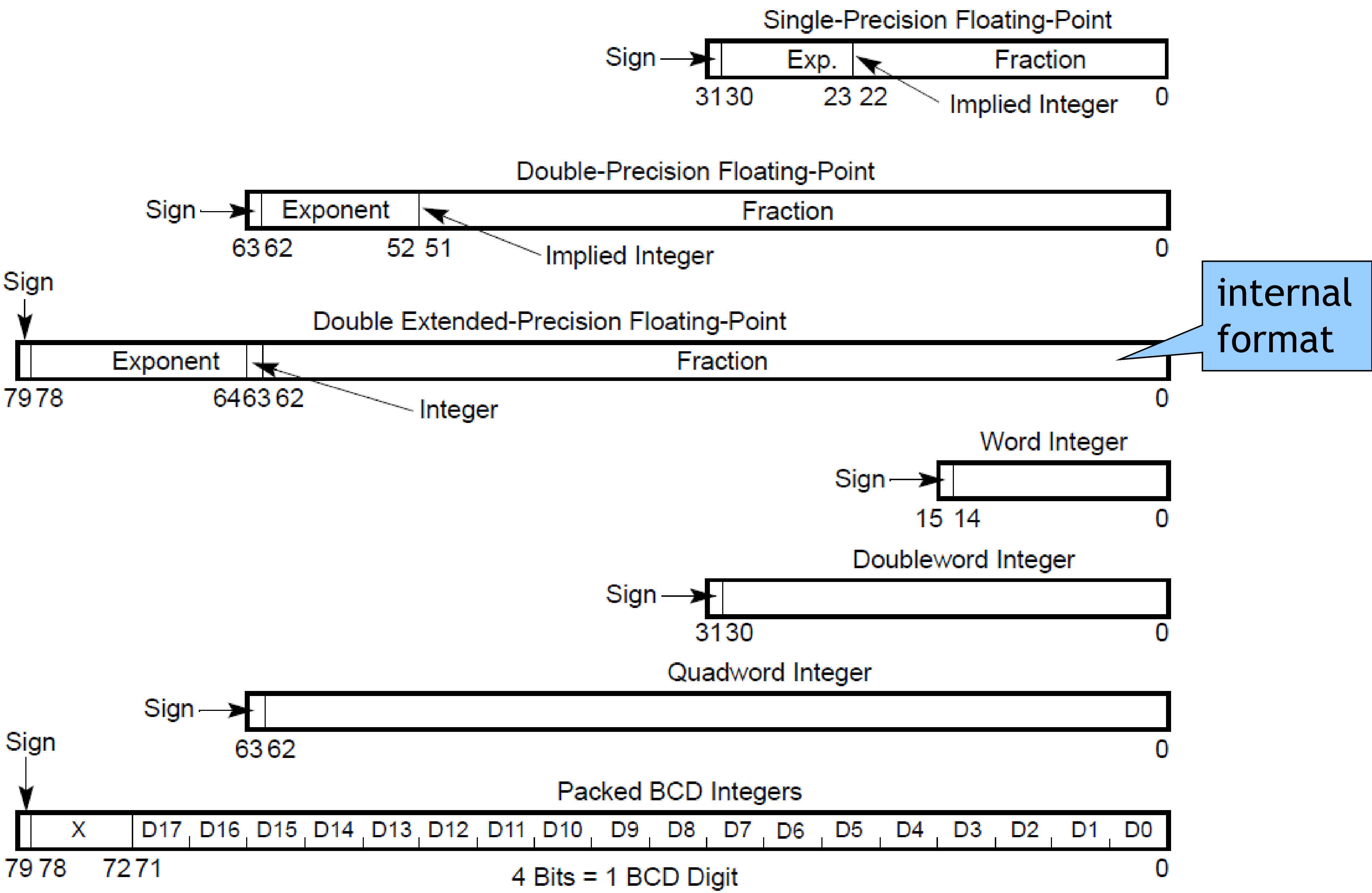
Floating point arithmetic

Operations on floating point numbers are performed in the Floating Point Unit (FPU).

