0[31])
87                  frac0   = z0 >> (10'b1 - exp_adj);
88              else
89                  frac0   = 0;
90          end else if (exp_adj == 0) begin
91              exp0    = 0;
92              frac0   = {1'b0,z0[31:2],|z0[1:0]};
93          end else begin
94              if (exp_adj > 254) begin
95                  exp0    = 10'hff;
96                  frac0   = 0;
97              end else begin
98                  exp0    = exp_adj;
99                  frac0   = z0;
100             end
101         end
102     end
```

FDIV

- Creates a 27-bit fraction from frac0.
- Determines whether to round up by adding 1 or not.
- rounding mode *e3_rm* and grs bits of the fraction
- Perform the actual rounding.
- Checks the fraction overflow and the exponent overflow.

```
103     wire    [26:0]  frac        = {frac0[31:6],frac0[5:0]};
104     wire            frac_plus_1 =
105                         ~e3_rm[1] & ~e3_rm[0] & frac[3] &  frac[2] & ~frac[1] & frac[0]  |
106                         ~e3_rm[1] & ~e3_rm[0] &           frac[2] & (frac[1] | frac[0]) |
107                         ~e3_rm[1] &  e3_rm[0] &          (frac[2] |  frac[1] | frac[0]) &  e3_sign |
108                          e3_rm[1] & ~e3_rm[0] &          (frac[2] |  frac[1] | frac[0]) & ~e3_sign;
109     wire    [24:0]  frac_round  = {1'b0,frac[26:3]} + frac_plus_1;
110     wire    [9:0]   exp1        = frac_round[24] ? exp0 + 10'h1 : exp0;
111     wire            overflow    = (exp1 >= 10'h0ff);
```

FDIV

```
114     assign          {exponent,fraction} = final_result(overflow,e3_rm,e3_sign,
115                                             e3_ae00,e3_aeff,e3_af00,e3_be00,e3_beff,
116                                             e3_bf00,{exp1[7:0],frac_round[22:0]});
117     assign          s                   = {e3_sign,exponent,fraction};
118     function  [30:0] final_result;
119         input         overflow;
120         input  [1:0] e3_rm;
121         input         e3_sign;
122         input         a_e00,a_eff,a_f00, b_e00,b_eff,b_f00;
123         input [30:0] calc;
124         casex ({overflow,e3_rm,e3_sign,a_e00,a_eff,a_f00,b_e00,b_eff,b_f00})
125             10'b100x_xxx_xxx : final_result = INF;    // overflow
126             10'b1010_xxx_xxx : final_result = MAX;    // overflow
127             10'b1011_xxx_xxx : final_result = INF;    // overflow
128             10'b1100_xxx_xxx : final_result = INF;    // overflow
129             10'b1101_xxx_xxx : final_result = MAX;    // overflow
130             10'b111x_xxx_xxx : final_result = MAX;    // overflow
131             10'b0xxx_010_xxx : final_result = NaN;    // NaN / any
132             10'b0xxx_011_010 : final_result = NaN;    // inf / NaN
133             10'b0xxx_100_010 : final_result = NaN;    // den / NaN
134             10'b0xxx_101_010 : final_result = NaN;    //   0 / NaN
135             10'b0xxx_00x_010 : final_result = NaN;    // nor / NaN
136             10'b0xxx_011_011 : final_result = NaN;    // inf / inf
137             10'b0xxx_100_011 : final_result = ZERO;   // den / inf
138             10'b0xxx_101_011 : final_result = ZERO;   //   0 / inf
139             10'b0xxx_00x_011 : final_result = ZERO;   // nor / inf
140             10'b0xxx_011_101 : final_result = INF;    // inf / 0
141             10'b0xxx_100_101 : final_result = INF;    // den / 0
142             10'b0xxx_101_101 : final_result = NaN;    //   0 / 0
143             10'b0xxx_00x_101 : final_result = INF;    // nor / 0
144             10'b0xxx_011_100 : final_result = INF;    // inf / den
145             10'b0xxx_100_100 : final_result = calc;   // den / den
146             10'b0xxx_101_100 : final_result = ZERO;   //   0 / den
147             10'b0xxx_00x_100 : final_result = calc;   // nor / den
148             10'b0xxx_011_00x : final_result = INF;    // inf / nor
149             10'b0xxx_100_00x : final_result = calc;   // den / nor
150             10'b0xxx_101_00x : final_result = ZERO;   //   0 / nor
151             10'b0xxx_00x_00x : final_result = calc;   // nor / nor
152             default          : final_result = ZERO;
153         endcase
154     endfunction
```

Assigns the final 32-bit result s by concatenating the sign bit, the calculated exponent, and the calculated fraction.

FSQRT

- Float sqrt of a normalized float number
  - ✓If ***ed – 127*** is even, ed is odd, shift ***1.fd*** to the right by 2 bits
  - ✓If ***ed – 127*** is odd, ed is even, shift ***1.fd*** to the right by 1 bit.
  - ✓whether ***ed*** is even or odd, always perform ***eq = ed >> 1 + 63 + ed%2***

- Float sqrt of a denormalized float number
  - ✓Shift ***0.fd*** by even bits to get ***0.fd'*** with a format of 0.1xx…x or 0.01x…x
  - ✓***1.fd = sqrt(0.fd') << 1***
  - ✓***eq = 63 – t/2***

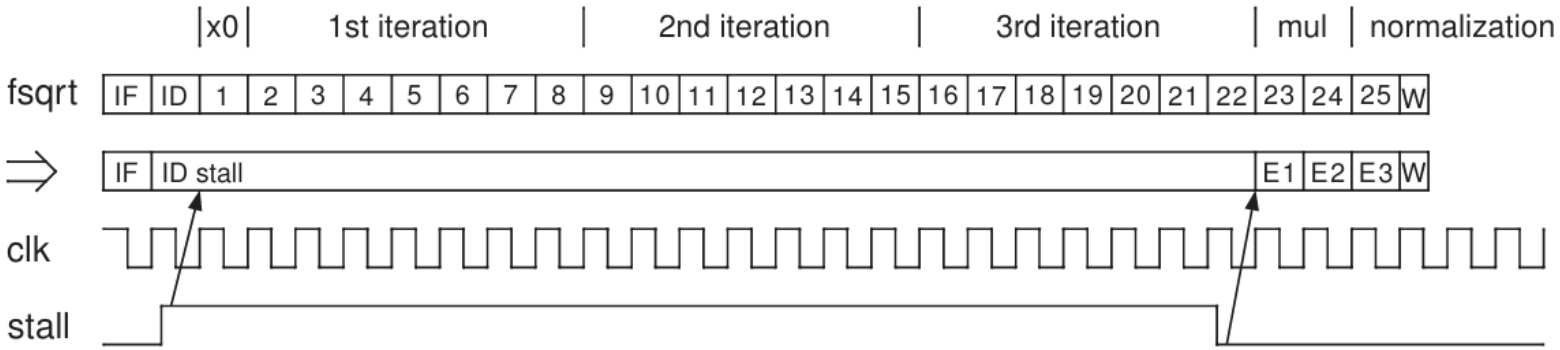
FSQRT

- The Newton-Raphson square root algorithm consist of two parts.
  - ✓iterative calculation, ***x_{i+1} = x_i * (3 – x_i^2 * d) / 2***
  - ✓square root calculation, ***q = d * x_n***

- The operation of the float square root takes four steps.
  - ✓Newton iteration, which calculates *x_n*
  - ✓partial product, which calculates carry and sum with Wallace CSA array
  - ✓addition, which calculates the product by adding the carry and sum
  - ✓normalization, which generates the final result in the IEEE float format

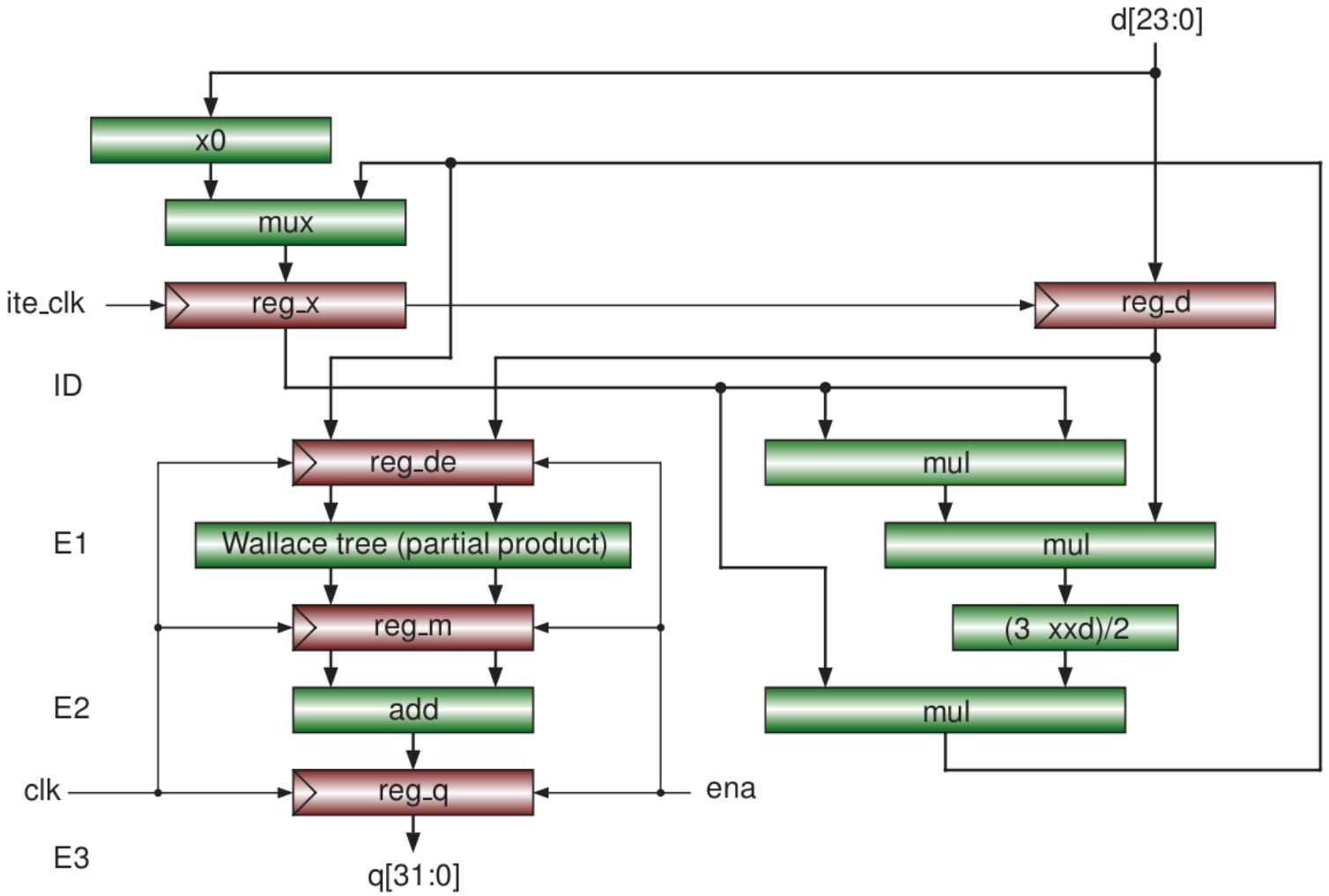
FSQRT

- Step 1 is done in an iterative manner.
- Step 2-4 is done in a pipeline manner.
- The pipeline is stalled during Newton iteration in ID stage.
- Each iteration takes 7 clock cycle.
  - ✓3 multiplication and 1 subtraction.
- The multiplication is done with Wallace tree (2 cycles).
- The subtraction takes 1 cycle.



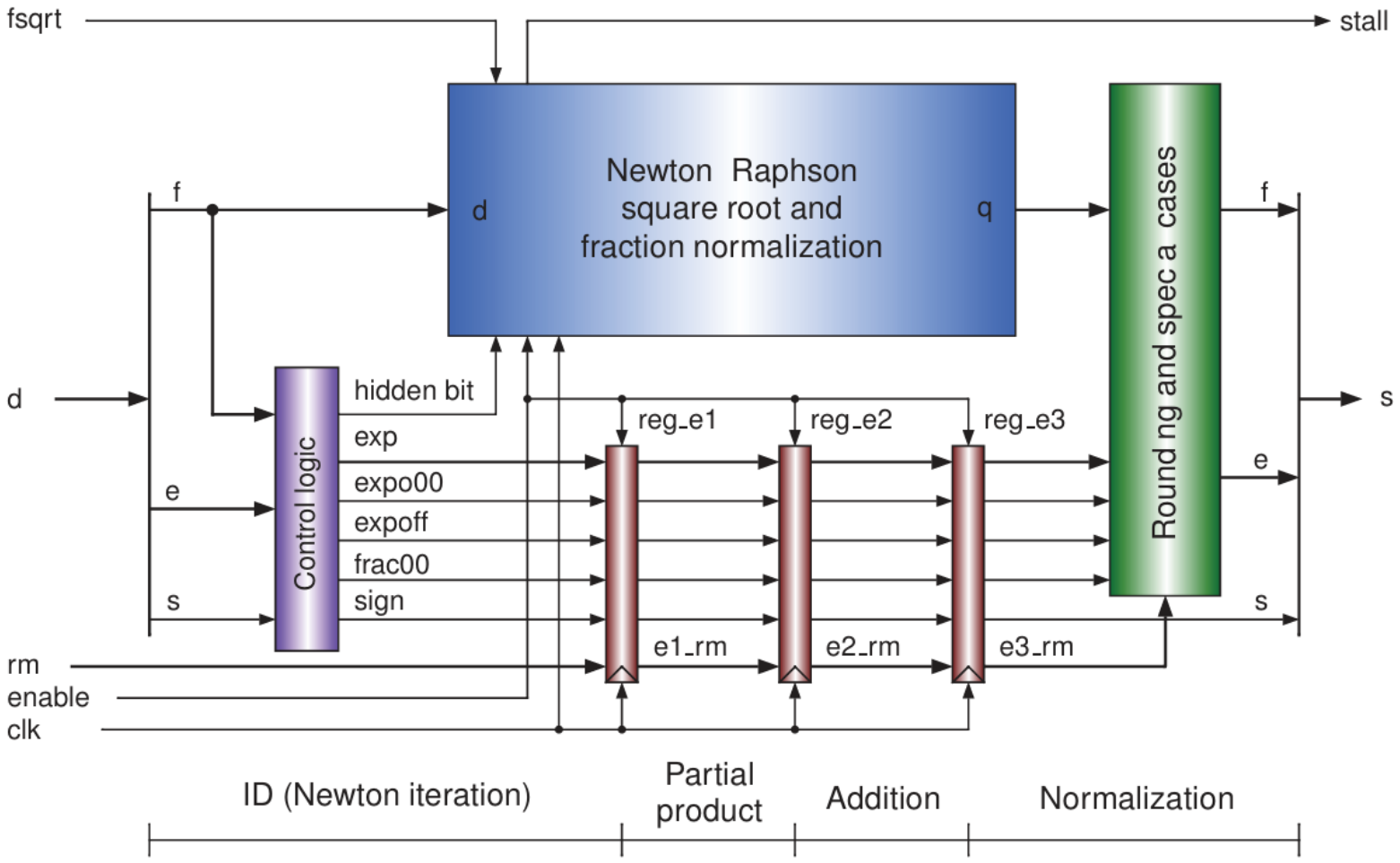
FSQRT

- Block diagram of the Newton-Raphson square rooter



FSQRT

- Block diagram of the floating-point square root



FSQRT

- Calculate the square root of 32-bit floating-point number *d* and the 32-bit floating-point result *s.*
- Newton-Raphson for square root reciprocal
  - ✓***X = 1 / sqrt(D)***
  - ✓***X_{i+1} = X_i * (3 – D * X_i ^2) / 2***
  - ✓***sqrt(D) = D * (1 / sqrt(D))***
- 3 stage pipeline for initial pre-processing and classification.
- Followed by the iterative sqrt reciprocal calculation and a final rounding or special case handling stage.

FSQRT

- **Pre-processing**
- Perform initial classification of the input *d.*
- Prepare the fraction and exponent for the core sqrt calculation.
- Identify special cases (zero, denormal, inf, and NaN)

