コンピュータ構成と設計（10）

浮動小数点数の演算

李　亜民

2024 年 12 月 05 日 (木)

クイズ：十進数の演算（有効数字）

$A = -1.0 \times 10^{-2}$　(有効数字２桁である)

$B = +8.4 \times 10^{-4}$　(有効数字２桁である)

$C = A + B = ?$

解答：

クイズ：十進数の演算（有効数字）

$A = -1.0 \times 10^{-2}$　(有効数字２桁である)

$B = +8.4 \times 10^{-4}$　(有効数字２桁である)

$C = A + B = ?$

解答：

$$C = -1.0 \times 10^{-2} + 8.4 \times 10^{-4}$$

$$= -(1.0 \times 10^{-2} - 8.4 \times 10^{-4}) \quad \text{(Sign and operation)}$$

$$= -(1.0 \times 10^{-2} - 0.084 \times 10^{-2}) \quad \text{(Alignment)}$$

$$= -0.916 \times 10^{-2} \quad \text{(Calculation)}$$

$$= -9.2 \times 10^{-3} \quad \text{(Normalization)}$$

浮動小数点数

Question: Convert following variables (in Java or C) into binary format numbers.

```
int     i =  3;
int     j = -2;
float   s = -2.625;
double pi =  3.14159265358979323846264338327950
             28841971693993751058209749445923
             07816406286208998628034825342117
             06798214808651328230664709384460
             95505822317253594081284811174502
             84102701938521105559644622948954
             93038196442881097566593344612847
             56482337867831652712019091456485
             66923460348610454326648213393607;
```

浮動小数点数

Answer:

```
int      i =  3;
Binary format: 00000000000000000000000000000011

int      j = -2;
Binary format: 11111111111111111111111111111110

float    s = -2.625;
Binary format: 11000000001010000000000000000000
(IEEE 754 single-precision format - 32 bits)

double pi =  3.14159265358979323846264338327950
Binary format: 0100000000001001001000011111011
               0101010001000100001011010001100

(IEEE 754 double-precision format - 64 bits)
```

IEEE 754 浮動小数点数のフォーマット

s: Sign (符号部)

e: Exponent (指数部)

f: Fraction (仮数部)

- Normalized: if $0 < e < 255$, then $V = (-1)^s \times 2^{e-127} \times 1.f$
- $+0, -0$: if $e = 0$ and $f = 0$, then $V = (-1)^s \times 0$
- Denormalized: if $e = 0$ and $f \neq 0$, then $V = (-1)^s \times 2^{-126} \times 0.f$
- $+\infty, -\infty$: if $e = 255$ and $f = 0$, then $V = (-1)^s \infty$
- NaN (Not a Number): if $e = 255$ and $f \neq 0$, then $V = \mathrm{NaN}$

単精度浮動小数点数の例

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

- $1\ 01111110\ 10000000000000000000000 = -1.1 \times 2^{126-127} = -0.75$
- $0\ 00000001\ 00000000000000000000000 = +1.0 \times 2^{1-127} = +2^{-126}$
- $0\ 11111110\ 00000000000000000000000 = +1.0 \times 2^{254-127} = +2^{127}$
- $0\ 11111110\ 11111111111111111111111 = +(2 - 2^{-23}) \times 2^{254-127}$

Zero, infinity, and NaN:

- $0\ 00000000\ 00000000000000000000000 = +0$
- $1\ 00000000\ 00000000000000000000000 = -0$
- $0\ 11111111\ 00000000000000000000000 = +\infty$
- $1\ 11111111\ 00000000000000000000000 = -\infty$
- $0\ 11111111\ 10000000000000000000000 = \mathrm{NaN}$
- $1\ 11111111\ 00000100001100000000000 = \mathrm{NaN}$

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

- $0\ 00000000\ 10000000000000000000000 = +0.1 \times 2^{-126} = +2^{-127}$
- $0\ 00000000\ 00000000000000000000001 = +2^{-23} \times 2^{-126} = +2^{-149}$

IEEE 754 倍精度のフォーマット

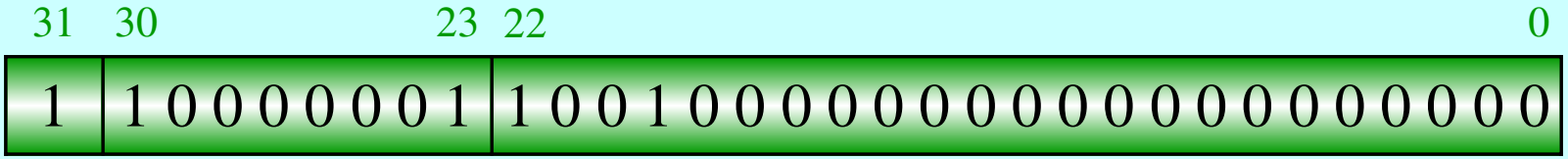
s: Sign (符号部)

e: Exponent (指数部)

f: Fraction (仮数部)

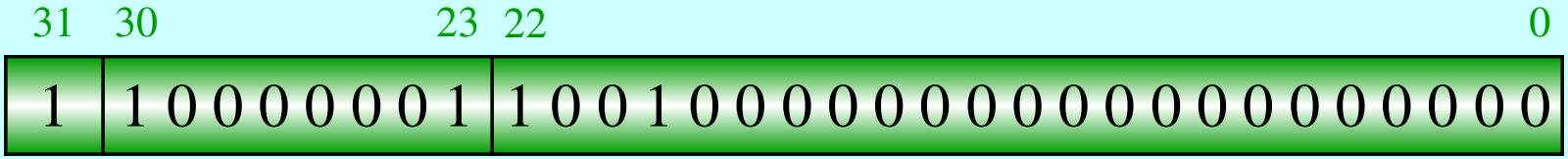
- Normalized: if $0 < e < 2047$, then $V = (-1)^s \times 2^{e-1023} \times 1.f$
- $+0, -0$: if $e = 0$ and $f = 0$, then $V = (-1)^s \times 0$
- Denormalized: if $e = 0$ and $f \neq 0$, then $V = (-1)^s \times 2^{-1022} \times 0.f$
- $+\infty, -\infty$: if $e = 2047$ and $f = 0$, then $V = (-1)^s \infty$
- NaN (Not a Number): if $e = 2047$ and $f \neq 0$, then $V = \mathrm{NaN}$

単精度バイナリ形式から十進への変換



$$
\begin{aligned}
V &= (-1)^1 \times 2^{129-127} \times 1.10010000000000000000000_2 \\
&= -2^2 \times 1.10010000000000000000000_2 \\
&= -110.01000000000000000000_2 \\
&= -(4 + 2 + 0.25)_{10} \\
&= -6.25_{10}
\end{aligned}
$$

十進から単精度バイナリ形式への変換



$$
\begin{aligned}
V &= -6.25_{10} \\
&= -(6_{10} + 0.25_{10}) \\
&= -(110_2 + 0.01_2) \\
&= -110.01_2 \\
&= -1.1001_2 \times 2^2 \\
&= -1.1001_2 \times 2^{129-127} \\
&= (-1)^s \times 2^{e-127} \times 1.f
\end{aligned}
$$

$s = 1, e = 10000001, f = 10010000000000000000$

正規化数以外の演算

n: normalized number

$n + (+\infty) = +\infty$	$(+\infty) + (+\infty) = +\infty$	$(+\infty) + (-\infty) = \mathrm{NaN}$
$n + (-\infty) = -\infty$	$(-\infty) + (-\infty) = -\infty$	$(-\infty) + (+\infty) = \mathrm{NaN}$
$n - (+\infty) = -\infty$	$(+\infty) - (-\infty) = +\infty$	$(+\infty) - (+\infty) = \mathrm{NaN}$
$n - (-\infty) = +\infty$	$(-\infty) - (+\infty) = -\infty$	$(-\infty) - (-\infty) = \mathrm{NaN}$
$n \times (+\infty) = +\infty$	$0 \times (+\infty) = \mathrm{NaN}$	$0/0 = \mathrm{NaN}$
$n \times (-\infty) = -\infty$	$0 \times (-\infty) = \mathrm{NaN}$	$\infty/\infty = \mathrm{NaN}$
$n/(+\infty) = +0$	$n/(+0) = +\infty$	if $n < 0$, $\sqrt{n} = \mathrm{NaN}$
$n/(-\infty) = -0$	$n/(-0) = -\infty$	$n \bmod 0 = \mathrm{NaN}$
		$\infty \bmod n = \mathrm{NaN}$

十進数の演算（有効数字）

$A = -1.0 \times 10^{-2}$　(有効数字２桁である)

$B = +8.4 \times 10^{-4}$　(有効数字２桁である)

$C = A + B = ?$

解答：

$$C = -1.0 \times 10^{-2} + 8.4 \times 10^{-4}$$

$$= -(1.0 \times 10^{-2} - 8.4 \times 10^{-4}) \quad \text{(Sign and operation)}$$

$$= -(1.0 \times 10^{-2} - 0.084 \times 10^{-2}) \quad \text{(Alignment)}$$

$$= -0.916 \times 10^{-2} \quad \text{(Calculation)}$$

$$= -9.2 \times 10^{-3} \quad \text{(Normalization)}$$

単精度浮動小数点数の演算

A:	0	0 1 1 1 1 1 0 1	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
B:	1	0 1 1 1 1 0 0 0	1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1

Ea − Eb = 0 1 1 1 1 1 0 1 − 0 1 1 1 1 0 0 0 = 0 0 0 0 0 1 0 1 = 5

Sc = 0, shift 1.Fb 5-bit right, subtraction, tempE = 0 1 1 1 1 1 0 1

Sticky bit is set whenever there are nonzero bits to the right of the round bit

Guard Round Sticky

	0 1. 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0; 0 0 0
Alignment −	0 0. 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0; 1 0 1
Calculation	0 0. 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1; 0 1 1
Normalization (shift)	0 1. 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0; 1 1 0
Normalization (round)	0 1. 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1;