The FPU supports seven different data types:

- floating point numbers represented in:
  - single precision (32 bits)
  - double precision (64 bits)
  - extended precision (80 bits) – internal format
- integers of type word, dword and qword
- packed BCD integers


Single-Precision Floating-Point
Sign
Exp.
Fraction
31 30
23 22
Implied Integer
0
Double-Precision Floating-Point
Sign
Exponent
Fraction
63 62
52 51
Implied Integer
0
Sign
Double Extended-Precision Floating-Point
Exponent
Fraction
79 78
64 63 62
Integer
0
internal format
Word Integer
Sign
15 14
0
Doubleword Integer
Sign
31 30
0
Quadword Integer
Sign
63 62
0
Sign
Packed BCD Integers
X
D17 D16 D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
79 78
72 71
4 Bits = 1 BCD Digit
0

Floating point numbers

A real number in the form 1.23 e 1 is defined by two numbers:

mantissa exponent

Value of a number is: mantissa $\times$ base raised to the power of exponent ($1.23 \times 10^1 = 12.3$)

Numbers are stored in a limited number of bits. As a consequence:

1. the range of numbers (determined by the exponent) is limited
2. the precision (given by the mantissa) is limited, i.e. the number of values between two consecutive numbers is limited

=> computation with real numbers approximates real arithmetic

Assume the format:

±m.mm±ee

Example 1: 1.23e1 + 4.56e0 = 1.23e1 + 0.456e1 = ?

a) 1.68e1 ... if two decimal places of the mantissa are used during the computation

b) 1.69e1 ... if three decimal places of the mantissa are used during the computation and the result is rounded

You get a more accurate result if you perform computation with numbers that have similar exponents.

Example 2: Add 1.00e0 ten times to the number 1.23e3.

Solution A:

1. step: 1.23e3 + 0.001e3 = 1.23e3
2. step: 1.23e3 + 0.001e3 = 1.23e3

...

Result: 1.23e3

Solution B:

1. step: 10 * 1.00e0 = 1.00e1
2. step: 1.23e3 + 0.01e3 = 1.24e3 ... correct result

The final result depends on the order in which partial operations are performed.

Multiplication and division increase the error.

Example 3: Multiply the result of the Example 2 by 2.

Solution A: $2 \times 1.23e3 = 2.46e3$

Solution B: $2 \times 1.24e3 = 2.48e3$ ... correct result

Multiply and divide first, then add and subtract.

$x * (y + z) \rightarrow x * y + x * z$

Comparison of real numbers in a computer program

A computer evaluates the relation x = y as true, only if all bits of the numbers x and y are identical.

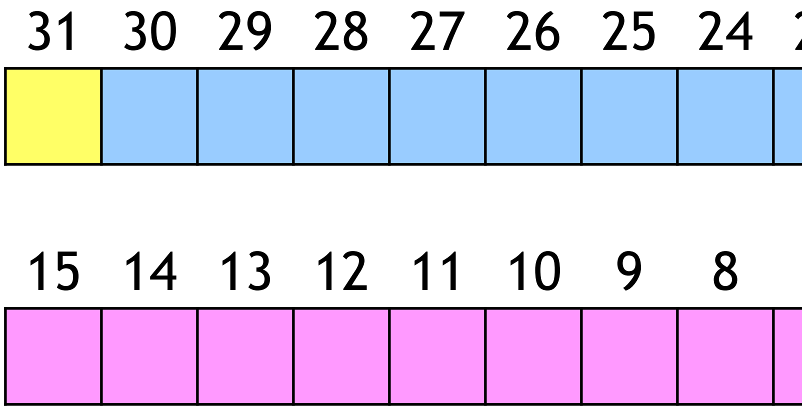
Tip: Choose a tolerance with which you consider two numbers equivalent.

x = y → if abs(x-y) <= tolerance then ...

x > y → if x-y > tolerance then ...

x < y → if x-y < -tolerance then ...

IEEE single precision format



mantissa

exponent

sign: + … 0, - … 1

The mantissa of a normal number is $1.m_{22}m_{21}...m_0$

m_{22}, m_{21}, ..., m_0 are significant digits

The leading 1 can be implied rather than explicitly present in the memory encoding.