```
17    wire          d_expo_is_00    = ~|d[30:23];
18    wire          d_expo_is_ff    =  &d[30:23];
19    wire          d_frac_is_00    = ~|d[22:00];
20    wire          sign            =   d[31];
```

FSQRT

- **Pre-processing**
- *exp_8*: calculate the initial exponent for the sqrt.
- *{1'b0,d[30:24]}*: perform e_d / 2 by right shift by 1 bit.
- + *8'd63*: add the bias 63 (127/2 = 63.5).
- + *d[23]*: If d[23] (LSB of exp) is 1 (odd), an extra 1 is added.
- For demoralized #, *exp_8* is 63.

```
21      //              eq      = (e_d >> 1)      + 63    + (e_d % 2)
22      wire    [7:0]   exp_8   = {1'b0,d[30:24]} + 8'd63 + d[23];       // normalized
23      //                      = 0               + 63    + 0 = 63       // denormalized
24      //              d_f24   = denormalized ? .f_d,0          : .1,f_d // shifted 1 bit
25      wire    [23:0]  d_f24   = d_expo_is_00 ? {d[22:0],1'b0} : {1'b1,d[22:0]};
```

FSQRT

- **Pre-processing**
- *d_f24*: prepares the 24-bit fraction for the sqrt calculation.
- If *d* is denormalized (*d_expo_is_00*), the fraction is used as is.
- If *d* is normalized, a hidden bit '1' is prepended to form *1.f.*

```
21      //              eq      = (e_d >> 1)      + 63    + (e_d % 2)
22      wire    [7:0]   exp_8   = {1'b0,d[30:24]} + 8'd63 + d[23];       // normalized
23      //                      = 0               + 63    + 0 = 63       // denormalized
24      //              d_f24   = denormalized ? .f_d,0          : .1,f_d // shifted 1 bit
25      wire    [23:0]  d_f24   = d_expo_is_00 ? {d[22:0],1'b0} : {1'b1,d[22:0]};
```

FSQRT

- **Pre-processing**
- *d_temp24*: adjust the fraction based on d[23].
- *shift_d*: The module for shifting *d_temp24* to left until the MSB is '1' or '01' depending on the expected format.
- *shamt_d*: record the even number of shifts.

```
26      //              tmp     = e_d is odd ? shift one more bit : 1 bit, no change
27      wire    [23:0]  d_temp24    = d[23]  ? {1'b0,d_f24[23:1]} : d_f24;
28      wire    [23:0]  d_frac24;   // .1xx...x or .01x...x for denormalized number
29      wire    [4:0]   shamt_d;   // shift amount, even number
30      shift_even_bits     shift_d (d_temp24, d_frac24, shamt_d);
```

FSQRT

- **Pre-processing**
- *exp0*: calculate the final effective exponent for core module.

```
31      // denormalized: e_q = 63 - shamt_d / 2
32      // normalized:   e_q = exp_8 - 0
33      wire    [7:0]   exp0 = exp_8 - {4'h0,shamt_d[4:1]};
```

FSQRT

- **Pipeline registers**

```
34      reg             e1_sign, e1_e00, e1_eff, e1_f00;
35      reg             e2_sign, e2_e00, e2_eff, e2_f00;
36      reg             e3_sign, e3_e00, e3_eff, e3_f00;
37      reg     [1:0]   e1_rm, e2_rm, e3_rm;
38      reg     [7:0]   e1_exp, e2_exp, e3_exp;
39      always @ (negedge clrn or posedge clk) begin
40          if (!clrn) begin  // 3 pipeline registers: reg_e1, reg_e2, and reg_e3
41              // reg_e1                reg_e2              reg_e3
42              e1_sign <= 0;             e2_sign <= 0;         e3_sign <= 0;
43              e1_rm   <= 0;             e2_rm   <= 0;         e3_rm   <= 0;
44              e1_exp  <= 0;             e2_exp  <= 0;         e3_exp  <= 0;
45              e1_e00  <= 0;             e2_e00  <= 0;         e3_e00  <= 0;
46              e1_eff  <= 0;             e2_eff  <= 0;         e3_eff  <= 0;
47              e1_f00  <= 0;             e2_f00  <= 0;         e3_f00  <= 0;
48          end else if (ena) begin
49              e1_sign <= sign;          e2_sign <= e1_sign;   e3_sign <= e2_sign;
50              e1_rm   <= rm;            e2_rm   <= e1_rm;     e3_rm   <= e2_rm;
51              e1_exp  <= exp0;          e2_exp  <= e1_exp;    e3_exp  <= e2_exp;
52              e1_e00  <= d_expo_is_00;  e2_e00  <= e1_e00;    e3_e00  <= e2_e00;
53              e1_eff  <= d_expo_is_ff;  e2_eff  <= e1_eff;    e3_eff  <= e2_eff;
54              e1_f00  <= d_frac_is_00;  e2_f00  <= e1_f00;    e3_f00  <= e2_f00;
55          end
56      end
```

FSQRT

- **Core sqrt reciprocal computation**
- *frac_newton*: core module
- *d_frac24*: the normalized and adjusted fraction.
- *fsqrt, ena, clk ,clrn*: control signals.
- frac0: output form *frac_newton*, the 32-bit fractional part of sqrt.
  - ✓***D * (1 / sqrt(D))***
- *busy, count, reg_x, stall*: control and debug outputs.
- *frac*: extract the most significant 26 bits of *frac0* and sticky bit.

```
57      wire    [31:0]  frac0;  // root = 1.xxx...x
58      root_newton24   frac_newton (d_frac24, fsqrt, ena, clk, clrn, frac0, busy, count, reg_x, stall);
59      wire    [26:0]  frac    = {frac0[31:6],|frac0[5:0]};    // gr,s
```

FSQRT

- **Core sqrt reciprocal computation**
- *frac_plus_1*: determine if the rounded fraction should be incremented by 1, based on rounding mode and grs.
- *frac_rnd*: perform the actual rounding of the fraction.

```
60    wire           frac_plus_1 =   ~e3_rm[1] & ~e3_rm[0] &  frac[3] &  frac[2] & ~frac[1] & ~frac[0]  |
61                                   ~e3_rm[1] & ~e3_rm[0] &            frac[2] & (frac[1] |  frac[0]) |
62                                   ~e3_rm[1] &  e3_rm[0] &          (frac[2] |  frac[1] |  frac[0]) &  e3_sign |
63                                    e3_rm[1] & ~e3_rm[0] &          (frac[2] |  frac[1] |  frac[0]) & ~e3_sign;
64    wire    [24:0] frac_rnd    = {1'b0,frac[26:3]} + frac_plus_1;
```

FSQRT

- **Core sqrt reciprocal computation**
- *expo_new*: adjust the exponent if the rounding caused the fraction to overflow.
- *frac_new*: select the final 23-bit fractional part after rounding and potential overflow from *frac_rnd*.

```
65      wire    [7:0]   expo_new    = frac_rnd[24] ? e3_exp + 8'h1 : e3_exp;
66      wire    [22:0]  frac_new    = frac_rnd[24] ? frac_rnd[23:1] : frac_rnd[22:0];
```

FSQRT

- **Special case handling and output**