C:	0	0 1 1 1 1 1 0 0	1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

非正規化数の演算

```
A = 0 00000111 00000000000000000000000 ; 03800000 = 1.0 * 2^{-120}
B = 0 00000000 10000000000000000000000 ; 00400000 = 0.5 * 2^{-126}

Ea - Eb = 00000111 - 00000000 = 7,
but B is denormalized (-126, 0.f), A is normalized (-127, 1.f),
then shift B to the right 7 - 1 = 6 bit.
```

```
    01.00000000000000000000000 000
+   00.00000010000000000000000 000
-------------------------------------
    01.00000010000000000000000 000
```

```
C = 0 00000111 00000010000000000000000 ; 03810000
```

丸めの手法

- Floating-point numbers are normally approximations for a number they cannot really represent.
- The best we can do is get the floating-point representation close to the actual.
- Four rounding modes defined by IEEE 754 are
  1. Round to nearest (to even if g r s = 1 0 0)
  2. Round toward minus infinity ($-\infty$)
  3. Round toward plus infinity ($+\infty$)
  4. Round toward zero (also called truncate)

丸めの例

Fraction	g	r	s	nearest	truncate	$+\infty$	$-\infty$
+...1001	0	0	0	+1001	+1001	+1001	+1001
+...1001	1	0	0	+1010	+1001	+1010	+1001
+...1000	1	0	0	+1000	+1000	+1001	+1000
+...1001	0	1	1	+1001	+1001	+1010	+1001
+...1001	1	1	0	+1010	+1001	+1010	+1001
−...1001	0	0	0	−1001	−1001	−1001	−1001
−...1001	1	0	0	−1010	−1001	−1001	−1010
−...1000	1	0	0	−1000	−1000	−1000	−1001
−...1001	0	1	1	−1001	−1001	−1001	−1010
−...1001	1	1	0	−1010	−1001	−1001	−1010

オーバーフロー

1. Round to nearest carries all overflows to ∞ with the sign of the intermediate result.
2. Round toward 0 carries all overflows to the format's largest finite number with the sign of the intermediate result.
3. Round toward $-\infty$ carries positive overflows to the format's largest finite number, and carries negative overflows to $-\infty$.
4. Round toward $+\infty$ carries negative overflows to the format's most negative finite number, and carries positive overflows to $+\infty$.

オーバーフロー丸めの例

```
A = 0 11111110 11111111111111111111111       ; 7f7fffff
B = 0 11111011 11111111111111111111111       ; 7dffffff
```

```
      01.11111111111111111111111 000
+     00.00111111111111111111111 111
-------------------------------------
      10.00111111111111111111110 111
→     01.00011111111111111111111 011
```

```
Nearest, Up: (Infinity)
C = 0 11111111 00000000000000000000000       ; 7f800000
Down, Chop: (Largest)
C = 0 11111110 11111111111111111111111       ; 7f7fffff
```

オーバーフロー丸めの例

```
A = 0 11111110 11111111111111111111111        ; 7f7fffff
B = 0 00000000 00000000000000000000001        ; 00000001
```

```
      01.11111111111111111111111 000
+     00.00000000000000000000000 001
-------------------------------------
      01.11111111111111111111111 001
←     10.00000000000000000000000 up
```

```
Up: (Infinity)
C = 0 11111111 00000000000000000000000        ; 7f800000
Near, Down, Chop: (Normal result)
C = 0 11111110 11111111111111111111111        ; 7f7fffff
```

C 言語における丸め方向指定方法

```
// Rounding test for (x64 + linux + gcc)

#include <stdio.h>

unsigned int _RoundNear = 0x00001f80; // Round Control (RC) = 00 nearest
unsigned int _RoundDown = 0x00003f80; // Round Control (RC) = 01 down
unsigned int _RoundUp   = 0x00005f80; // Round Control (RC) = 10 up
unsigned int _RoundChop = 0x00007f80; // Round Control (RC) = 11 chop

#define Near() asm volatile("ldmxcsr %0" : : "m"(_RoundNear)) // for 64 bits
#define Down() asm volatile("ldmxcsr %0" : : "m"(_RoundDown)) // for 64 bits
#define Up()   asm volatile("ldmxcsr %0" : : "m"(_RoundUp))   // for 64 bits
#define Chop() asm volatile("ldmxcsr %0" : : "m"(_RoundChop)) // for 64 bits
```

C 言語における丸め方向指定方法

```
int main(void){
  union {
    int intword;
    float floatword;
  } u, v, s, t;

  while(1){
    fprintf(stderr,"input 1st fp number (32 bits) in hex format (ex. 3c600011): ");
    fscanf(stdin,"%x",&u.intword);
    fprintf(stderr,"input 2nd fp number (32 bits) in hex format (ex. be820000): ");
    fscanf(stdin,"%x",&v.intword);

    fprintf(stderr,"the two 32-bit fp numbers are: "
                   "%08x and %08x\n\n",u.intword,v.intword);
```

C 言語における丸め方向指定方法

```
Near(); // Near
s.floatword = u.floatword + v.floatword;
t.floatword = u.floatword - v.floatword;
fprintf(stderr,"the sum of two fp numbers is (near): "
               "%08x\t%08x\n",s.intword,t.intword);

Down(); // Down
s.floatword = u.floatword + v.floatword;
t.floatword = u.floatword - v.floatword;
fprintf(stderr,"the sum of two fp numbers is (down): "
               "%08x\t%08x\n",s.intword,t.intword);
```

C 言語における丸め方向指定方法

```
    Up(); // Up
    s.floatword = u.floatword + v.floatword;
    t.floatword = u.floatword - v.floatword;
    fprintf(stderr,"the sum of two fp numbers is ( up ): "
                   "%08x\t%08x\n",s.intword,t.intword);

    Chop(); // Chop
    s.floatword = u.floatword + v.floatword;
    t.floatword = u.floatword - v.floatword;
    fprintf(stderr,"the sum of two fp numbers is (chop): "
                   "%08x\t%08x\n\n",s.intword,t.intword);
  }
  return 0;
}
```

Download **rounding64float.c**

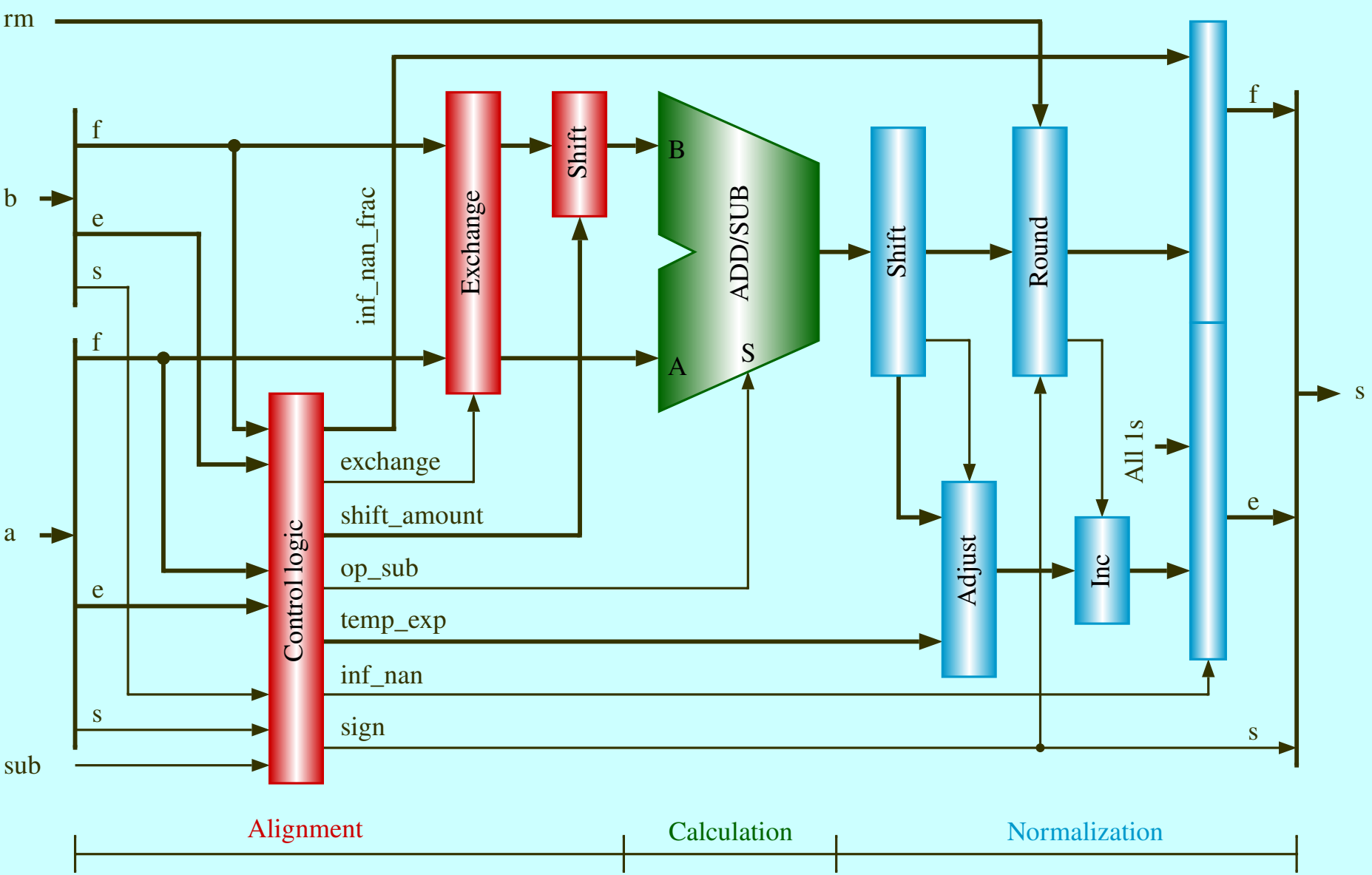
C 言語における丸め方向指定方法

```
[yamin@localhost cod]$ gcc rounding64float.c -o rounding64float
[yamin@localhost cod]$ ./rounding64float
input 1st fp number (32 bits) in hex format (ex. 3c600011): 3c600011
input 2nd fp number (32 bits) in hex format (ex. be820000): be820000
the two 32-bit fp numbers are: 3c600011 and be820000
the sum of two fp numbers is (near): be75ffff    3e890001
the sum of two fp numbers is (down): be75ffff    3e890000
the sum of two fp numbers is ( up ): be75fffe    3e890001
the sum of two fp numbers is (chop): be75fffe    3e890000
input 1st fp number (32 bits) in hex format (ex. 3c600011): 7f7fffff
input 2nd fp number (32 bits) in hex format (ex. be820000): 7dffffff
the two 32-bit fp numbers are: 7f7fffff and 7dffffff
the sum of two fp numbers is (near): 7f800000    7f5fffff
the sum of two fp numbers is (down): 7f7fffff    7f5fffff
the sum of two fp numbers is ( up ): 7f800000    7f600000
the sum of two fp numbers is (chop): 7f7fffff    7f5fffff
```