The exponent value used in the arithmetic may be in the range $\langle -126; 127 \rangle$. In the encoding it is shifted by a bias so that the exponent could be an 8-bit unsigned integer $\in \langle 1; 254 \rangle$ (bias 127 is added to the original exponent value).

Exponents of 0 a 255 are reserved for special numbers, e.g.:

Zero: exponent: 0, mantissa: 0, sign: 0 or 1

+infinity: exponent: 255, mantissa: 0, sign: 0

Example: What is the single precision format of 0.3?

$(0.3)_{10} = (?)_2$

0.3 * 2

0.6 * 2

1.2 → 0.2 * 2

0.4 * 2

0.8 * 2

1.6 → 0.6 * 2

1.2 → 0.2 * 2

0.4 * 2

0.8 * 2

1.6 ...

Result: 0.0100110011001100110011001**10...**

In the normalized form, after rounding:

mantissa: $1.00110011001100110011010_2$

exponent: -2

Biased exponent: -2+127=125

$(125)_{10} = (?)_2$

Exponent encoding: 01111101

Mantissa: 1. 00110011001100110011010

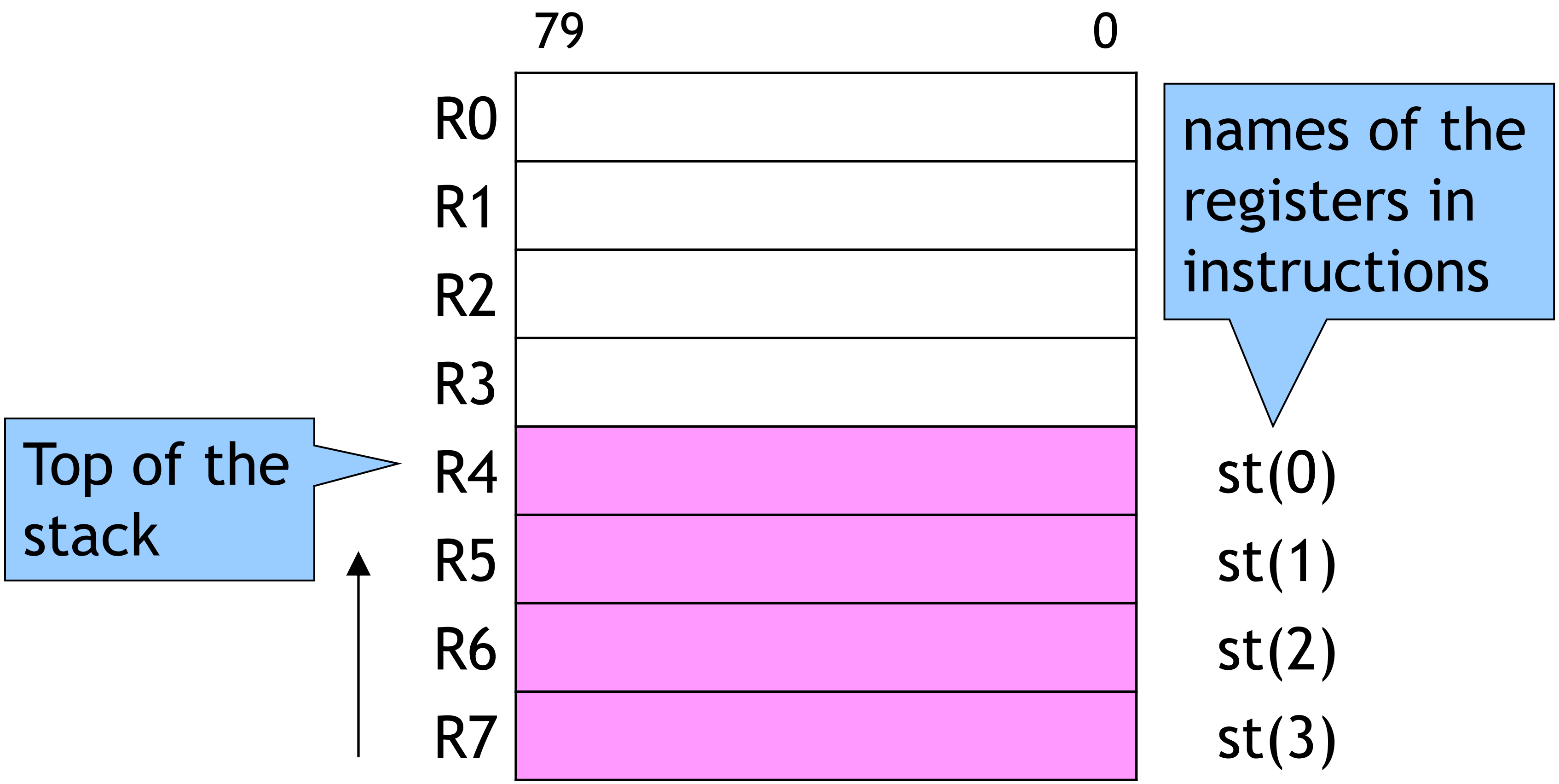
Single precision encoding of 0.3:

0 01111101 00110011001100110011010

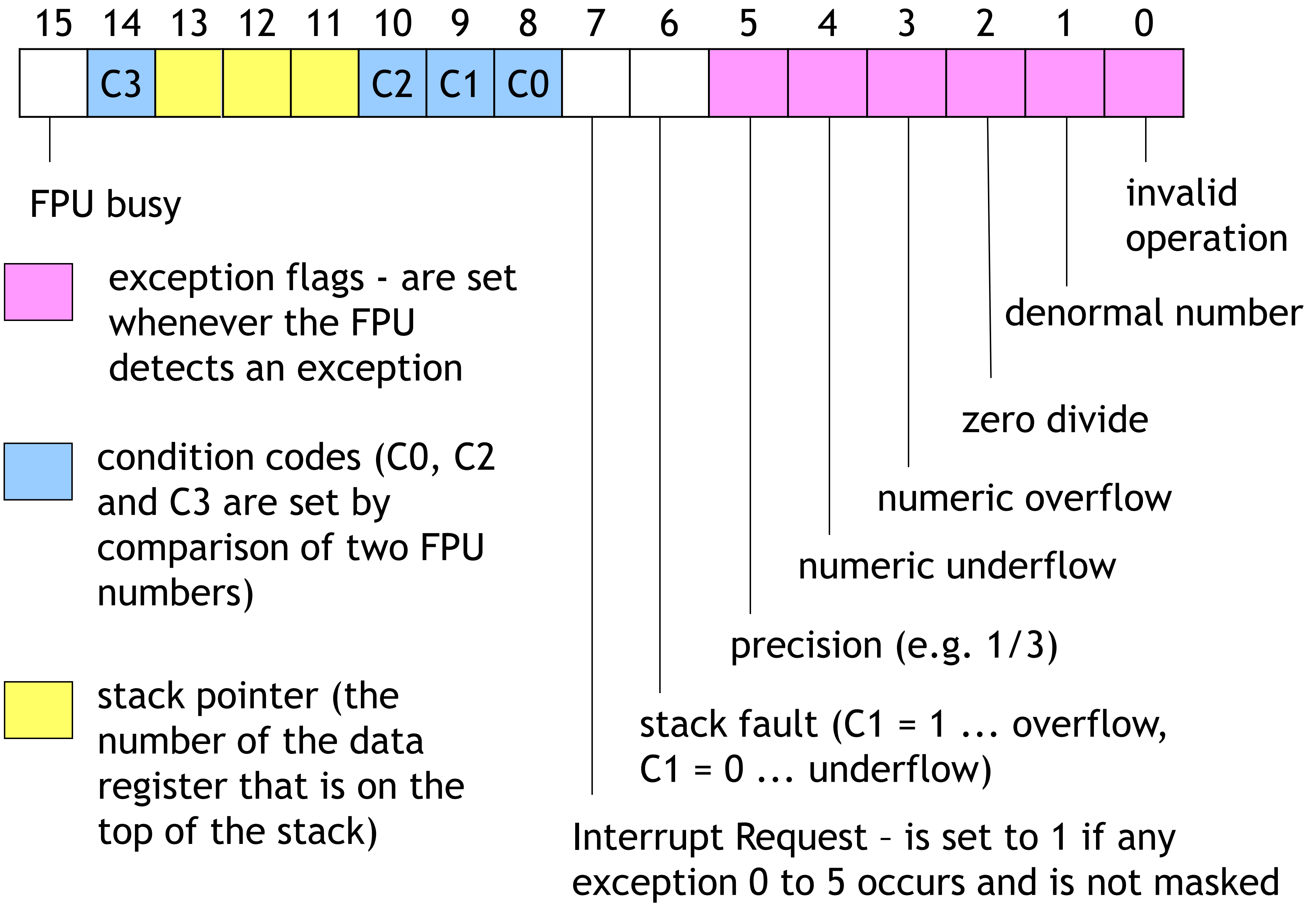
```
DD 0.3          ; 9A 99 99 3E
```

FPU data registers

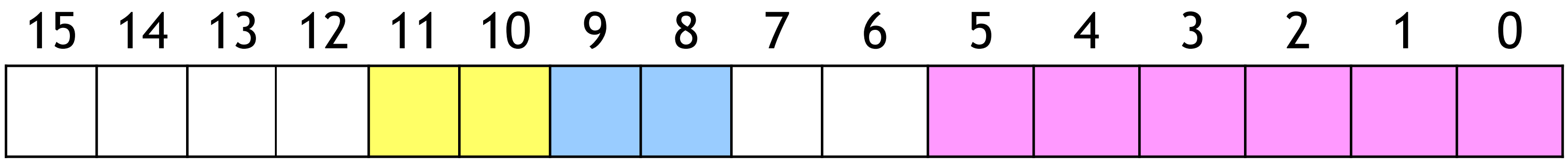
- 8 80-bit data registers R0 to R7
  - contain instruction operands
  - organized as a stack
  - instructions refer to them as st(0) to st(7), the index is relative to the top of the stack



FPU status register



FPU control register



exception masks: mask = 0 => when the corresponding condition occurs, then the FPU generates an interrupt
mask = 1 (default setting) => FPU indicates the corresponding exception in the status register but it does not initiate the interrupt

specify the precision during computation:
00 – 23 bits, 10 – 52 bits, 11 (default setting) – 63 bits

provide rounding control:
00 (default) – round to nearest
01 – round down (toward -∝)
10 – round up (toward +∝)
11 – truncate (toward 0)

Example: Round to 3 decimal places

Method	1.0111	-1.0111
round to nearest	1.100	-1.100
round down (toward -∝)	1.011	-1.100
round up (toward +∝)	1.100	-1.011
truncate (toward 0)	1.011	-1.011

FPU instruction set

Data movement instructions

fld real32 / real64 / real80 / st(i)	load
fild int16 / int32 / int64	integer load
fbld BCD	BCD load

- load the operand onto the floating point stack - copy the operand to st(0)
- automatically convert the operand to an 80 bit extended precision format (internal FPU format)
- operand: a variable or an st(i) register