```
67    assign          s   = final_result(e3_sign, e3_e00, e3_eff, e3_f00, {e3_sign,expo_new,frac_new});
68    function  [31:0] final_result;
69        input        d_sign,d_e00,d_eff,d_f00;
70        input [31:0] calc;
71        casex ({d_sign,d_e00,d_eff,d_f00})
72            4'b1xxx : final_result = NaN;       // -
73            4'b000x : final_result = calc;      // nor
74            4'b0100 : final_result = calc;      // den
75            4'b0010 : final_result = NaN;       // nan
76            4'b0011 : final_result = INF;       // inf
77            default : final_result = ZERO;      // 0
78        endcase
79    endfunction
```

CPU/FPU Pipeline Model

- The pipeline models of the computational floating-point instructions are more complex due to the iterations for the float division and square root instructions.
- FPU execute the float instructions, float add/sub/mul/div/sqrt.
- These instructions fetch operands from a float-point register file.
- Therefore, it needs to implement the float load/store instructions.

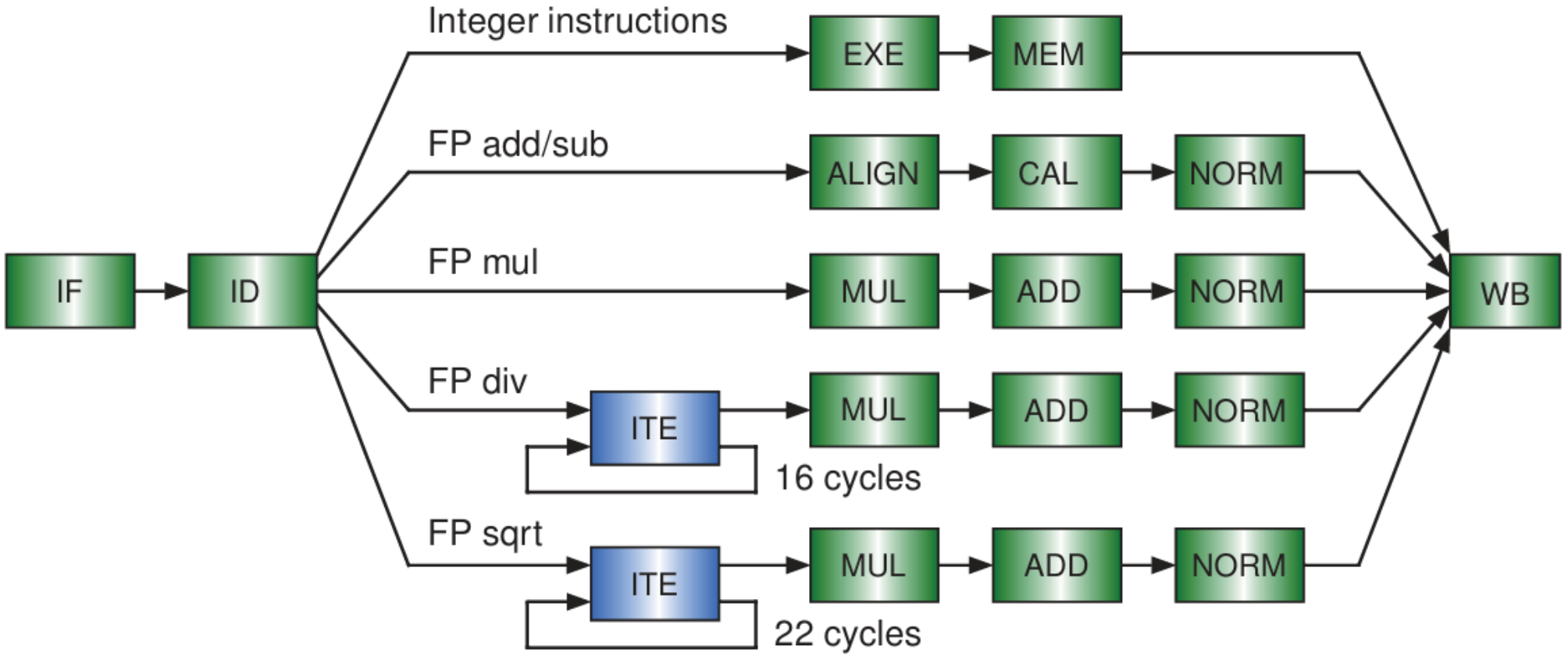
CPU/FPU Pipeline Model

- FPU instructions
- add.s/sub.s/mul.s/div.s *fd, fs, ft* # *fd* ← *fs* op *ft*;
- sqrt.s *fd*, fs # ← root (*fs*);
- lwc1 *ft*, offset(*rs*) # *ft* ← memory[*rs* + *offset*];
- swc1 *ft*, offset(*rs*) # ← memory[*rs* + *offset*];

Inst.	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	Description
lwc1	110001	rs	ft		offset		Load FP word
swc1	111001	rs	ft		offset		Store FP word
add.s	010001	10000	ft	fs	fd	000000	FP add
sub.s	010001	10000	ft	fs	fd	000001	FP subtract
mul.s	010001	10000	ft	fs	fd	000010	FP multiplication
div.s	010001	10000	ft	fs	fd	000011	FP division
sqrt.s	010001	10000	00000	fs	fd	000100	FP square root

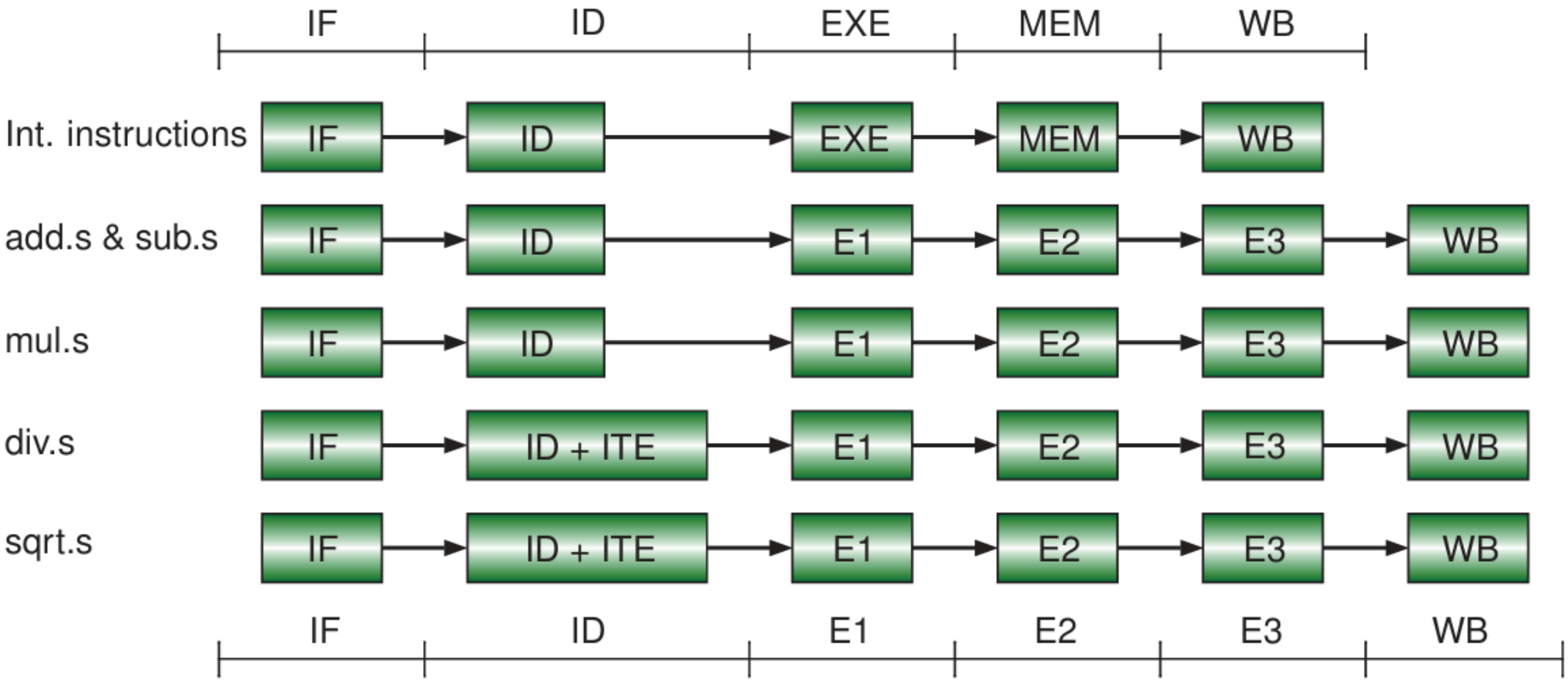
CPU/FPU Pipeline Model

- Pipeline models of the IU and the FPU



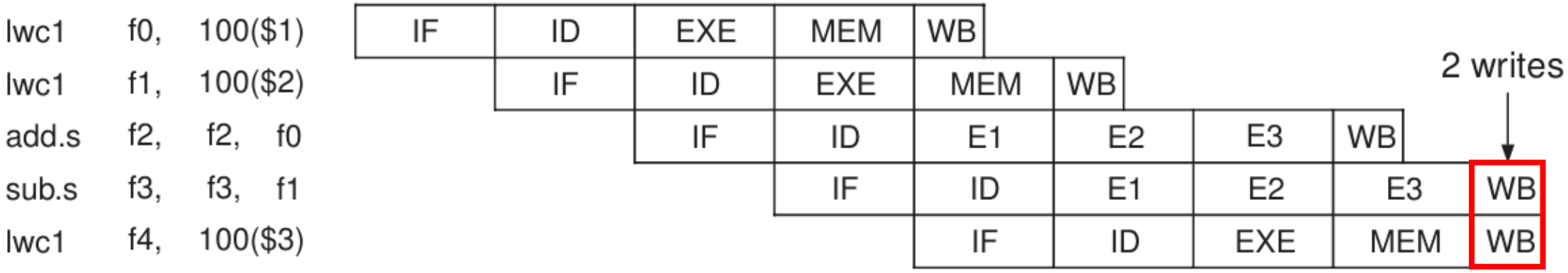
CPU/FPU Pipeline Model

- Unified floating-point pipeline model



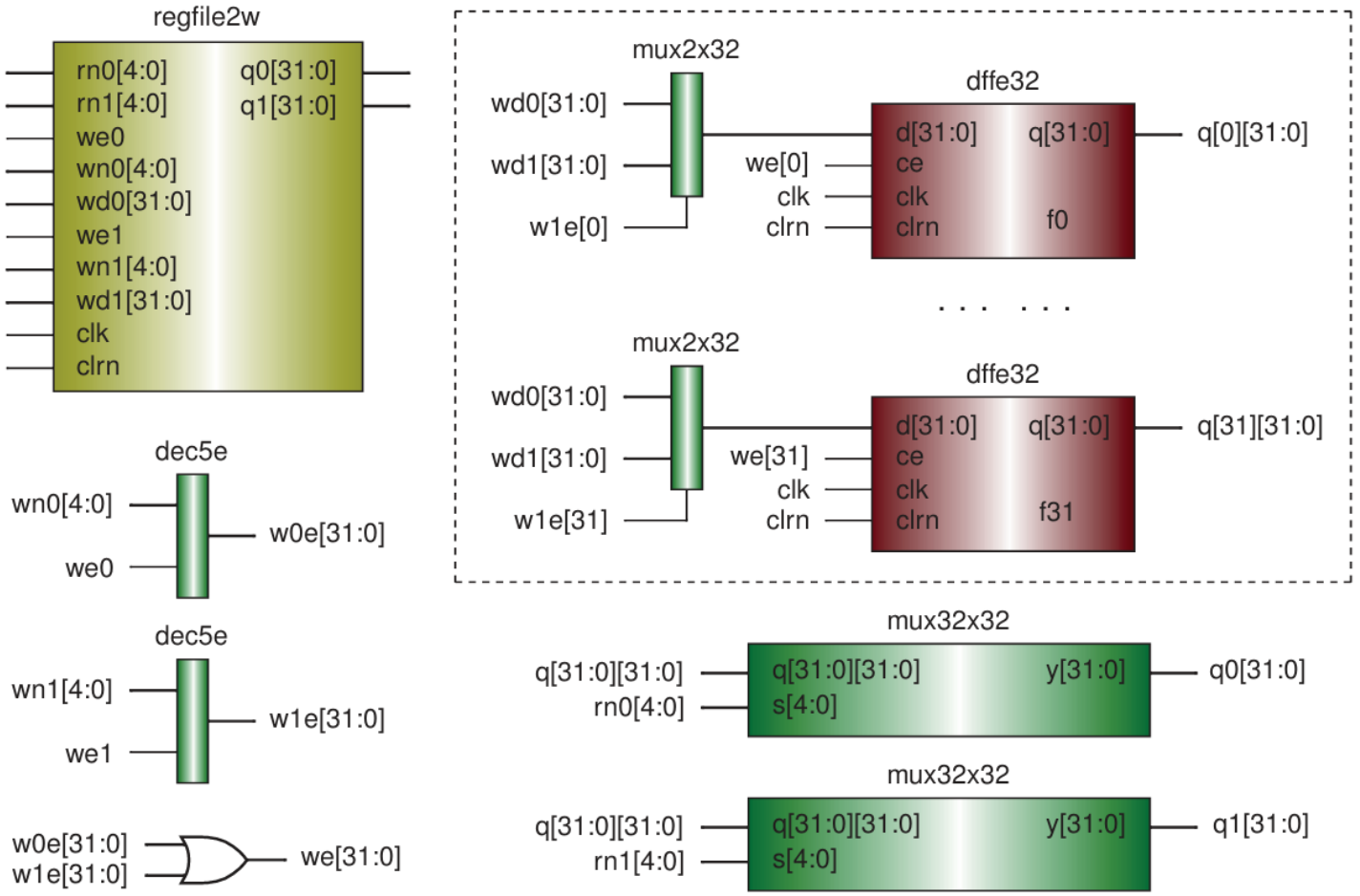
Register File with Two Write Ports

- There are 32 registers in the floating-point register file.
- Different from the general-purpose RF, register 0 in the floating-point RF is a regular register.
- In this floating-point pipeline model, there is a case where two writes may appear at a same clock cycle.



Register File with Two Write Ports

- Schematic diagram of a register file with two write ports



Register File with Two Write Ports

- **Register file with two write ports**

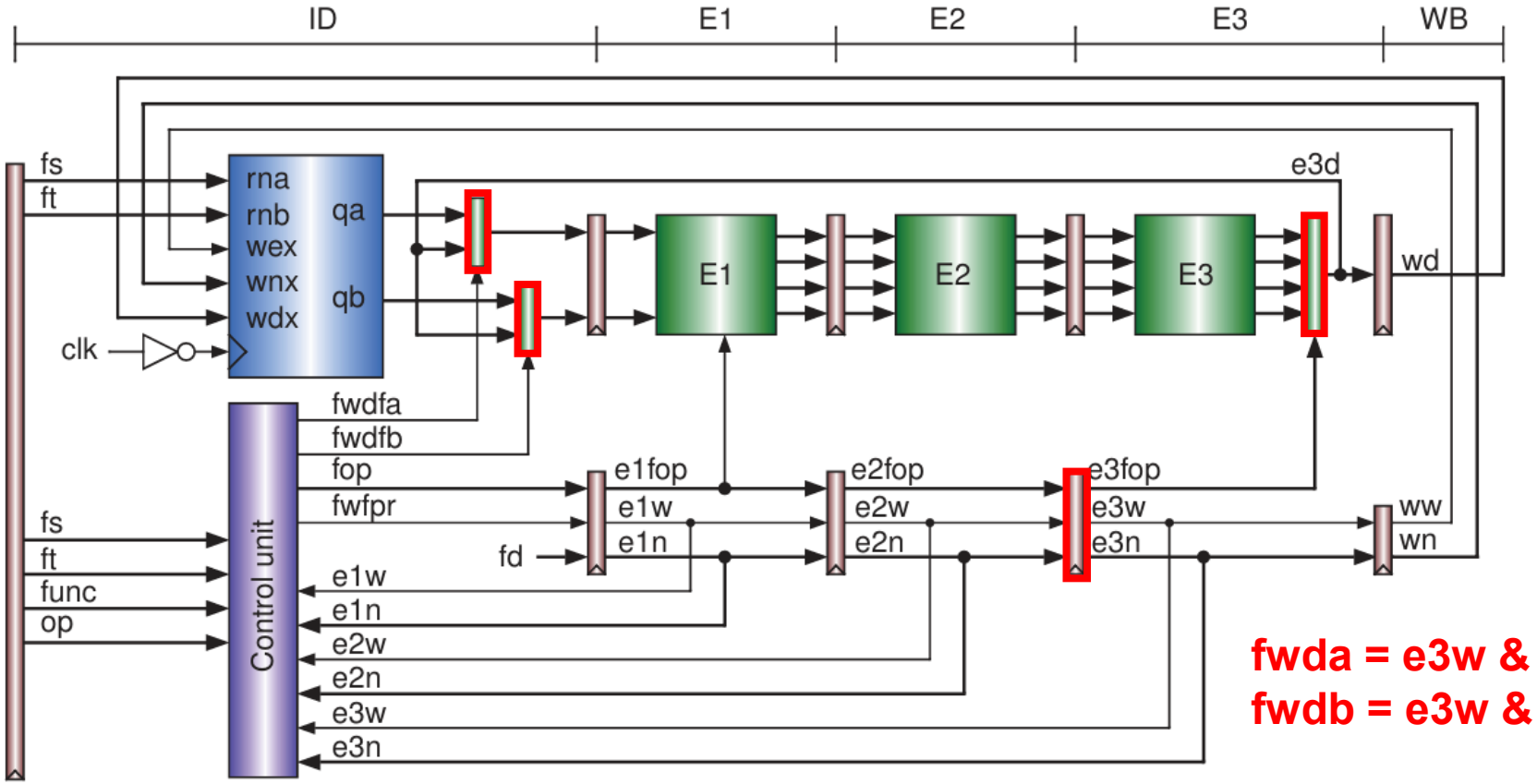
```
1  module regfile2w (
2      input   [31:0]  dx, dy,
3      input   [4:0]   rna, rnb, wnx, wny,
4      input           wex, wey,
5      input           clk, clrn,
6      output  [31:0]  qa, qb,
7  );
8      reg     [31:0]  register [0:31];
9      assign          qa  = register[rna];
10     assign          qb  = register[rnb];
11
12     integer i;
13     always @(posedge clk or negedge clrn) begin
14         if (!clrn) begin
15             for (i=0; i<32; i=i+1)
16                 register[i] <= 0;
17         end else begin
18             if (wey) // write port y has ahigher priority
19                 register[wny] <= dy;
20             if (wex && ( !wey || (wnx != wny) ))
21                 register[wnx] <= dx;
22         end
23     end
24 endmodule
```

Data Dependency and Pipeline Stalls

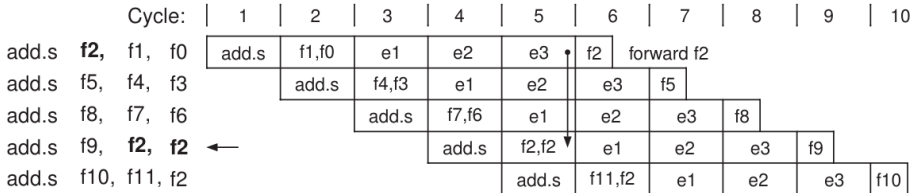
- The data dependencies among float computational instructions.
- The data dependencies between lwc1/swc1 instructions.
- The pipeline stalls for the execution of the div.s and sqrt.s.

Data Dependency and Pipeline Stalls

- No-stall case for floating point data hazard.

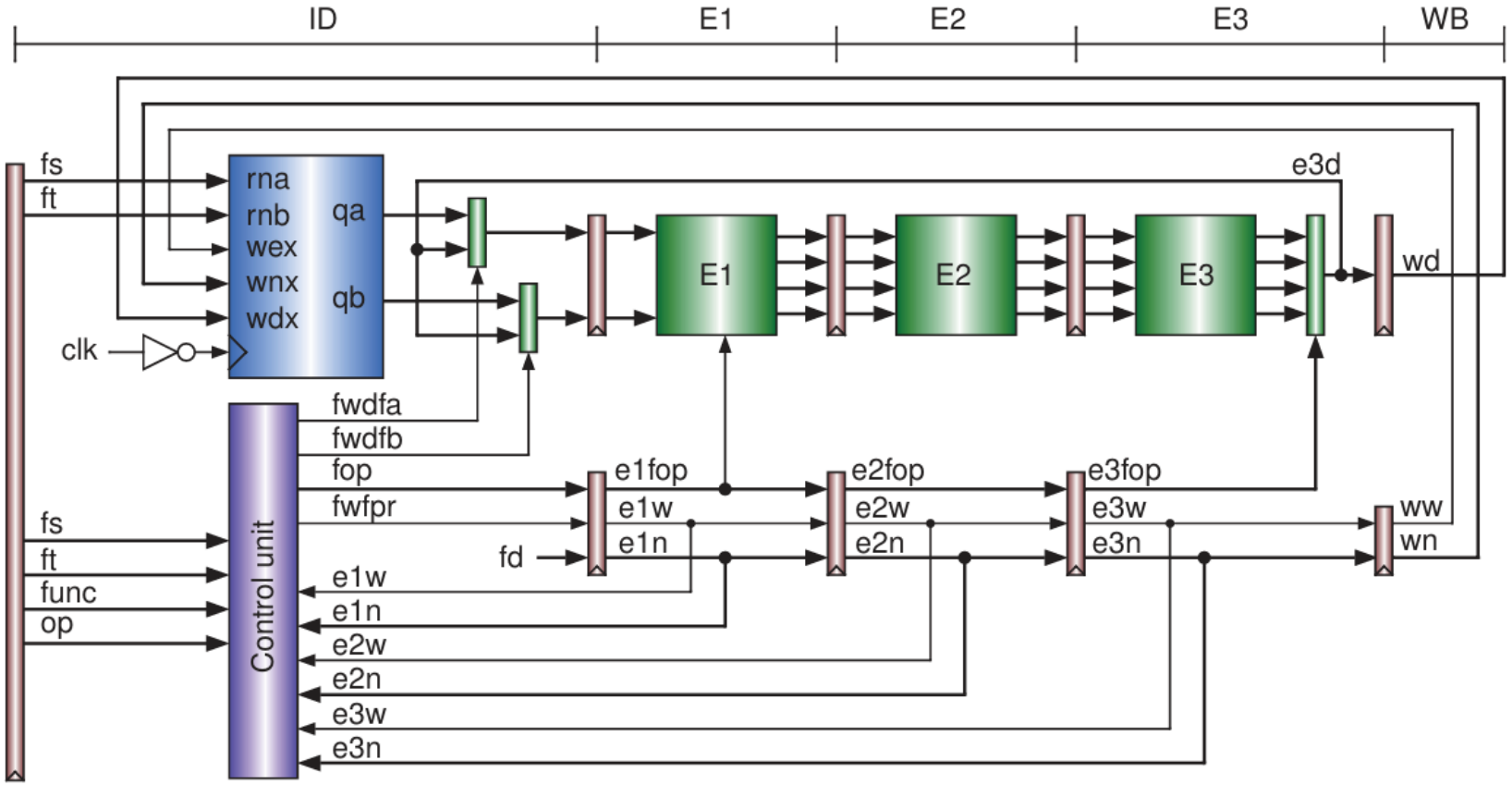


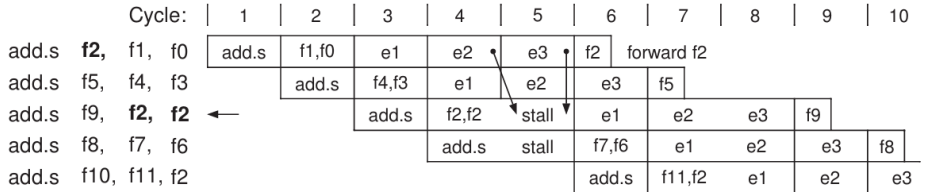
fwda = e3w & (e3n == fs);
fwdb = e3w & (e3n == ft);



Data Dependency and Pipeline Stalls

- One-cycle stall case for floating-point data hazard.

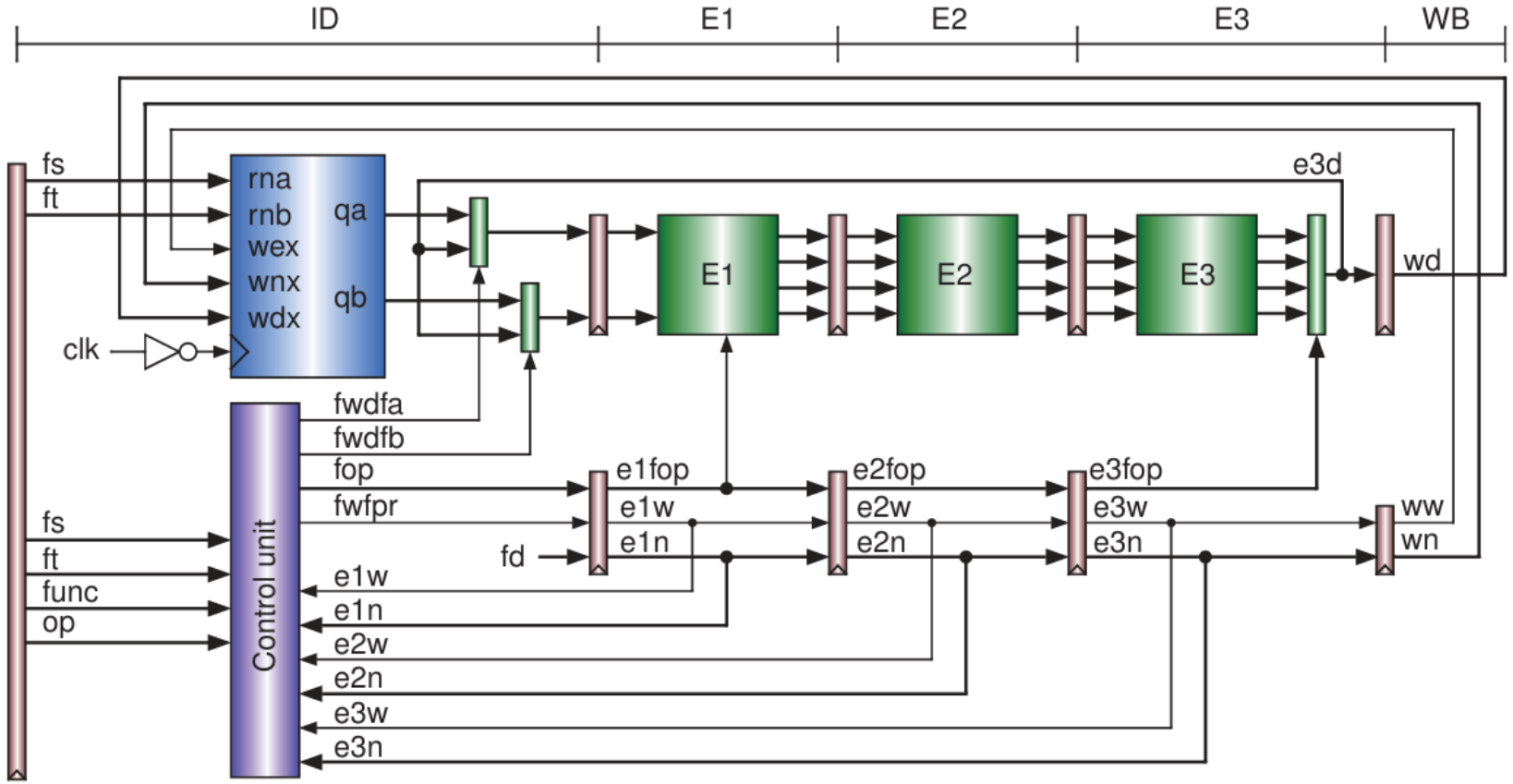


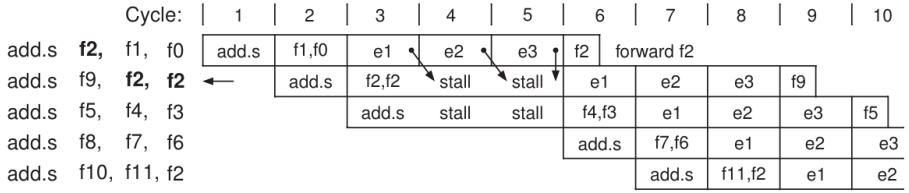