For x64 double version, see: **rounding64double.c**

For ARM floating version, see: **rounding_float.c**　double version: **rounding_double.c**

単精度浮動小数点演算回路



fadder.v

```
‘timescale 1ns/1ns // By Yamin Li, yamin@ieee.org
module fadder (a,b,sub,rm,s);                                   // fadder
    input  [31:0] a,b;                                          // fp a and b
    input   [1:0] rm;                                           // round mode
    input         sub;                                          // 1: sub; 0: add
    output [31:0] s;                                            // fp output
    wire          exchange = ({1’b0,b[30:0]} > {1’b0,a[30:0]});
    wire   [31:0] fp_large = exchange? b : a;
    wire   [31:0] fp_small = exchange? a : b;
    wire          fp_large_hidden_bit = |fp_large[30:23];
    wire          fp_small_hidden_bit = |fp_small[30:23];
    wire   [23:0] large_frac24 = {fp_large_hidden_bit,fp_large[22:0]};
    wire   [23:0] small_frac24 = {fp_small_hidden_bit,fp_small[22:0]};
    wire    [7:0] temp_exp = fp_large[30:23];
    wire          sign = exchange? sub ^ b[31] : a[31];
    wire          op_sub = sub ^ fp_large[31] ^ fp_small[31];
    wire          fp_large_expo_is_ff =  &fp_large[30:23]; // exp == 0xff
    wire          fp_small_expo_is_ff =  &fp_small[30:23];
    wire          fp_large_frac_is_00 = ~|fp_large[22:0];  // frac == 0x0
    wire          fp_small_frac_is_00 = ~|fp_small[22:0];
    wire          fp_large_is_inf=fp_large_expo_is_ff & fp_large_frac_is_00;
    wire          fp_small_is_inf=fp_small_expo_is_ff & fp_small_frac_is_00;
```

fadder.v

```
    wire          fp_large_is_nan=fp_large_expo_is_ff &~fp_large_frac_is_00;
    wire          fp_small_is_nan=fp_small_expo_is_ff &~fp_small_frac_is_00;
    wire          s_is_inf = fp_large_is_inf | fp_small_is_inf;
    wire          s_is_nan = fp_large_is_nan | fp_small_is_nan |
                             ((sub ^ fp_small[31] ^ fp_large[31]) &
                             fp_large_is_inf & fp_small_is_inf);
    wire   [22:0] nan_frac = ({1'b0,a[22:0]} > {1'b0,b[22:0]}) ?
                              {1'b1,a[21:0]} : {1'b1,b[21:0]};
    wire   [22:0] inf_nan_frac = s_is_nan? nan_frac : 23'h0;
    wire    [7:0] exp_diff = fp_large[30:23] - fp_small[30:23];
    wire          small_den_only = (fp_large[30:23] != 0) &
                                   (fp_small[30:23] == 0);
    wire    [7:0] shift_amount = small_den_only? exp_diff - 8'h1 : exp_diff;
    wire   [49:0] small_frac50 = (shift_amount >= 26)?
                  {26'h0,small_frac24} :
                  {small_frac24,26'h0} >> shift_amount;
    wire   [26:0] small_frac27 = {small_frac50[49:24],|small_frac50[23:0]};
    wire   [27:0] aligned_large_frac = {1'b0,large_frac24,3'b000};
    wire   [27:0] aligned_small_frac = {1'b0,small_frac27};
    wire   [27:0] cal_frac = op_sub?
                  aligned_large_frac - aligned_small_frac :
                  aligned_large_frac + aligned_small_frac;
```

fadder.v

```
    wire   [26:0] f4,f3,f2,f1,f0;
    wire    [4:0] zeros;
    assign        zeros[4] = ~|cal_frac[26:11];            // 16-bit 0
    assign        f4 = zeros[4]? {cal_frac[10:0],16'b0} : cal_frac[26:0];
    assign        zeros[3] = ~|f4[26:19];                  //  8-bit 0
    assign        f3 = zeros[3]? {f4[18:0], 8'b0} : f4;
    assign        zeros[2] = ~|f3[26:23];                  //  4-bit 0
    assign        f2 = zeros[2]? {f3[22:0], 4'b0} : f3;
    assign        zeros[1] = ~|f2[26:25];                  //  2-bit 0
    assign        f1 = zeros[1]? {f2[24:0], 2'b0} : f2;
    assign        zeros[0] = ~f1[26];                      //  1-bit 0
    assign        f0 = zeros[0]? {f1[25:0], 1'b0} : f1;

    reg     [7:0] exp0;
    reg    [26:0] frac0;

    always @ * begin
        if (cal_frac[27]) begin           // 1x.xxxxxxxxxxxxxxxxxxxxxxx xxx
            frac0 = cal_frac[27:1];       //  1.xxxxxxxxxxxxxxxxxxxxxxx xxx
            exp0 = temp_exp + 8'h1;
        end else begin
            if ((temp_exp > zeros) && (f0[26])) begin // a normalized number
```

fadder.v

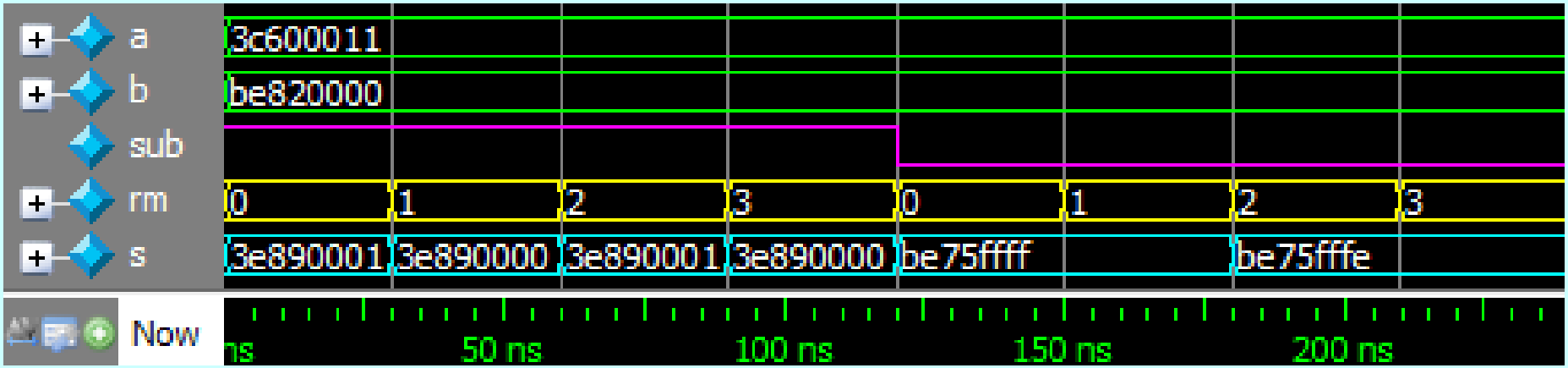
```
            exp0 = temp_exp - zeros;
            frac0 = f0;                 //  1.xxxxxxxxxxxxxxxxxxxxxxx xxx
        end else begin                  // is a denormalized number or 0
            exp0 = 0;
            if (temp_exp != 0)          // (e - 127) = ((e - 1) - 126)
              frac0 = cal_frac[26:0] << (temp_exp - 8'h1);
            else frac0 = cal_frac[26:0];
        end
    end
end

wire frac_plus_1 =                      // for rounding
     ~rm[1] & ~rm[0] &  frac0[2] & (frac0[1] |  frac0[0]) |
     ~rm[1] & ~rm[0] &  frac0[2] & ~frac0[1] & ~frac0[0]  &  frac0[3] |
     ~rm[1] &  rm[0] & (frac0[2] |  frac0[1] |  frac0[0]) &  sign |
      rm[1] & ~rm[0] & (frac0[2] |  frac0[1] |  frac0[0]) & ~sign;
wire   [24:0] frac_round = {1'b0,frac0[26:3]} + frac_plus_1;
wire    [7:0] exponent = frac_round[24]? exp0 + 8'h1 : exp0;
wire          overflow = &exp0 | &exponent;

assign s = final_result(overflow,rm,sign,s_is_nan,s_is_inf,exponent,
                        frac_round[22:0],inf_nan_frac);
```