fst real32 / real64 / st(i)	store
fist int16 / int32	integer store

- copy the value on the top of the floating point stack to another FPU register or a variable
- automatically convert the operand to the desired format

fstp real32 / real64 / real80 / st(i)	store and pop
fistp int16 / int32 / int64	integer store and pop
fbstp BCD	BCD store and pop

- copy the value on the top of the floating point stack to another FPU register or a variable and pop the value off the stack (i is index before pop)
- automatically convert the operand to the desired format

fxch st(i)   exchange registers

- exchanges the contents of the st(0) and st(i) registers
- fxch without an operand exchanges the contents of the st(0) and st(1) registers

Arithmetic instructions

```
fadd real32 / real64
fiadd int16 / int32
```

- add the operand to st(0)

```
fadd st(0), st(i)
fadd st(i), st(0)
```

- add the operands and store the result into the left operand

```
faddp st(i), st(0)
```

- st(i) = st(i) + st(0) and pops st(0) off the stack

FPU stack:

st(0)	10.1
st(1)	234.56

FPU stack after `fadd st(1),st(0)`:

st(0)	10.1
st(1)	244.66

`fadd` and `faddp` without operands do the same as `faddp st(1),st(0)`

FPU stack after fadd:

st(0)	244.66

Subtraction: fsub

Multiplication: fmul

Division: fdiv

Reverse subtraction: fsubr (swaps the operands)

fsub; st(1) = st(1) – st(0) and pop st(0)

fsubr; st(1) = st(0) – st(1) and pop st(0)