```
i_fs = i_fadd | i_fsub | i_fmul | i_fdiv | i_fsqrt;
i_ft = i_fadd | i_fsub | i_fmul | i_fdiv;
stall_fp = (e1w & ( i_fs & (e1n == fs) | i_ft & (e1n == ft) )) |
           (e2w & ( i_fs & (e2n == fs) | i_ft & (e2n == ft) ));
```

Data Dependency and Pipeline Stalls

- Two-cycle stall case for floating-point data hazard.

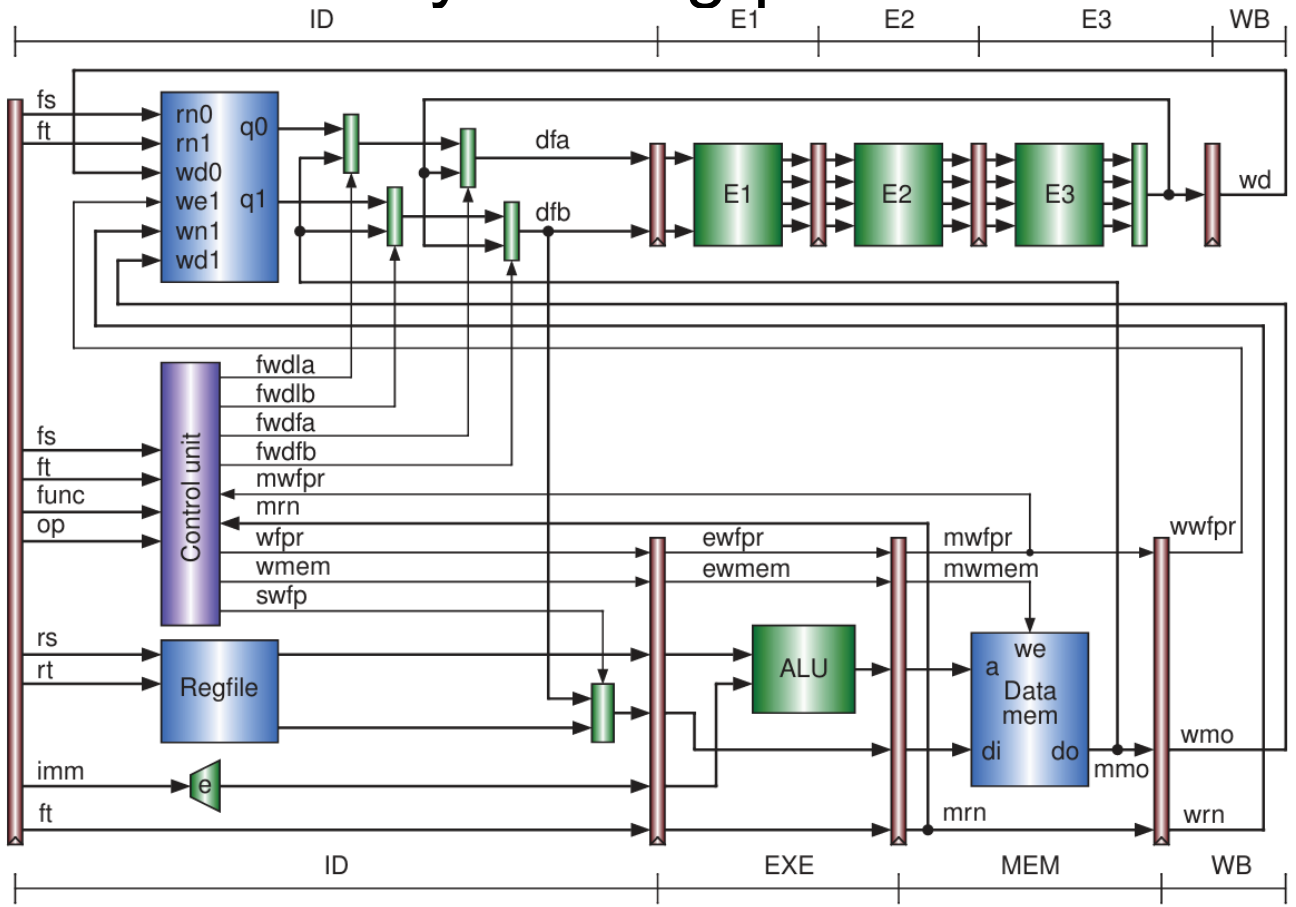


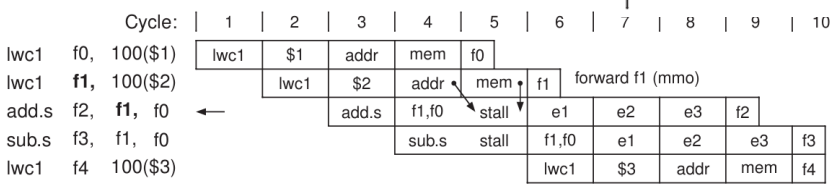