fadder.v

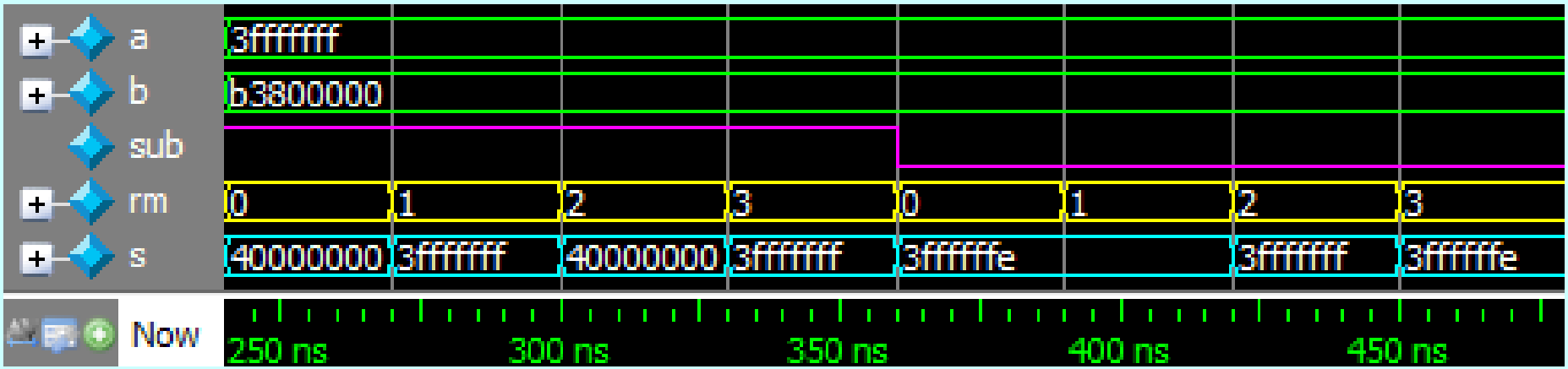
```
    function  [31:0] final_result;
        input        overflow;
        input  [1:0] rm;
        input        sign;
        input        is_nan;
        input        is_inf;
        input  [7:0] exponent;
        input [22:0] fraction, inf_nan_frac;
        casex ({overflow,rm,sign,s_is_nan,s_is_inf})
            6'b1_00_x_0_x : final_result = {sign,8'hff,23'h000000};   // inf
            6'b1_01_0_0_x : final_result = {sign,8'hfe,23'h7fffff};   // max
            6'b1_01_1_0_x : final_result = {sign,8'hff,23'h000000};   // inf
            6'b1_10_0_0_x : final_result = {sign,8'hff,23'h000000};   // inf
            6'b1_10_1_0_x : final_result = {sign,8'hfe,23'h7fffff};   // max
            6'b1_11_x_0_x : final_result = {sign,8'hfe,23'h7fffff};   // max
            6'b0_xx_x_0_0 : final_result = {sign,exponent,fraction};  // nor
            6'bx_xx_x_1_x : final_result = {1'b1,8'hff,inf_nan_frac}; // nan
            6'bx_xx_x_0_1 : final_result = {sign,8'hff,inf_nan_frac}; // inf
            default       : final_result = {sign,8'h00,23'h000000};   // 0
        endcase
    endfunction
endmodule
```

fadder.v fadder_tb.v

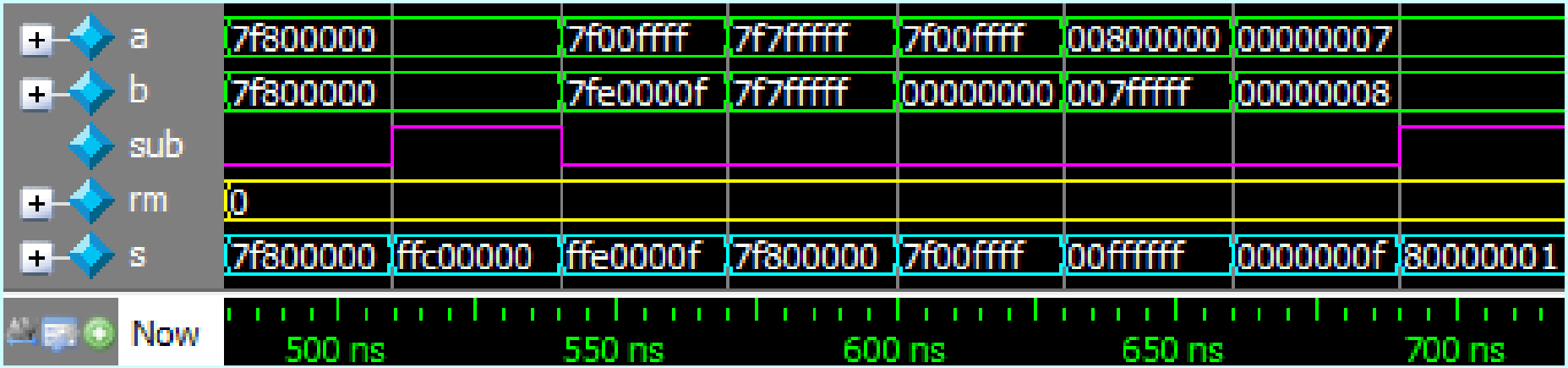
浮動小数点加減算器波形



Increasing exponent caused by rounding

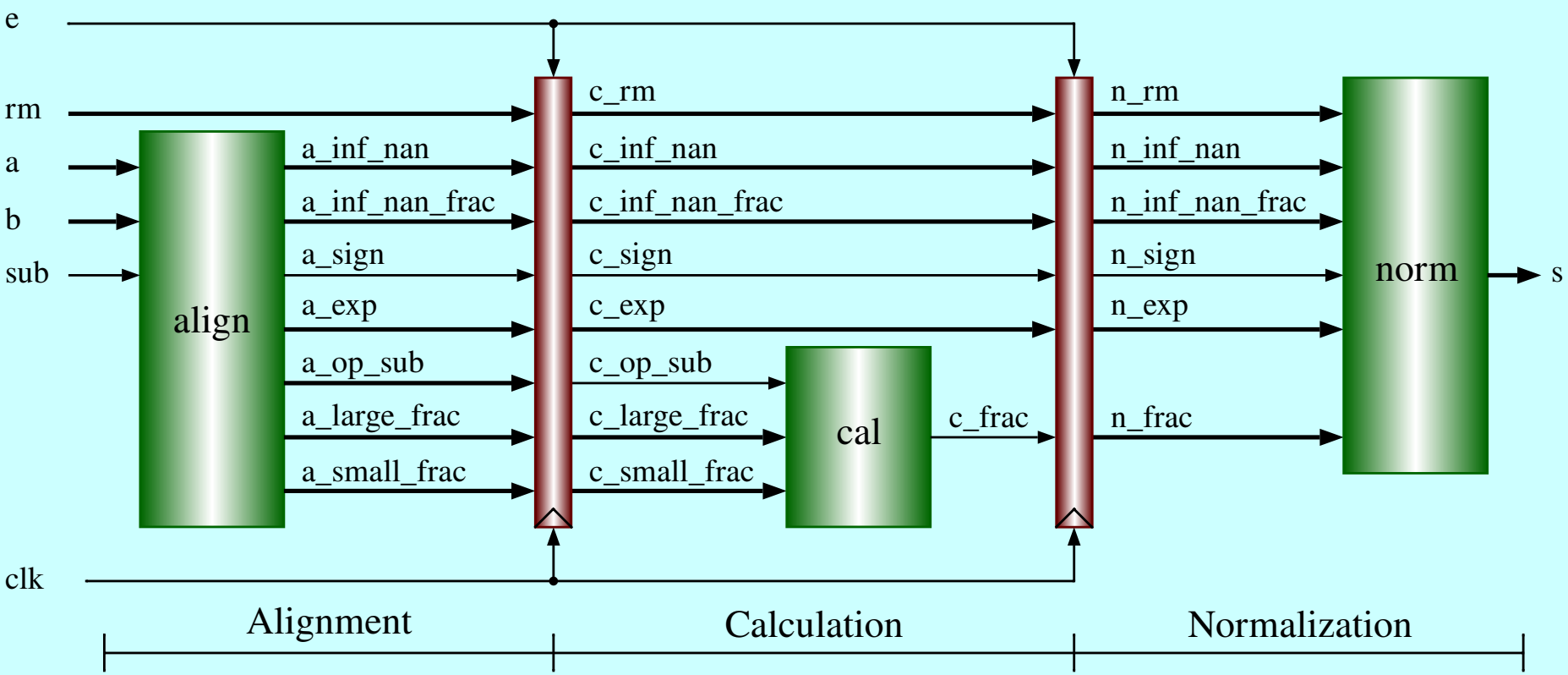


浮動小数点加減算器波形



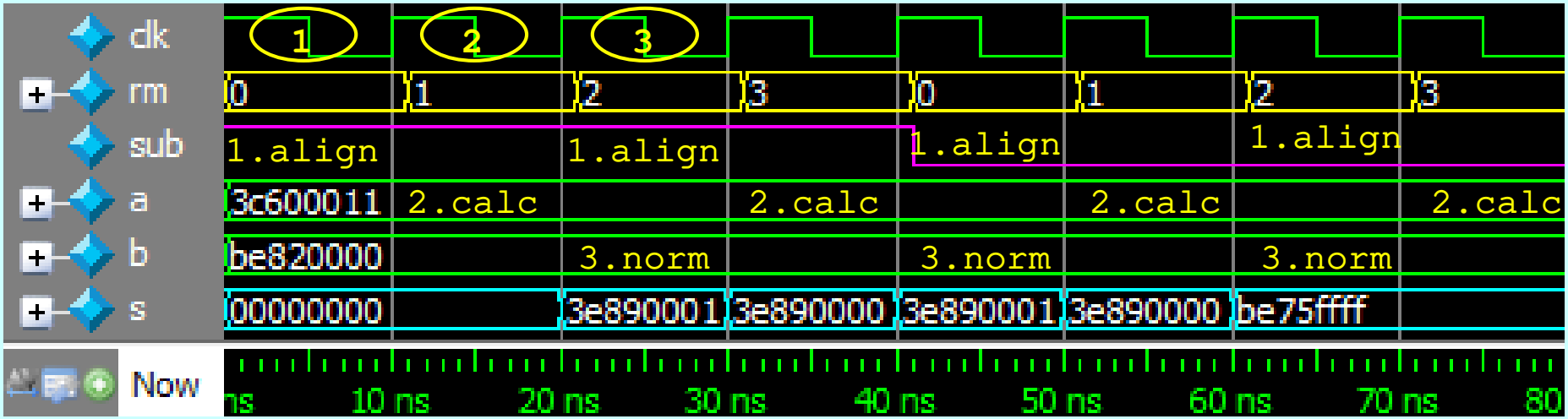
1) $(+\infty) + (+\infty) = +\infty$;
2) $(+\infty) - (+\infty) = \mathrm{NaN}$;
3) a normalized number + NaN = NaN;
4) the sum of the two largest numbers is $+\infty$ (round to nearest);
5) a normalized number + 0;
6) the smallest normalized number + the largest denormalized number;
7) a denormalized number + a denormalized number; and
8) a denormalized number − a denormalized number.