Reverse division: fdivr (swaps the dividend and divisor)

Instruction	Operation with st(0)
`fchs`	st(0) = - st(0)
`fsqrt`	$st(0) = \sqrt{st(0)}$
`fabs`	st(0) = |st(0)|
`fsin`	st(0) = sin(st(0))
`fcos`	st(0) = cos(st(0))

Comparison instructions

fcom real32 / real64 / st(i)	compare
ficom int16 / int32	integer compare

- compare the operand with st(0) and set condition bits C0, C2 and C3 in the status register.

fcomp real32 / real64 / st(i)	compare and pop
ficomp int16 / int32	integer compare and pop

- compare the operand with st(0), set condition bits C0, C2 and C3 in the status register and pop st(0) off the stack.

fcom and fcomp without an operand have the implicit operand st(1)

fcompp compare and pop twice

- compares st(1) with st(0), sets condition bits C0, C2 and C3 in the status register and pops st(0) and st(1) off the stack.

ftst test stack top

- compares st(0) to 0.

Condition code bits C3, C2 and C0 after comparison :

C3	C2	C0	Condition
0	0	0	st(0) > operand
0	0	1	st(0) < operand
1	0	0	st(0) = operand
1	1	1	st(0) or source undefined

Control instructions

finit

- initializes the FPU – the status register is set to 0, the control register is set as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	0	1	1	1	1	1	1	1

fldcw memory

- loads the control register from a 16-bit memory location.

fstcw memory

- stores the control register to a 16-bit memory location.

fstsw memory/AX

- stores the status register to a 16-bit memory location or to AX.

Converting floating point expressions to assembly language

The FPU uses postfix (reverse, polish) notation, that places the operator after the operands, as opposed to standard infix notation, which places the operator between the operands.

a+b → ab+

Infix notation requires brackets to change the order of the operations. Postfix notation does not need brackets:

(a+b)*(c+d) → ab+cd+*

- Write the expression ((a+b)/c)*(e-f) in postfix notation.

(a+b)*(c+d) → ab+cd+*

=> first you have to push the operands onto the stack and then execute the arithmetic or comparison instruction.

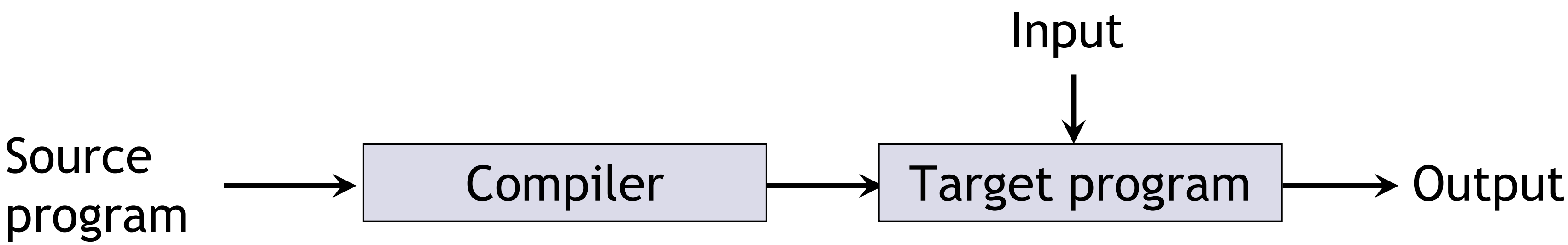
	st(0)	st(1)	st(2)
finit;			
fld a;	a		
fld b;	b	a	
fadd;	a+b		
fld c;	c	a+b	
fld d;	d	c	a+b
fadd;	c+d	a+b	
fmul;	(a+b)*(c+d)		