```
i_fs = i_fadd | i_fsub | i_fmul | i_fdiv | i_fsqrt;
i_ft = i_fadd | i_fsub | i_fmul | i_fdiv;
stall_fp = (e1w & ( i_fs & (e1n == fs) | i_ft & (e1n == ft) )) |
           (e2w & ( i_fs & (e2n == fs) | i_ft & (e2n == ft) ));
```

Data Dependency and Pipeline Stalls

- One-cycle stall caused by floating-point load.

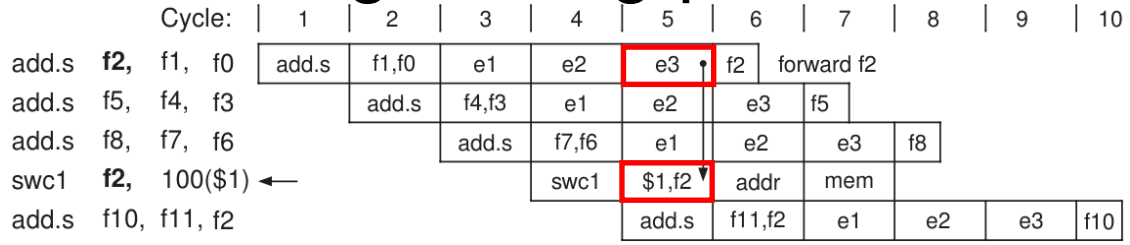




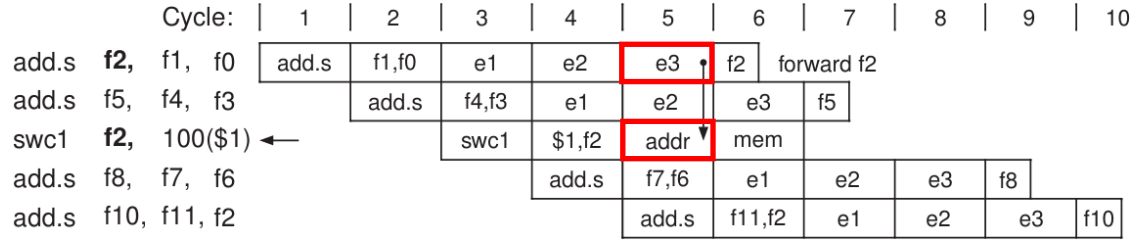

```
fwdla = mwfpr & (mrn == fs); // forward mmo to fp a
fwdlb = mwfpr & (mrn == fs); // forward mmo to fp b
stall_lwc1 = ewfpr & (i_fs & (ern == fs) | i_ft (ern == ft));
```

Data Dependency and Pipeline Stalls

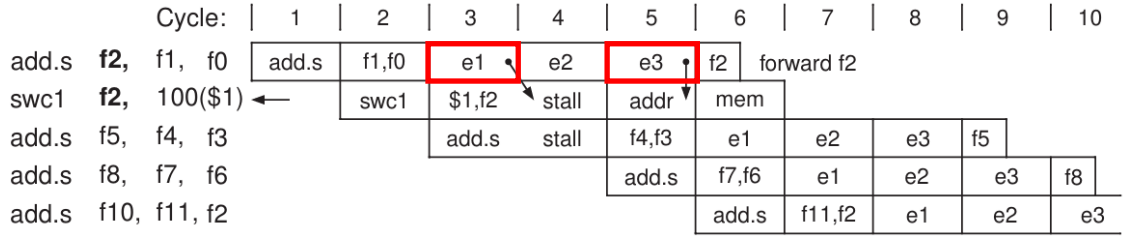
- Forwarding floating-point data to the ID stage.



- Forwarding floating-point data to the EXE stage.



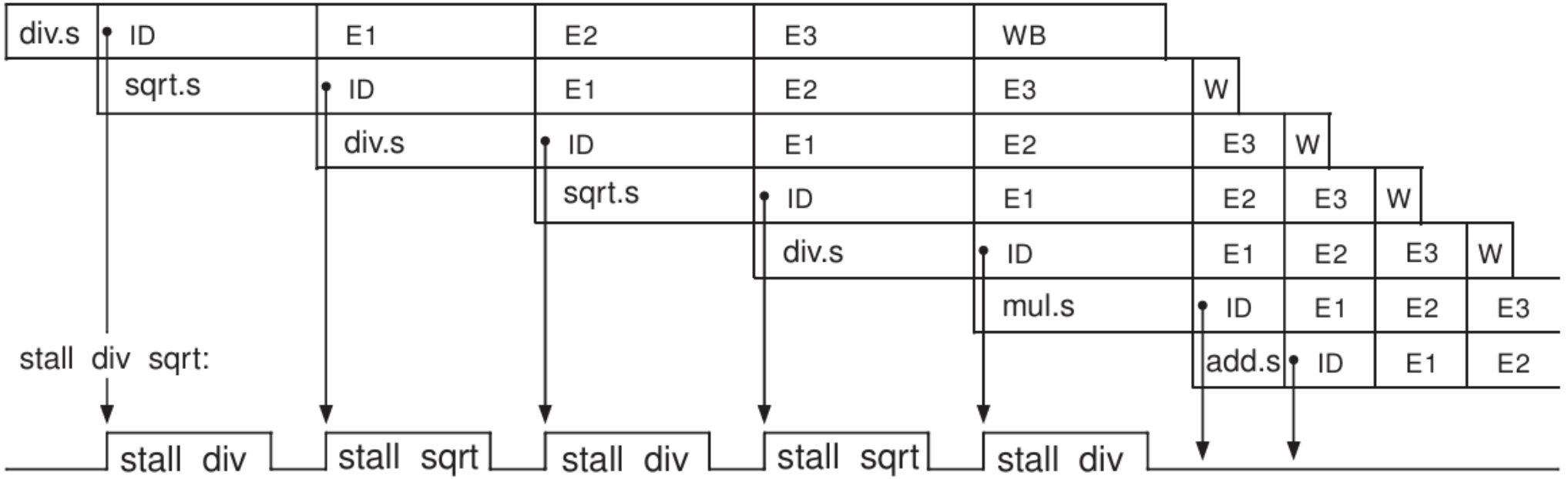
- One-cycle stall caused by floating-point store.




```
fwdf  = swfp & e3w & (ft == e3n); // forward to id  stage
fwdfe = swfp & e2w & (ft == e2n); // forward to exe stage
stall_swc1 = swfp & e1w & (ft == e1n); // stall cause by swc1
```

Data Dependency and Pipeline Stalls

- Pipeline stall caused by division and square root.
  - ✓caused by Newton-Raphson iterations