パイプライン浮動小数点加減算器



ステージとステージの間にはレジスタを挿入する

パイプライン浮動小数点加減算器波形



The first result is available in the third clock cycle.

CPU (IU + FPU) の設計

IU: Integer Unit
FPU: Floating-Point Unit

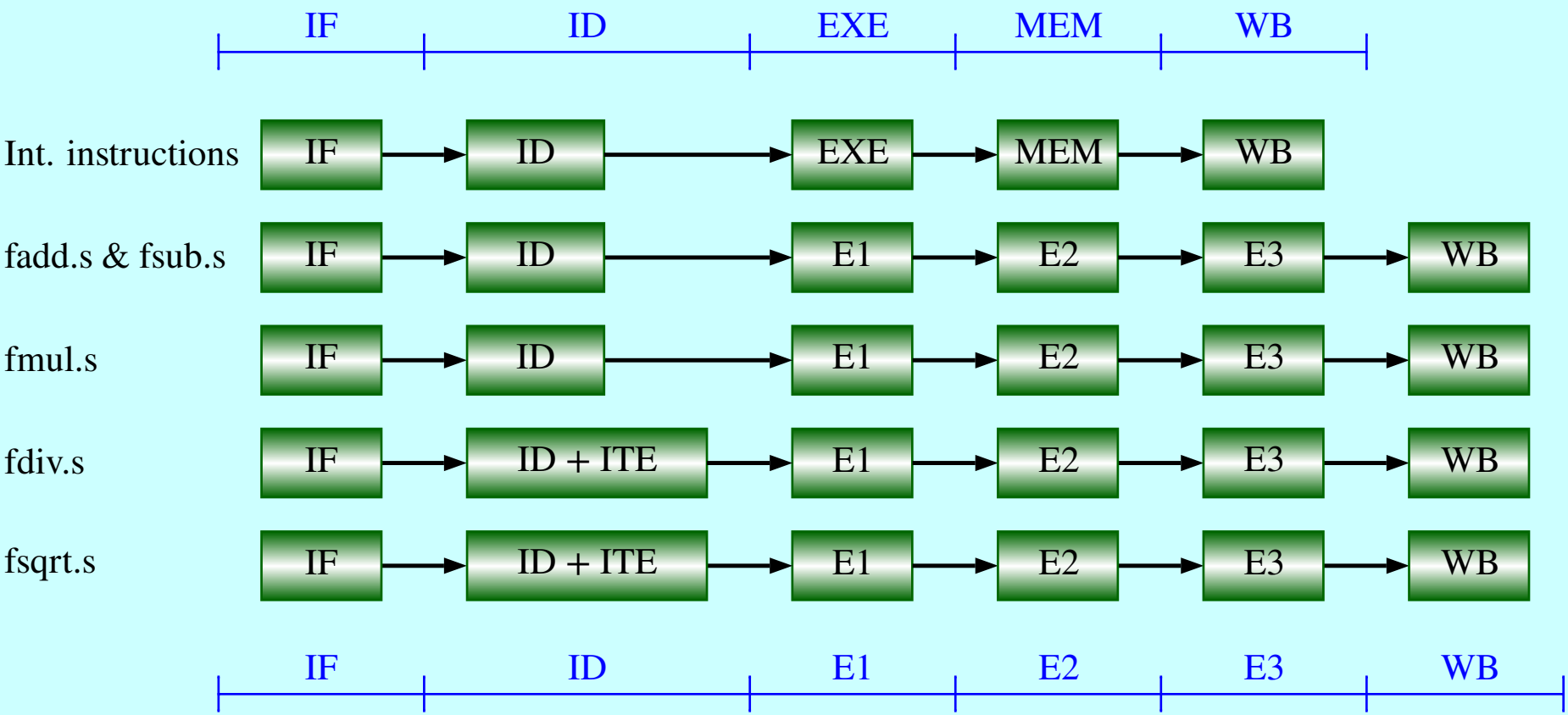
RV32F (Float) Instructions

imm[11:0]		rs1	010	rd	0000111	flw
imm[11:5]	rs2	rs1	010	imm[4:0]	0100111	fsw
0000000	rs2	rs1	rm	rd	1010011	fadd.s
0000100	rs2	rs1	rm	rd	1010011	fsub.s
0001000	rs2	rs1	rm	rd	1010011	fmul.s
0001100	rs2	rs1	rm	rd	1010011	fdiv.s
0101100	00000	rs1	rm	rd	1010011	fsqrt.s

Examples

```
flw     fa4, 0(a5)      # floating point load from memory
flw     fa5, 4(a5)      # floating point load from memory
fadd.s  fa6, fa4, fa5   # floating point addition
fsub.s  fa7, fa4, fa5   # floating point subtraction
fmul.s  fa5, fa4, fa5   # floating point multiplication
fdiv.s  fa5, fa6, fa5   # floating point division
fsqrt.s fa6, fa5        # floating point square root
fsw     fa6, 8(a5)      # floating point store to memory
```

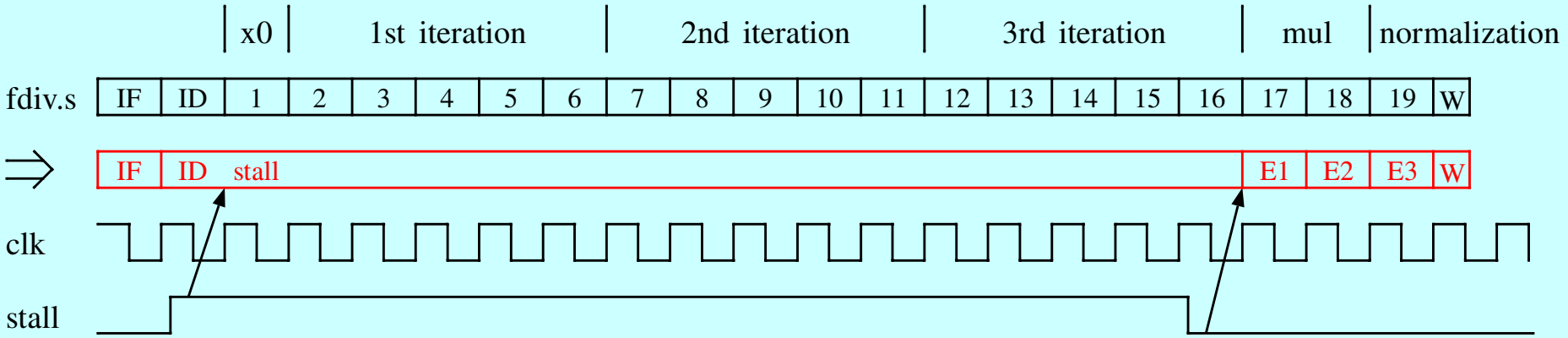
RISC-V IU and FPU Pipeline Models



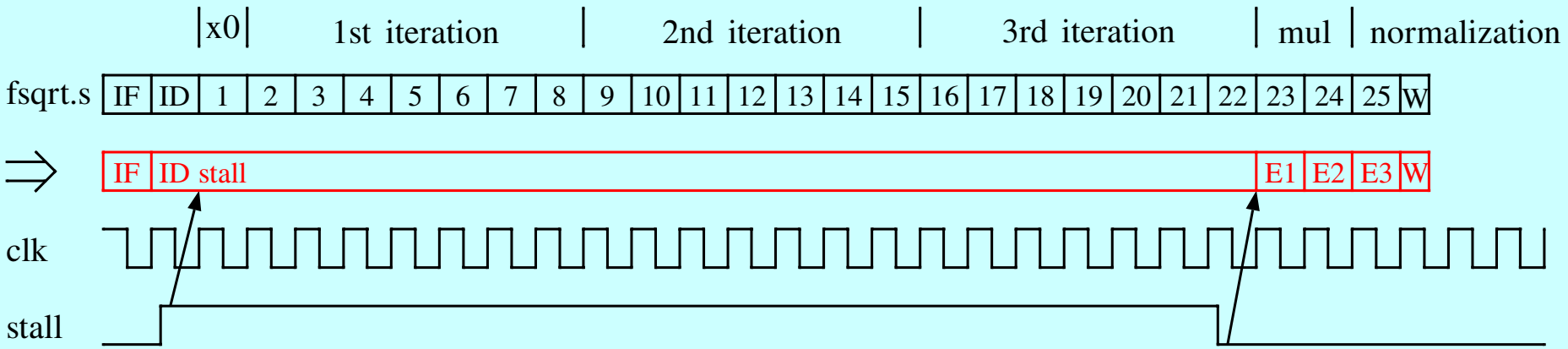
ITE: Newton-Raphson Iterations for division and square root calculations