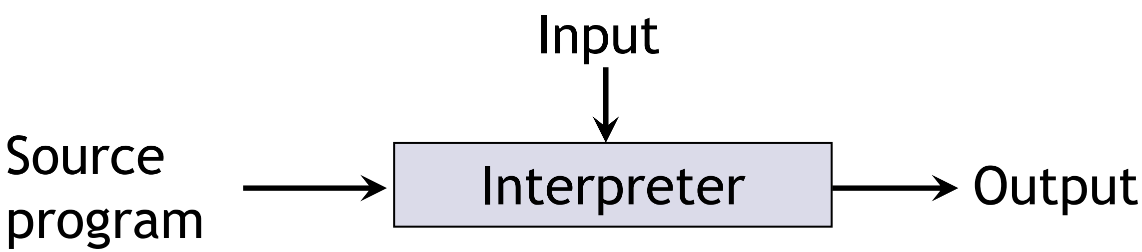
- Write a computer program that evaluates the expression D = - A + (B * C).

```
.data
A DD 1.5
B DD 2.5
C DD 3.0
D DD ?
.code
```

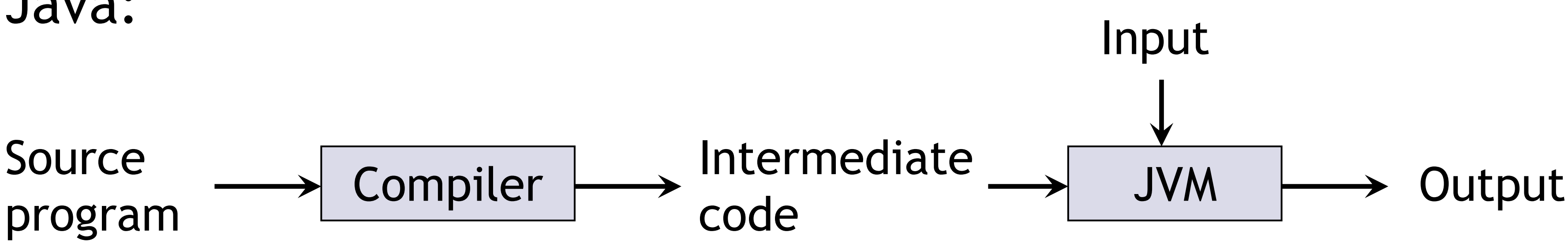
Java Virtual Machine

The difference between a compiled and interpreted language:





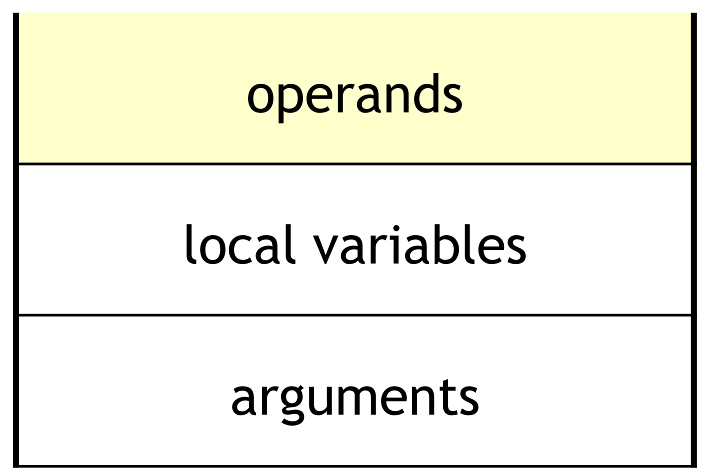
Java:



Java Virtual Machine

- interprets the intermediate code (bytecode)
- bytecode consists of simple instructions in assembly language style
- bytecode is in the .class file that can be view by the javap command
- stack machine - actual parameters (arguments) of the method, local variables and operands are stored in the memory organized as a stack, all calculations are only with the stack operands (no registers) in reverse notation.

Stack frame:



Java:

```
public void Sucet() {
        int x;
        int y=0;
        int z=1;
        x = y+z;
    }
```

Bytecode:

```
0: iconst_0   load the int value 0 onto the stack
1: istore_2   pop int value from the stack and store it into
   local variable 2
2: iconst_1
3: istore_3
4: iload_2    load an int value from local variable 2
5: iload_3
6: iadd       pop two int values from the stack, add them and
   push the result onto the stack
7: istore_1
8: return
```