```
stall_div_sqrt = stall_div | stall_sqrt;
w_fp_pipe_reg = ~stall_div_sqrt; // fp pipe_reg write enable
```

Data Dependency and Pipeline Stalls

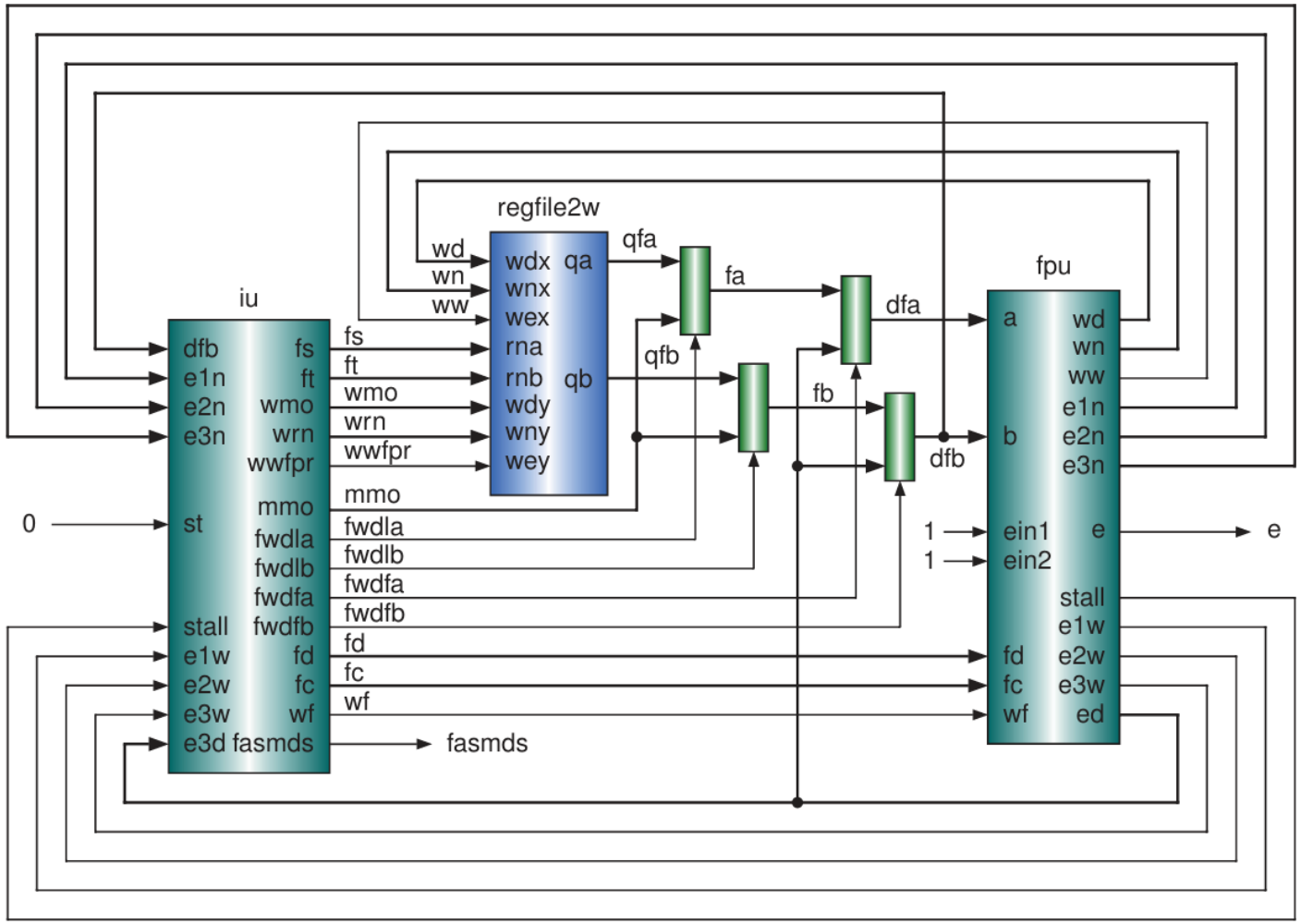
- The stall signals
  - ✓stall_lw: casued by integer data dependency with lw;
  - ✓stall_lwc1: caused by float data dependency with lwc1;
  - ✓stall_swc1: caused by float data dependency with swc1;
  - ✓stall_fp: caused by float data dependency between float instructions;
  - ✓stall _div_sqrt: caused by Newton-Raphson iterations;
- Every signal disable the writings to the PC and IF/ID pipeline registers.
  - ✓wpcir = ~(stall_div_sqrt | stall_others);
  - ✓stall_others = stall_lw | stall_fp | stall_lwc1 | stall_swc1;

Data Dependency and Pipeline Stalls

- To prevent an instruction from executing twice or more, cancel the instruction during the stall by disabling write signals.
  - ✓wreg = (i_add | i_sub | i_and | i_or | i_xor | i_sll | i_srl | i_sra | i_addi | i_andi | i_ori | i_xori | i_lw | i_lui | i_jal) & wpcir;
  - ✓wmem = (i_sw | i_swc1) & wpcir;
  - ✓wf = (i_fadd | i_fsub | i_fmul | i_fdiv | i_fsqrt) & wpcir;
  - ✓wfpr = i_lwc1 & wpcir;

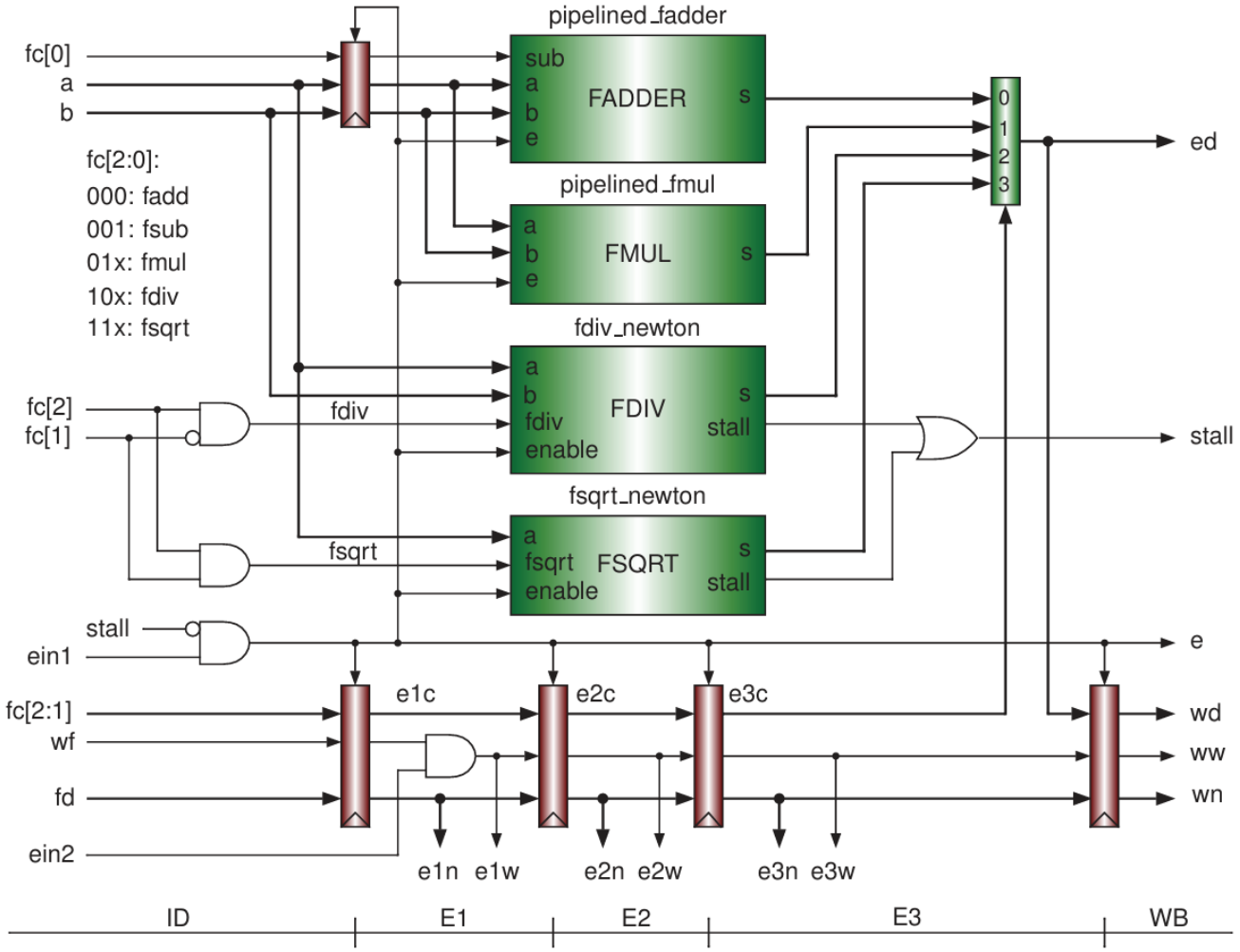
Pipelined CPU/FPU Design

- Block diagram of CPU with one IU and one FPU



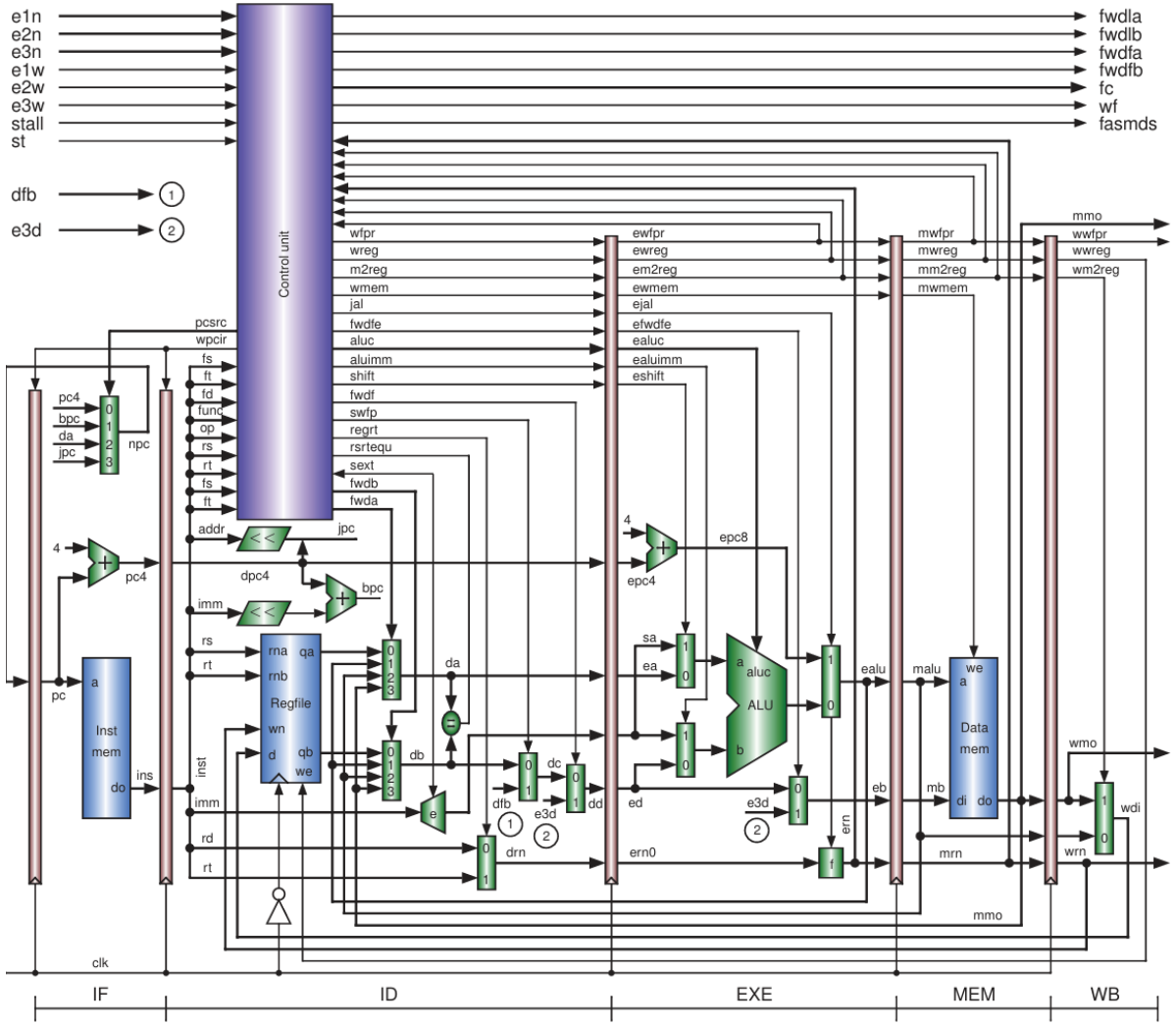
Pipelined CPU/FPU Design

- Block diagram of the floating-point unit



Pipelined CPU/FPU Design

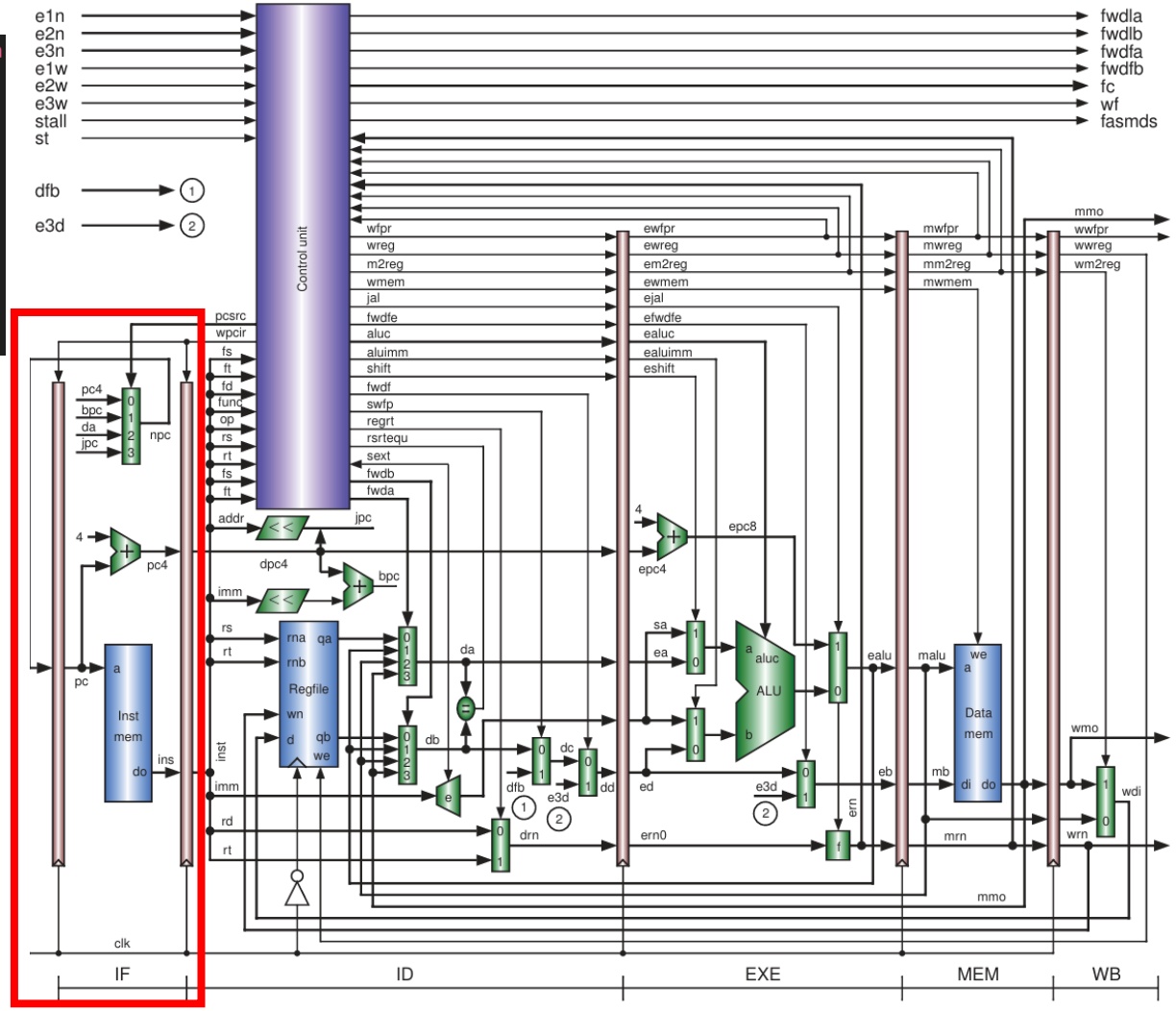
- Block diagram of the integer unit



Pipelined CPU/FPU Design

- Block diagram of the integer unit

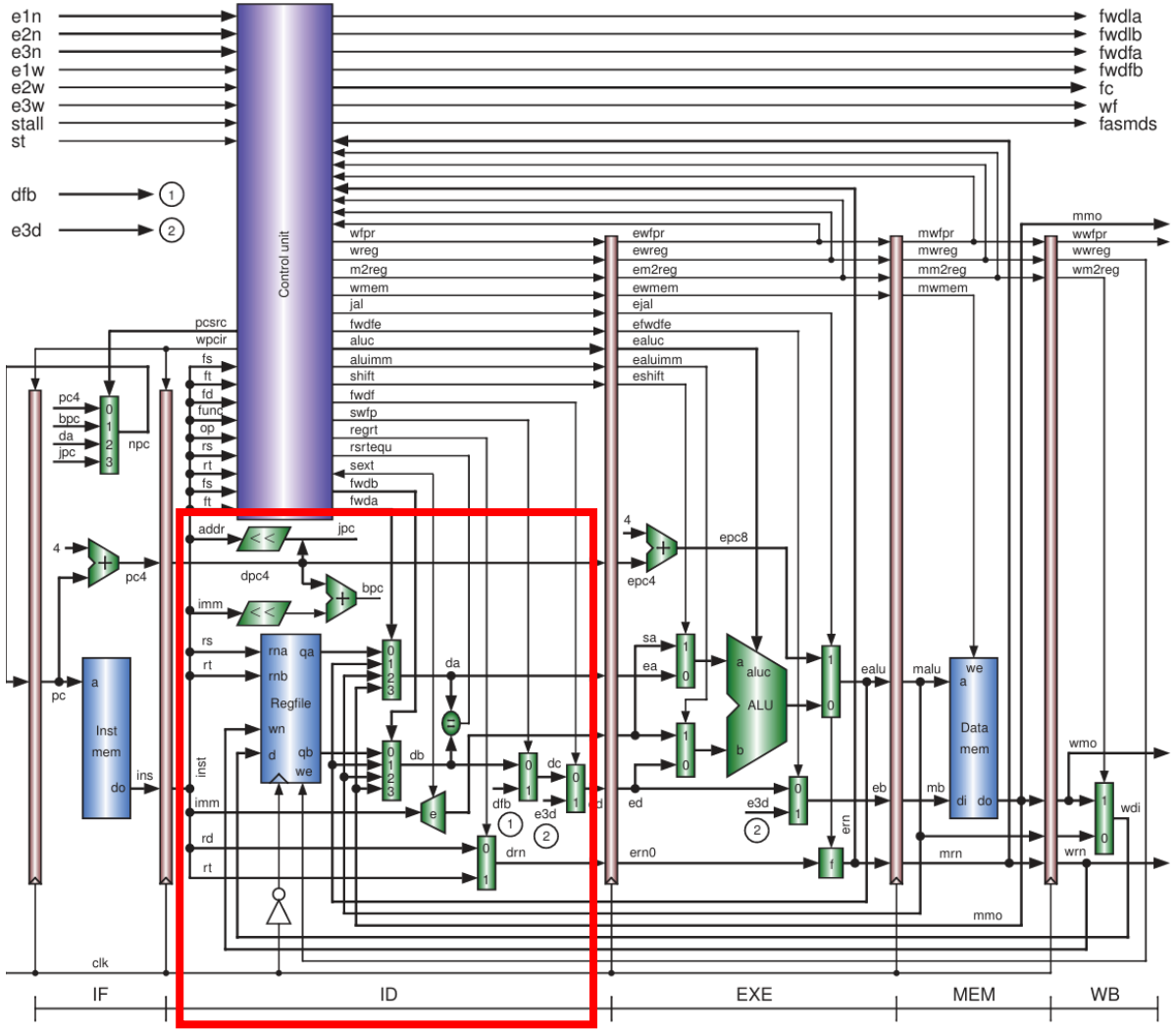
```
44    always @(posedge clk or negedge clrn) begin
45        if (!clrn) begin
46            pc      <= 0;
47            dpc4    <= 0;
48            inst    <= 0;
49        end else if (wpcir) begin
50            pc      <= npc;
51            dpc4    <= pc4;
52            inst    <= ins;
53        end
54    end
```



Pipelined CPU/FPU Design

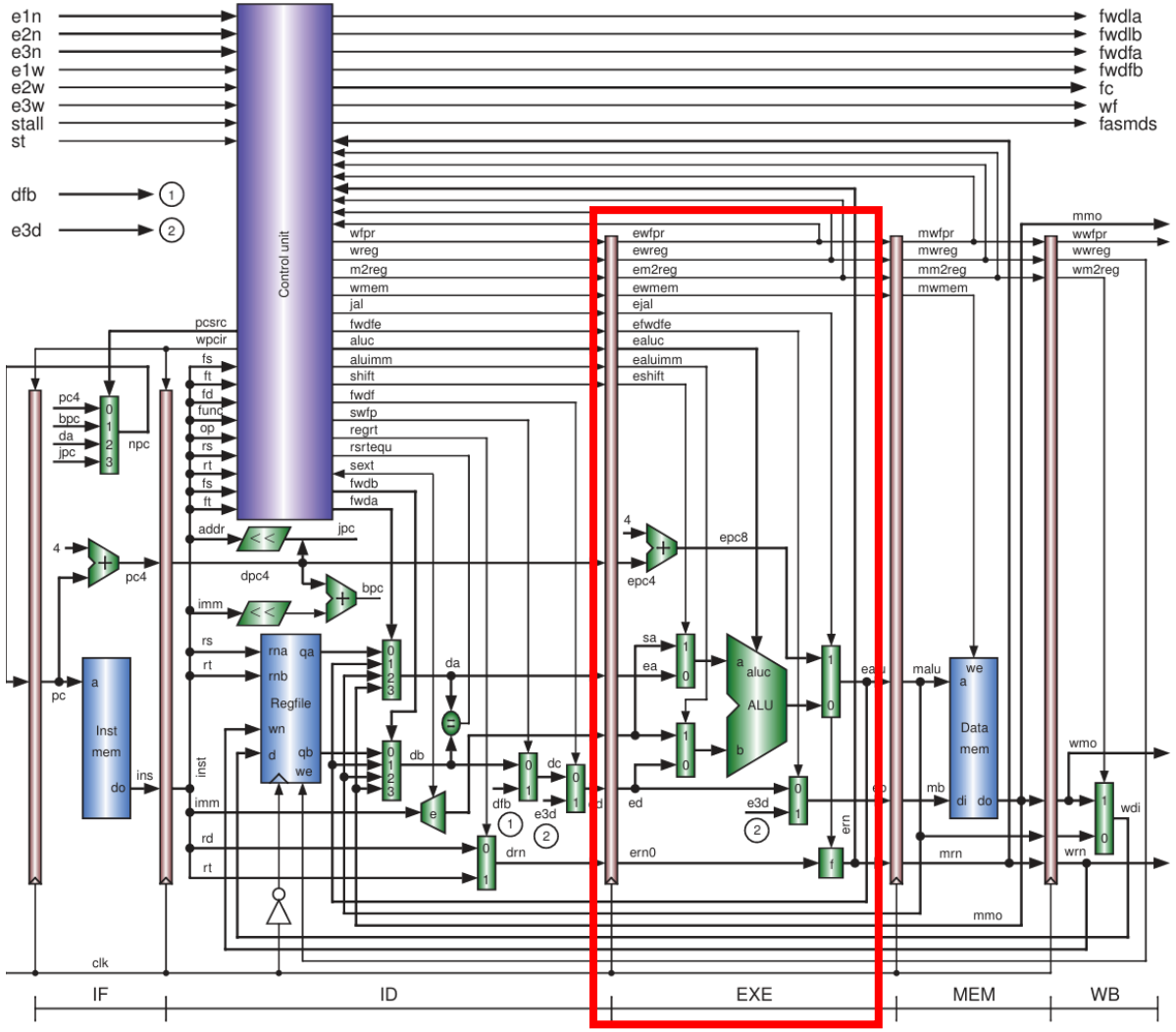
- Block diagram of the integer unit