FDIV.S and FSQRT.S Pipeline Stages

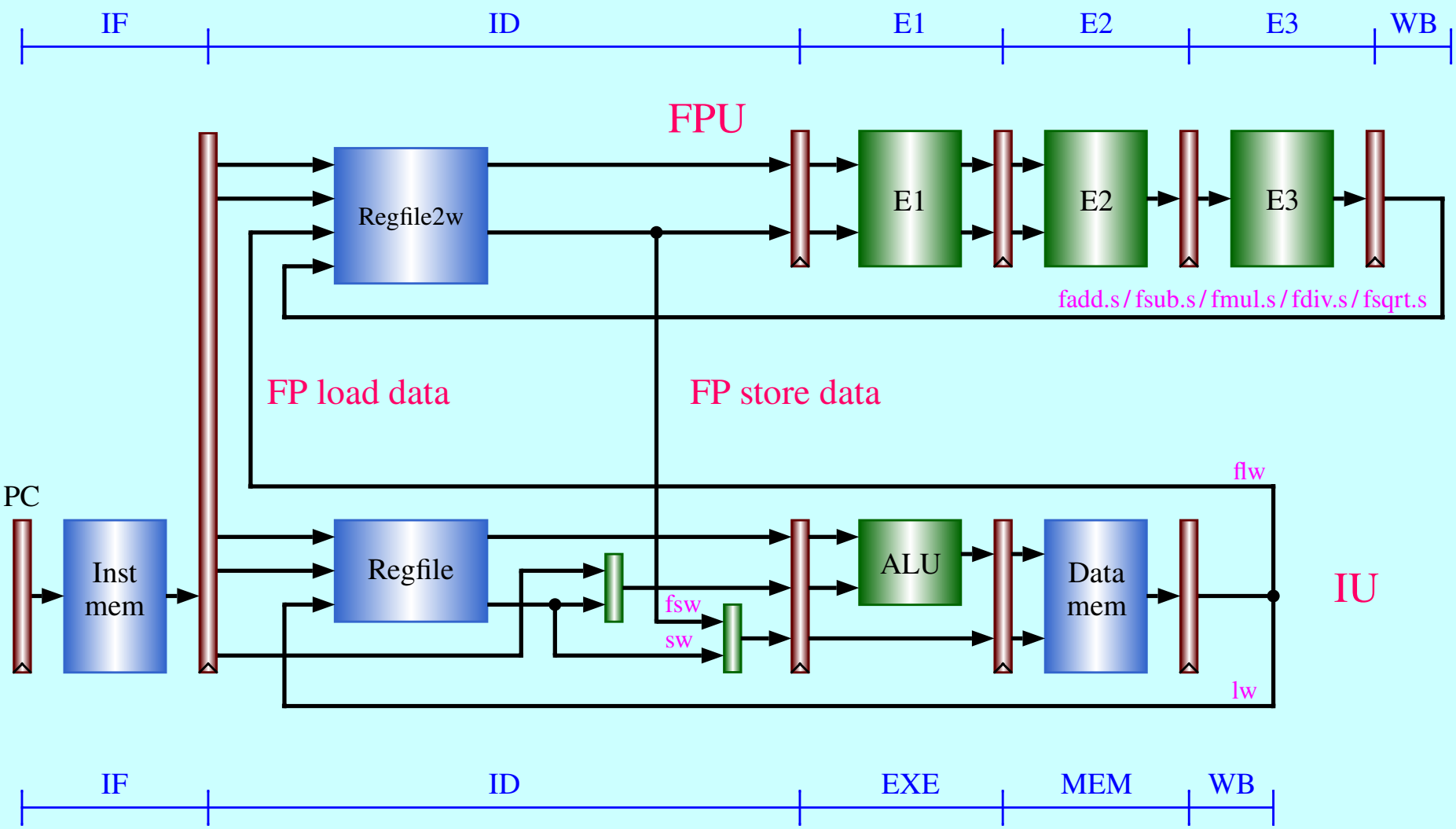
fdiv.s pipeline stages



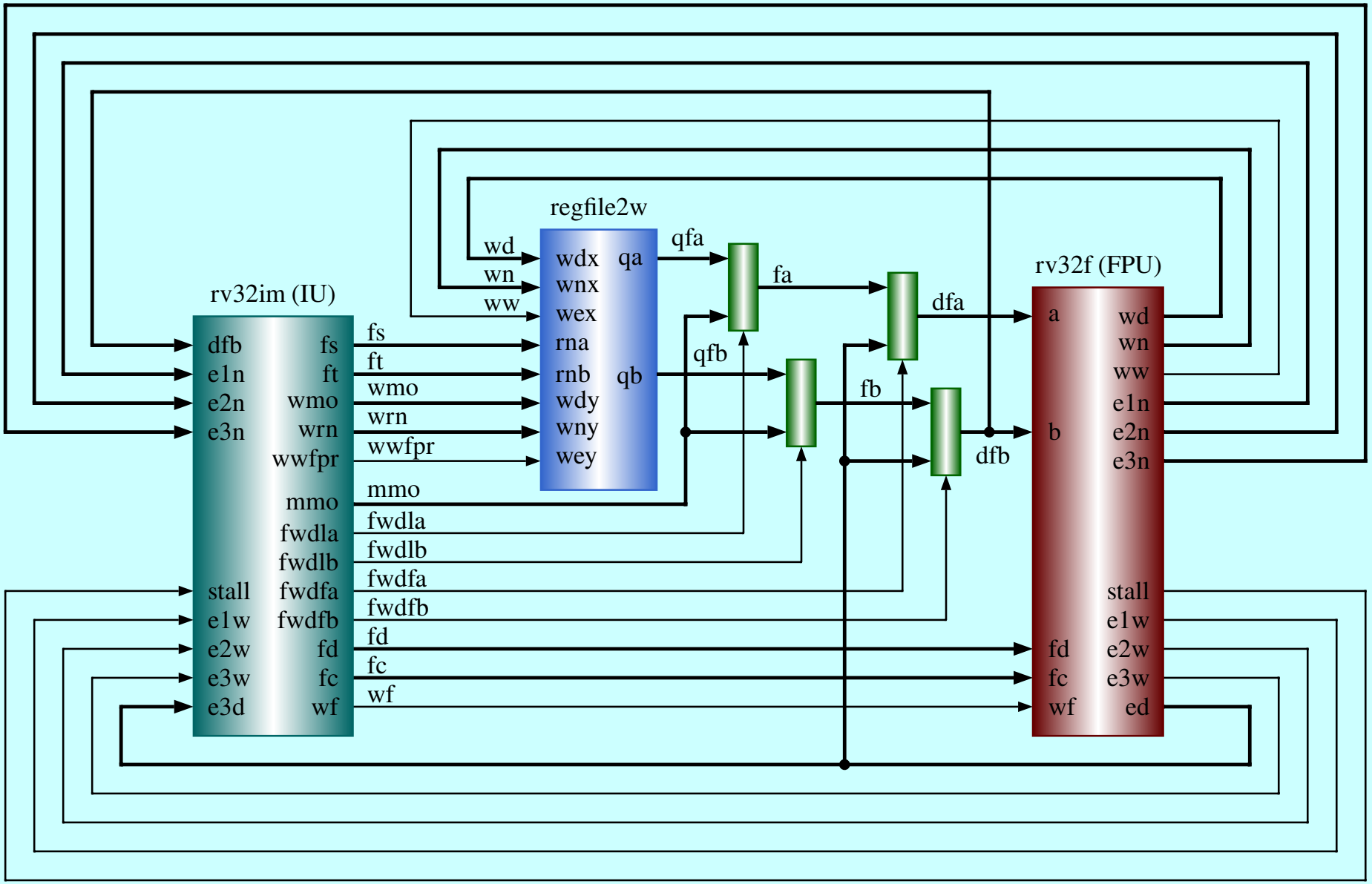
fsqrt.s pipeline stages



RISC-V CPU/FPU Baseline



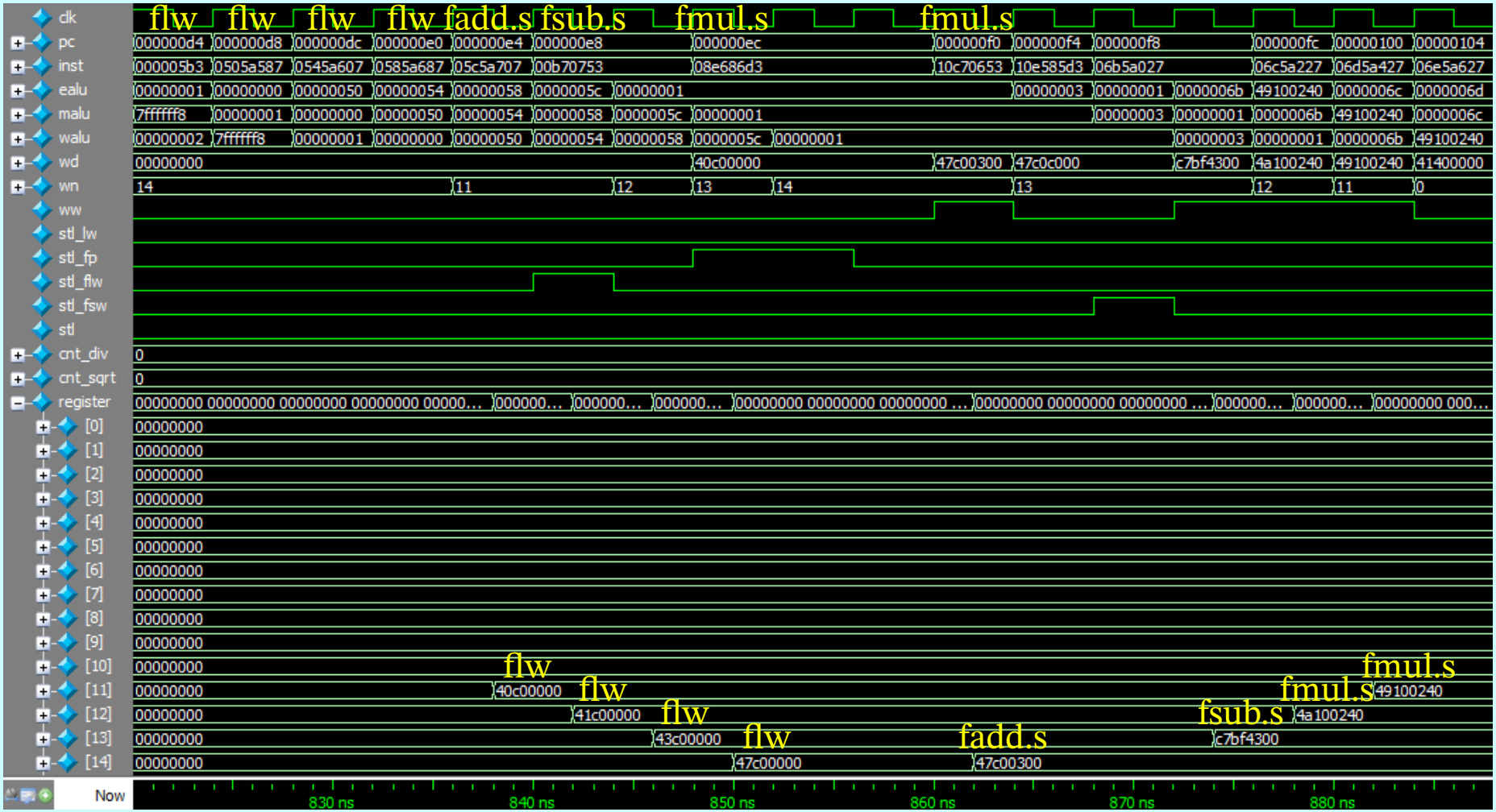
RISC-V CPU/FPU Block Diagram



RISC-V RV32F Test Program

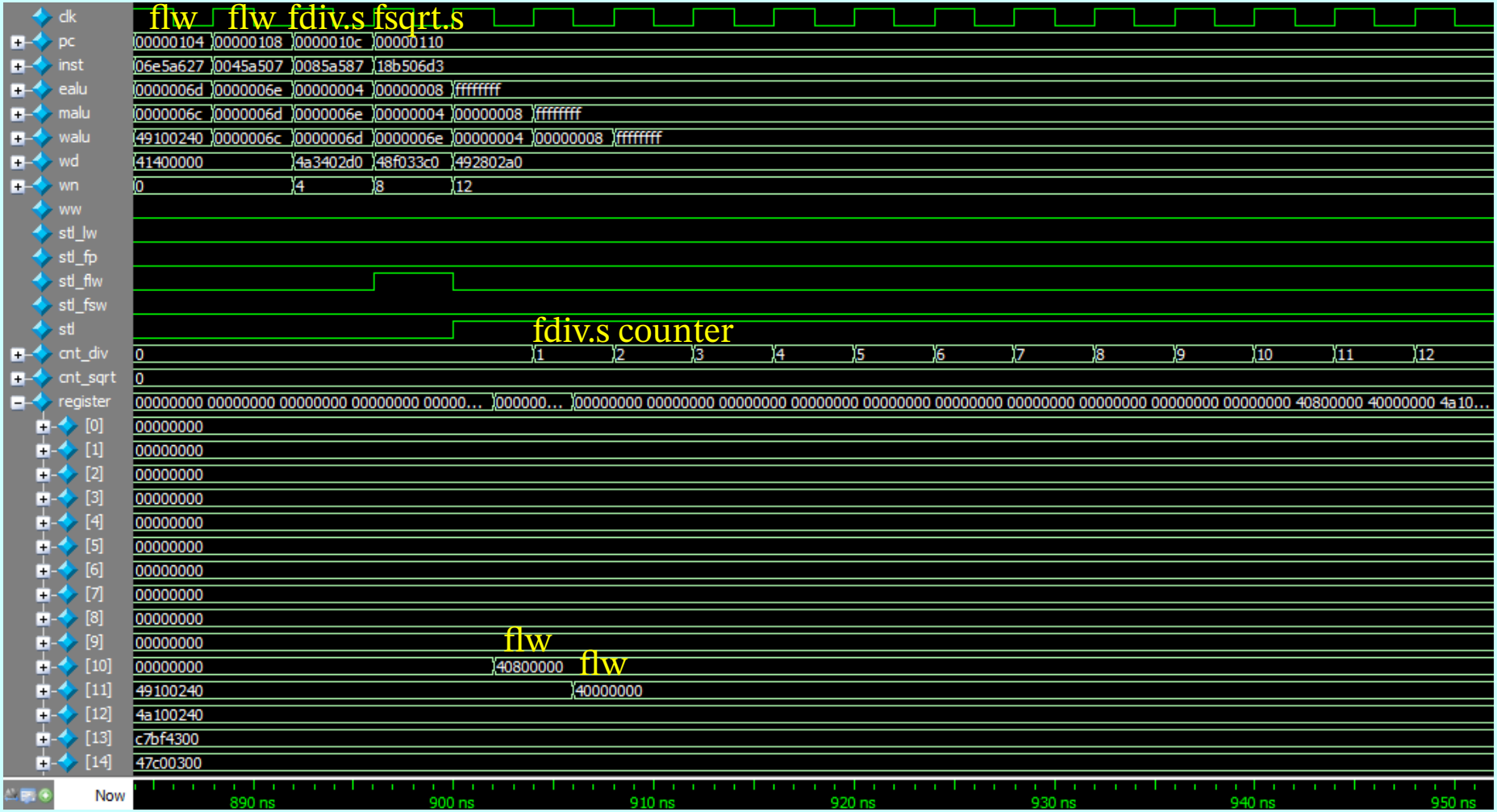
```
fpu: add     a1,   x0,  x0     # (d0)   address 0
     flw     fa1,  80(a1)      # (d4)   0x40c00000 <--
     flw     fa2,  84(a1)      # (d8)   0x41c00000 <--
     flw     fa3,  88(a1)      # (dc)   0x43c00000 <--
     flw     fa4,  92(a1)      # (e0)   0x47c00000 <--
     fadd.s  fa4,  fa4, fa1    # (e4) = 0x47c00300 <--
     fsub.s  fa3,  fa3, fa4    # (e8) = 0xc7bf4300 <--