```
55    wire            swfp, regrt, sext, fwdf, fwdfe, wfpr;
56    assign          op      = inst[31:26];
57    assign          rs      = inst[25:21];
58    assign          rt      = inst[20:16];
59    assign          rd      = inst[15:11];
60    assign          ft      = inst[20:16];
61    assign          fs      = inst[15:11];
62    assign          fd      = inst[10:06];
63    assign          func    = inst[05:00];
64    assign          simm    = {{16{sext&inst[15]}},inst[15:0]};
65    assign          jpc     = {dpc4[31:28],inst[25:0],2'b00};
66 //  cla32           br_addr (dpc4,{simm[29:00],2'b00},1'b0,bpc);
67    assign          bpc     = dpc4 + {simm[29:00],2'b00};
68    regfile         rf (
69        // input ports
70        .clk(~clk),             .clrn(clrn),
71        .wn(wrn),               .we(wwreg),
72        .rna(rs),               .rnb(rt),
73        .d(wdi),
74        // output ports
75        .qa(qa),                .qb(qb)
76    );
77    mux4x32         alu_a (qa, ealu, malu, mmo, fwda, da);
78    mux4x32         alu_b (qb, ealu, malu, mmo, fwdb, db);
79    mux2x32         store_f (db, dfb, swfp, dc);
80    mux2x32         fwd_f_d (dc, e3d, fwdf, dd);
81    wire            rsrtequ = ~|(da^db);
82    mux2x5          des_reg_no (rd, rt, regrt, drn);
```



Pipelined CPU/FPU Design

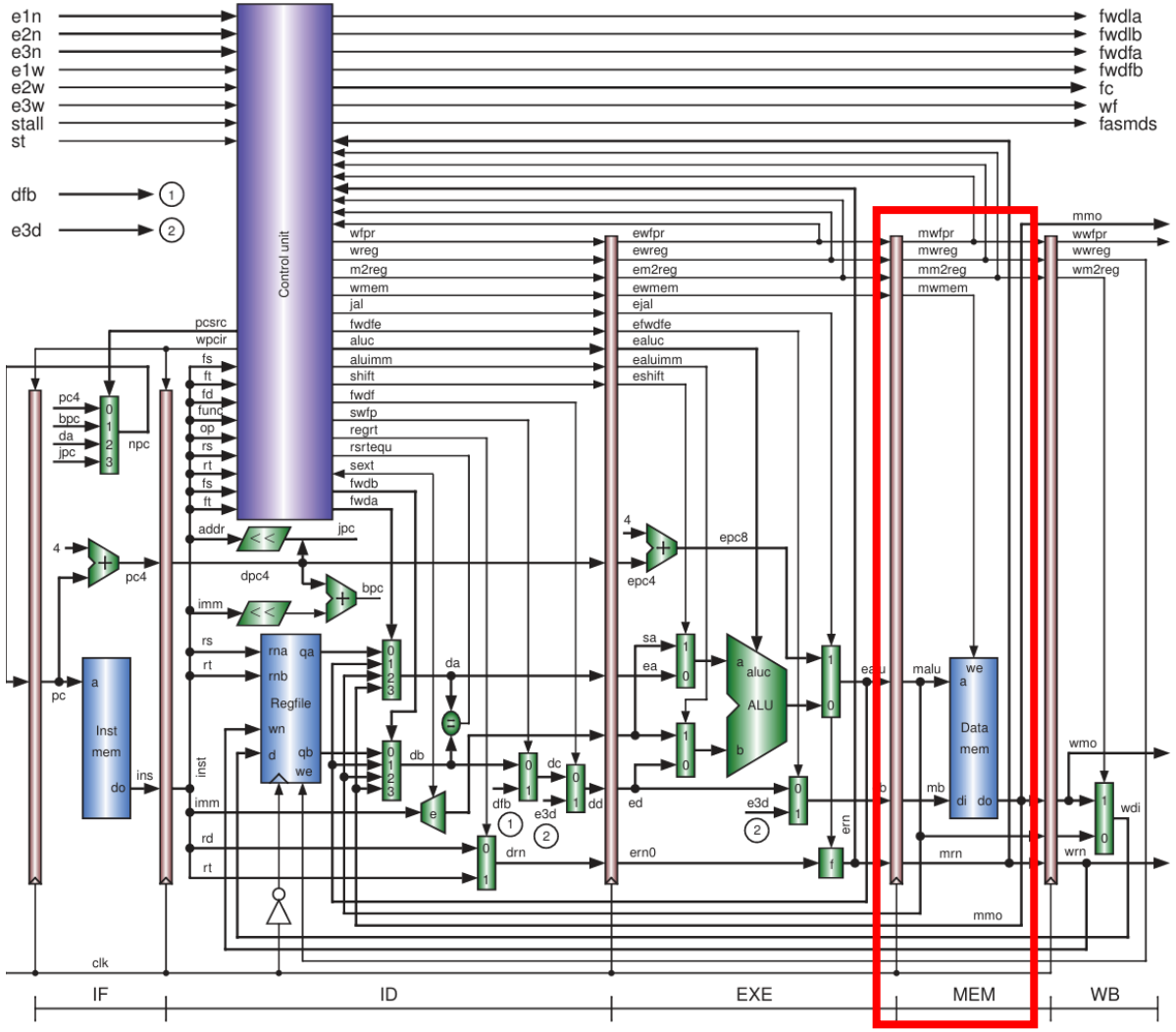
- Block diagram of the integer unit