     fmul.s  fa2,  fa4, fa2    # (ec) = 0x4a100240 <--
     fmul.s  fa1,  fa1, fa4    # (f0) = 0x49100240 <--
     fsw     fa1,  96(a1)      # (f4)
     fsw     fa2, 100(a1)      # (f8)
     fsw     fa3, 104(a1)      # (fc)
     fsw     fa4, 108(a1)      # (100)
     flw     fa0,   4(a1)      # (104)   4.000000 (0x40800000) <--
     flw     fa1,   8(a1)      # (108)   2.000000 (0x40000000) <--
     fdiv.s  fa3,  fa0, fa1    # (10c)   2.000000 (0x40000000) <--
     fsqrt.s fa3,  fa3         # (110) = 1.414214 (0x3fb504f3) <--
     fsqrt.s fa3,  fa3         # (114) = 1.189207 (0x3f9837f0) <--
```

Simulation Waveform



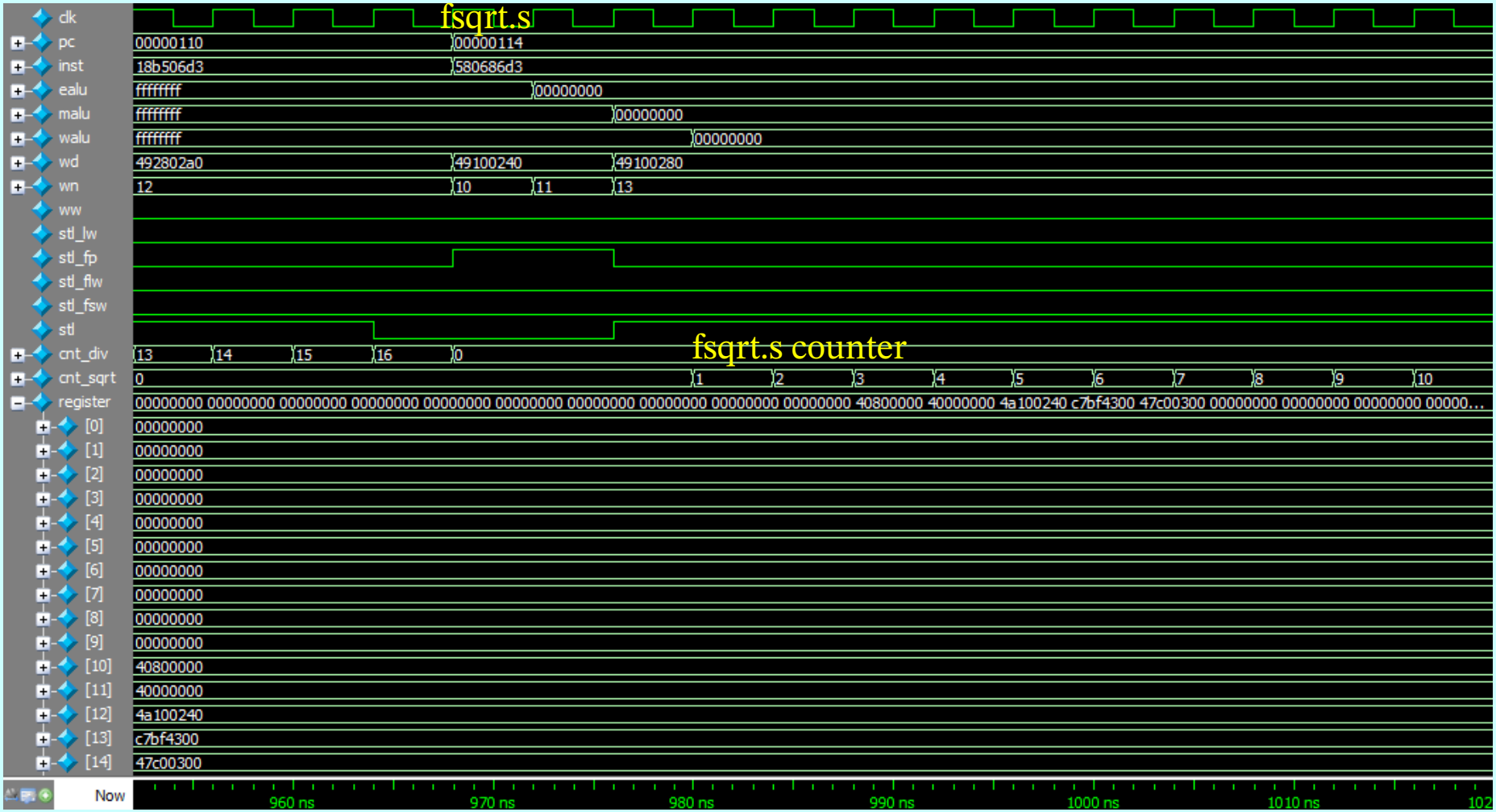
Results of fadd.s, fsub.s, fmul.s, and fmul.s.