```
105     always @(posedge clk or negedge clrn) begin
106         if (!clrn) begin
107             ewfpr   <= 0;               ewreg   <= 0;
108             em2reg  <= 0;               ewmem   <= 0;
109             ejal    <= 0;               ealuimm <= 0;
110             efwdfe  <= 0;               ealuc   <= 0;
111             eshift  <= 0;               epc4    <= 0;
112             ea      <= 0;               ed      <= 0;
113             eimm    <= 0;               ern0    <= 0;
114         end else begin
115             ewfpr   <= wfpr;            ewreg   <= wreg;
116             em2reg  <= m2reg;           ewmem   <= wmem;
117             ejal    <= jal;             ealuimm <= aluimm;
118             efwdfe  <= fwdfe;           ealuc   <= aluc;
119             eshift  <= shift;           epc4    <= dpc4;
120             ea      <= da;              ed      <= dd;
121             eimm    <= simm;            ern0    <= drn;
122         end
123     end
124 //  cla32           ret_addr (epc4, 32'h4, 1'b0, epc8);
125     assign          epc8    = epc4 + 32'h4;
126     assign          sa      = {eimm[5:0],eimm[31:6]};
127     mux2x32         alu_ina (ea, sa, eshift, alua);
128     mux2x32         alu_inb (ed, eimm, ealuimm, alub);
129     mux2x32         save_pc8 (ealu0, epc8, ejal, ealu);
130     alu             al_unit (alua, alub, ealuc, ealu0, z);
131     assign          ern     = ern0 | {5{jal}};
132     mux2x32         fwd_f_e (ed, e3d, efwdfe, eb);
```

Pipelined CPU/FPU Design

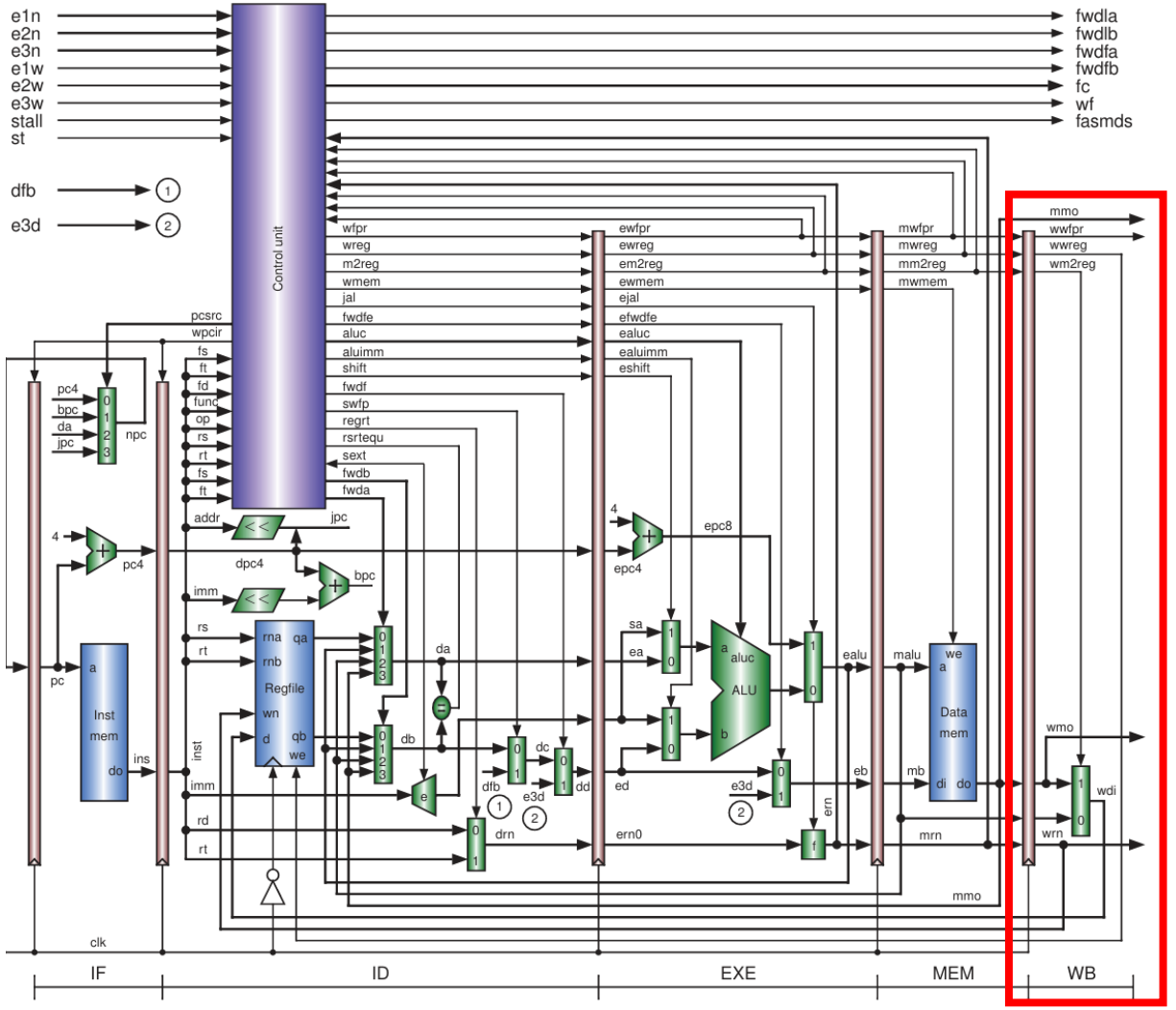
- Block diagram of the integer unit




```
133     always @(posedge clk or negedge clrn) begin
134         if (!clrn) begin
135             mwfpr   <= 0;               mwreg   <= 0;
136             mm2reg  <= 0;               mwmem   <= 0;
137             malu    <= 0;               mb      <= 0;
138             mrn     <= 0;
139         end else begin
140             mwfpr   <= ewfpr;           mwreg   <= ewreg;
141             mm2reg  <= em2reg;          mwmem   <= ewmem;
142             malu    <= ealu;            mb      <= eb;
143             mrn     <= ern;
144         end
145     end
146 //  data_mem        d_mem (mwmem, malu, mb, memclk, mmo);
147     data_mem        d_mem (
148         .addra(malu[6:2]),
149         .clka(memclk),
150         .dina(mb),
151         .douta(mmo),
152         .wea(mwmem)
153     );
```

Pipelined CPU/FPU Design

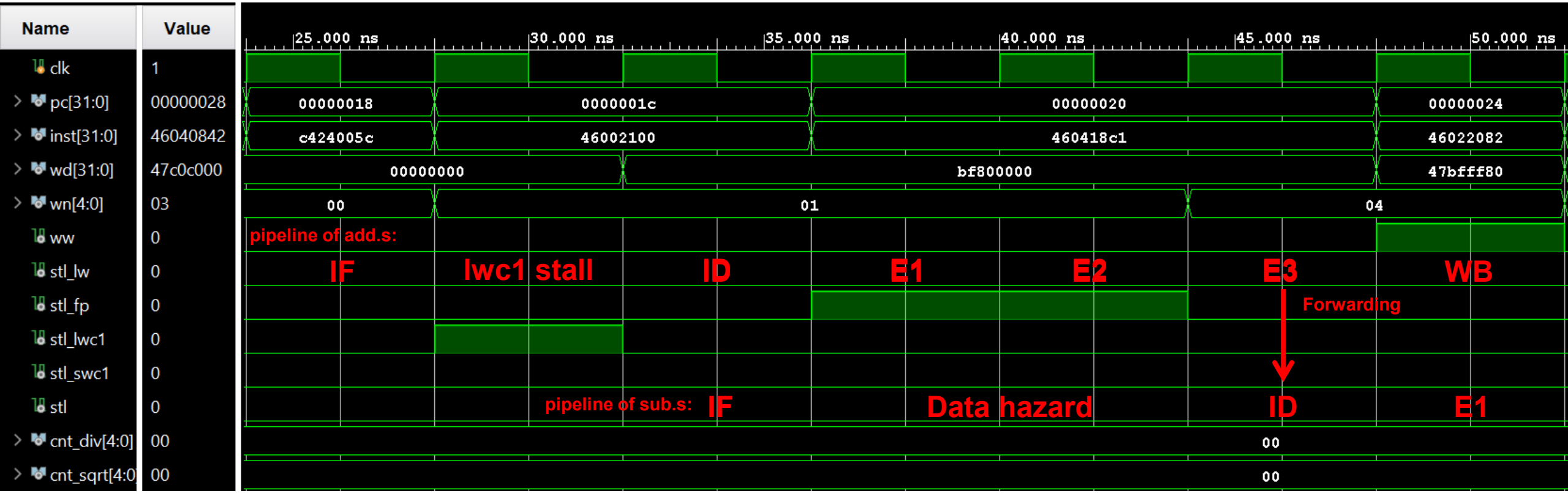
- Block diagram of the integer unit




```
154     always @(posedge clk or negedge clrn) begin
155         if (!clrn) begin
156             wwfpr   <= 0;               wwreg   <= 0;
157             wm2reg  <= 0;               wmo     <= 0;
158             walu    <= 0;               wrn     <= 0;
159         end else begin
160             wwfpr   <= mwfpr;           wwreg   <= mwreg;
161             wm2reg  <= mm2reg;          wmo     <= mmo;
162             walu    <= malu;            wrn     <= mrn;
163         end
164     end
165     mux2x32         wb_sel (walu, wmo, wm2reg, wdi);
```

Simulation

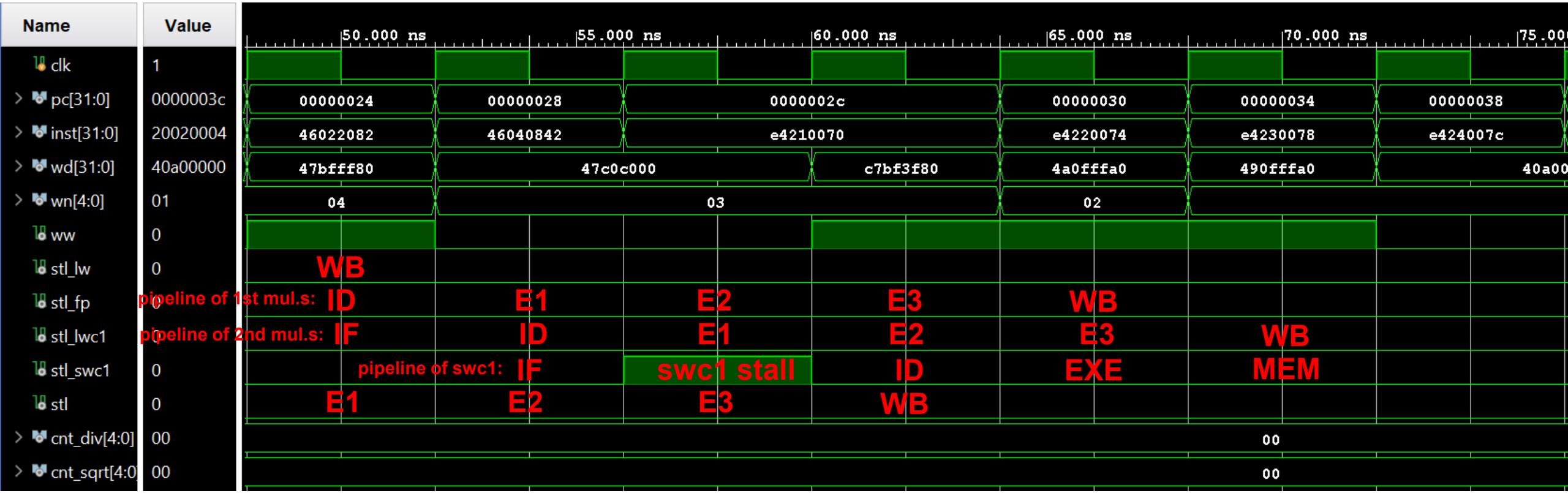
- Waveform 1 of CPU/FPU (lwc1, add.s, and sub.s)




```
5 : c424005c; % (14)        lwc1   f4    92($1)   # load fp data          %
6 : 46002100; % (18)        add.s  f4    f4   f0   # f4: stall 1          %
7 : 460418c1; % (1c)        sub.s  f3,   f3,  f4   # f4: stall 2          %
```

Simulation

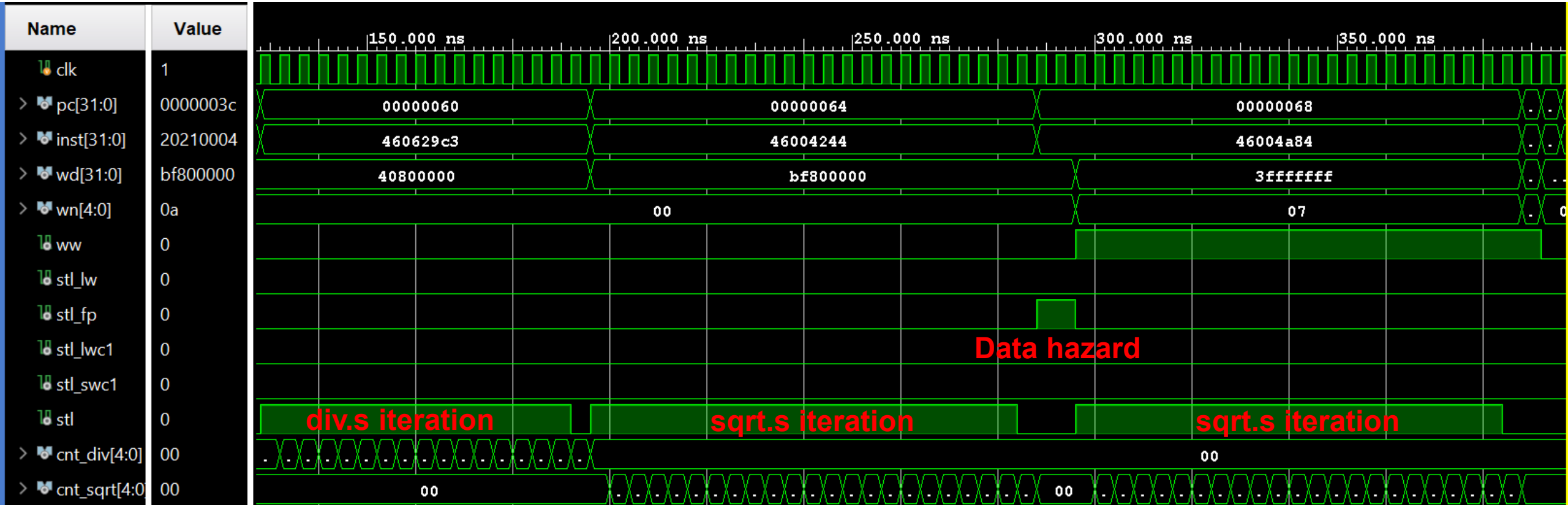
- Waveform 2 of CPU/FPU (sub.s, mul.s ,mul.s, and swc1)




```
6 : 46002100; % (18)        add.s  f4,   f4, f0  # f4: stall 1          %
7 : 460418c1; % (1c)        sub.s  f3,   f3, f4  # f4: stall 2          %
8 : 46022082; % (20)        mul.s  f2,   f4, f2  # mul                  %
9 : 46040842; % (24)        mul.s  f1,   f1, f4  # mul                  %
a : e4210070; % (28)        swc1   f1, 112($1)   # f1: stall 1          %
```

Simulation

- Waveform 3 of CPU/FPU(div.s, sqrt.s, and sqrt.s)




```
17 : 460629c3; % (5c)        div.s  f7,  f5, f6   # div                %
18 : 46004244; % (60)        sqrt.s f9,  f8       # sqrt               %
19 : 46004a84; % (64)        sqrt.s f10, f9       # sqrt               %
```

References

[1] Yamin Li, *Computer Principles and Design in Verilog HDL*, Wiley, 2016.