Simulation Waveform



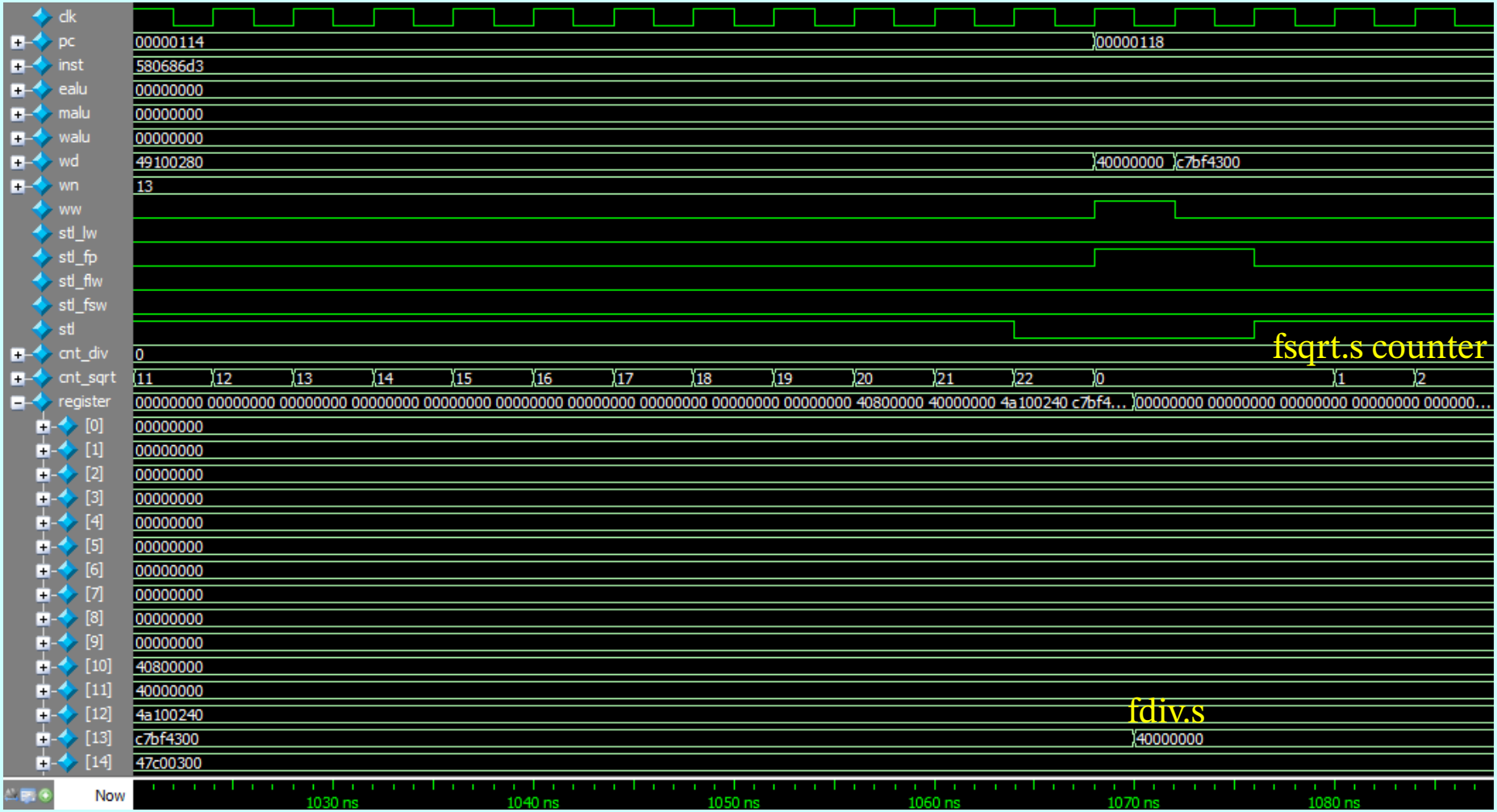
Newton iteration of fdiv.s

Simulation Waveform



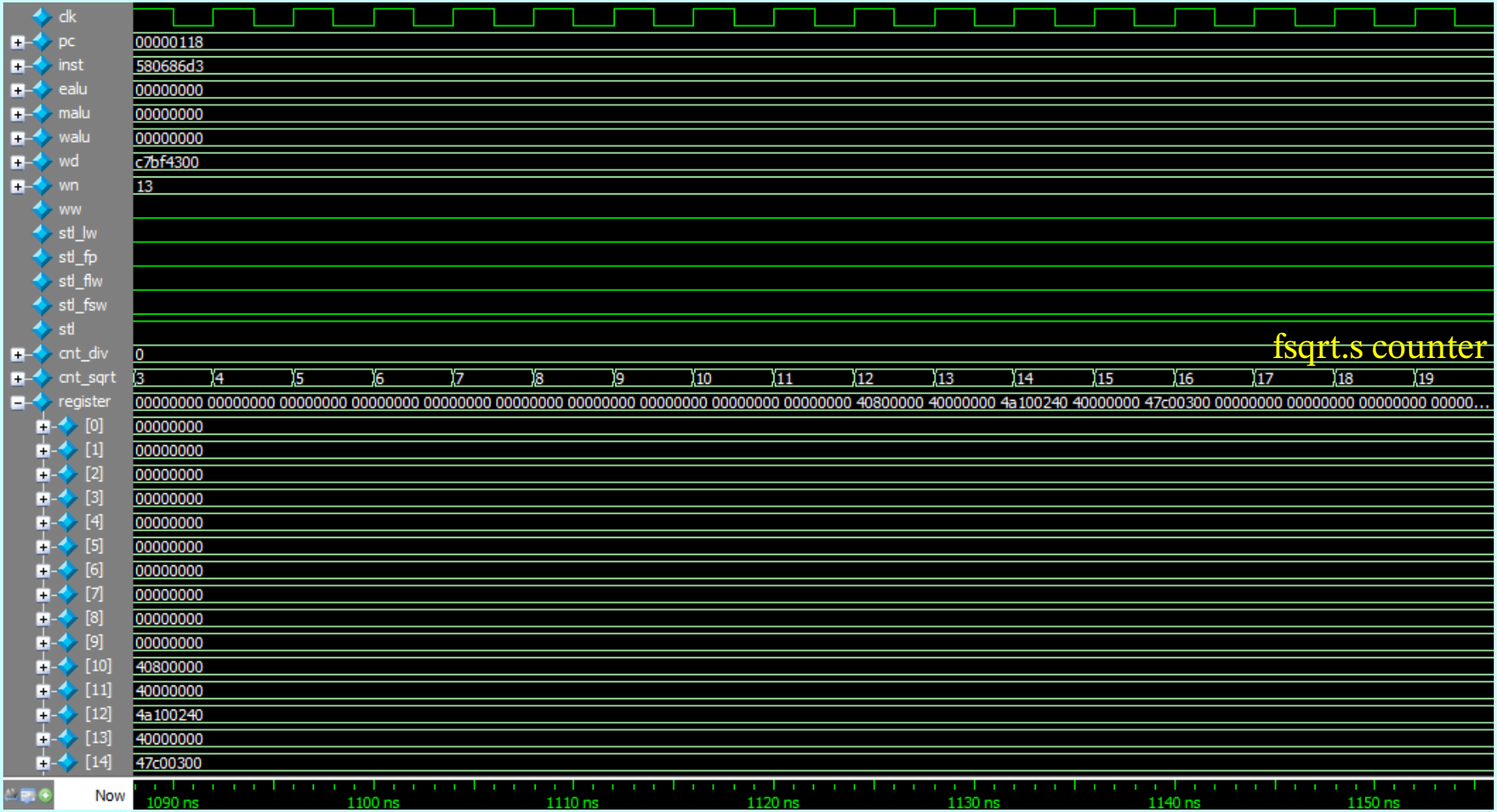
Newton iterations of fdiv.s and fsqrt.s

Simulation Waveform



Results of fdiv.s

Simulation Waveform



Newton iteration of fsqrt.s

Simulation Waveform

Results of fsqrt.s, and fsqrt.s.

課題 X (100 点 + 100 点)

1. Convert decimal number −17.125 to IEEE single-precision floating-point number.
2. Convert IEEE single-precision floating-point number 0x3d600000 to decimal number.
3. オプション (+50 点): Write a program in C/C++, Python, or Java to convert a decimal real number (in String format) to IEEE single-precision floating-point number, and show the result in binary or hexadecimal format for pi = 3.1415926.

文字列	'3'	'.'	'1'	'4'	'1'	'5'	'9'	'2'	'6'
ASCII	0x33	0x2e	0x31	0x34	0x31	0x35	0x39	0x32	0x36

課題 X (100 点 + 100 点)

4. オプション (+50 点):
   Design and simulate an RISC-V CPU RV32IMF that contains an IU and an FPU.