
Digital Design: A Systems Approach

Lecture 4: Numbers and Arithmetic

Readings


- L4: Chapters 10 & 11
- L5: Review Lectures 1-4

Numbers

- Digital logic works with *binary* numbers
 - Each bit has a value based on its position
 - e.g., 10101
 - $1+4+16 = 21$
 - Or $1+4-16 = -11$

From Decimal To Binary

- Example: 17_{10}
- Convert number to binary base:

| | | | | |
|-----------------|-----------|---|---|-----|
| $17 \div 2 = 8$ | remainder | 1 |  | LSB |
| $8 \div 2 = 4$ | remainder | 0 | | |
| $4 \div 2 = 2$ | remainder | 0 | | |
| $2 \div 2 = 1$ | remainder | 0 | | |
| $1 \div 2 = 0$ | remainder | 1 | | MSB |

- $17_{10} = 10001_2$

From Decimal To Binary To Hexadecimal

- Example: 1963_{10}
- $1963_{10} = 0111\ 1010\ 1011_2$

 ┌───┐ ┌───┐ ┌───┐
 ↓ ↓ ↓
 7 A B_{hex}

Binary addition

1-bit $0+0 = 0$, $1+0 = 1$, $1+1 = 10$

$$\begin{array}{r} \text{110} \\ 6 \text{ 110} \\ + 3 \text{ 011} \\ \hline 9 \text{ 1001} \end{array}$$

$$\begin{array}{r} \text{111} \\ 5 \text{ 101} \\ + 7 \text{ 111} \\ \hline 12 \text{ 1100} \end{array}$$

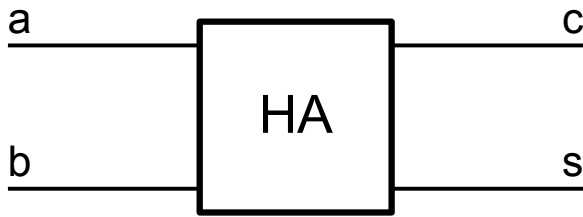
| c[i] | b[i] | a[i] | count | c[i+1] | s[i] |
|------|------|------|-------|--------|------|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 2 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 2 | 1 | 0 |
| 1 | 1 | 0 | 2 | 1 | 0 |
| 1 | 1 | 1 | 3 | 1 | 1 |

One bit of an adder

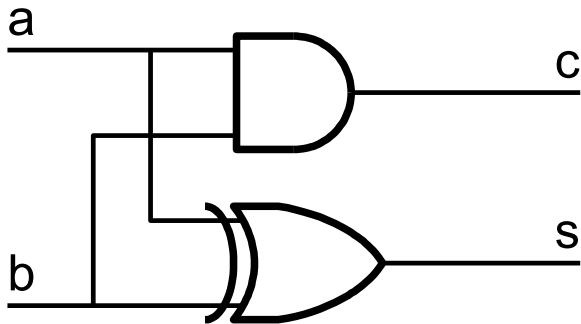
- Counts the number of “1” bits on its input
- Outputs the result in binary
- For a half-adder, 2 inputs, output can be 0, 1, or 2
- For a full-adder, 3 inputs, output can be 0, 1, 2, or 3

| c[i] | b[i] | a[i] | count | c[i+1] | s[i] |
|------|------|------|-------|--------|------|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 2 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 2 | 1 | 0 |
| 1 | 1 | 0 | 2 | 1 | 0 |
| 1 | 1 | 1 | 3 | 1 | 1 |

Half Adder



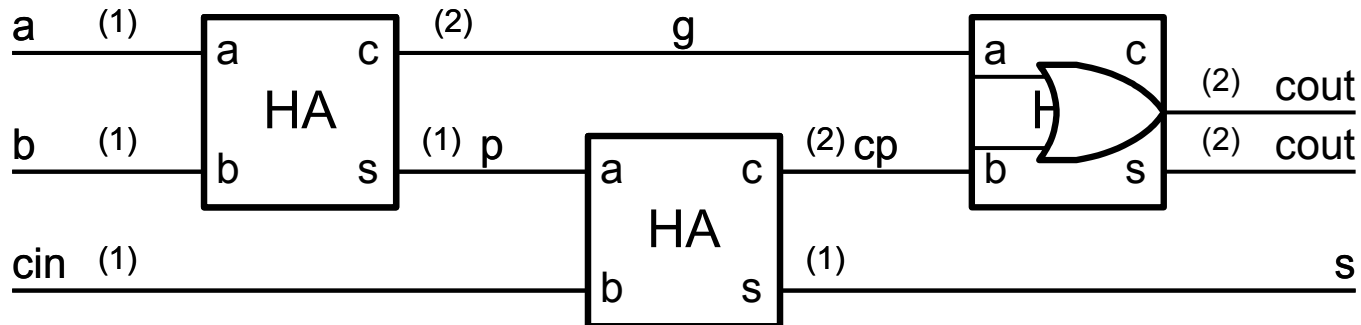
```
// half adder
module HalfAdder(a,b,c,s) ;
    input a,b ;
    output c,s ;    // carry and sum
    wire s = a ^ b ;
    wire c = a & b ;
endmodule
```



| | | |
|---|----|----|
| # | 00 | 00 |
| # | 01 | 01 |
| # | 10 | 01 |
| # | 11 | 10 |

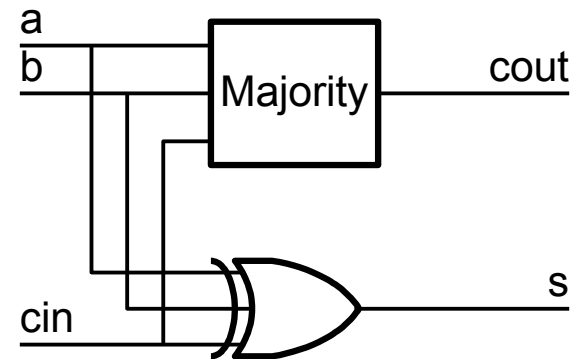

```
// full adder - from half adders
module FullAdder1(a,b,cin,cout,s) ;
    input a,b,cin ;
    output cout,s ; // carry and sum
    wire g,p ;      // generate and propagate
    wire cp ;
    HalfAdder ha1(a,b,g,p) ;
    HalfAdder ha2(cin,p,cp,s) ;
    or o1(cout,g,cp) ;
endmodule
```

| | | |
|---|-----|----|
| # | 000 | 00 |
| # | 001 | 01 |
| # | 010 | 01 |
| # | 011 | 10 |
| # | 100 | 01 |
| # | 101 | 10 |
| # | 110 | 10 |
| # | 111 | 11 |



Full adder from a truth table

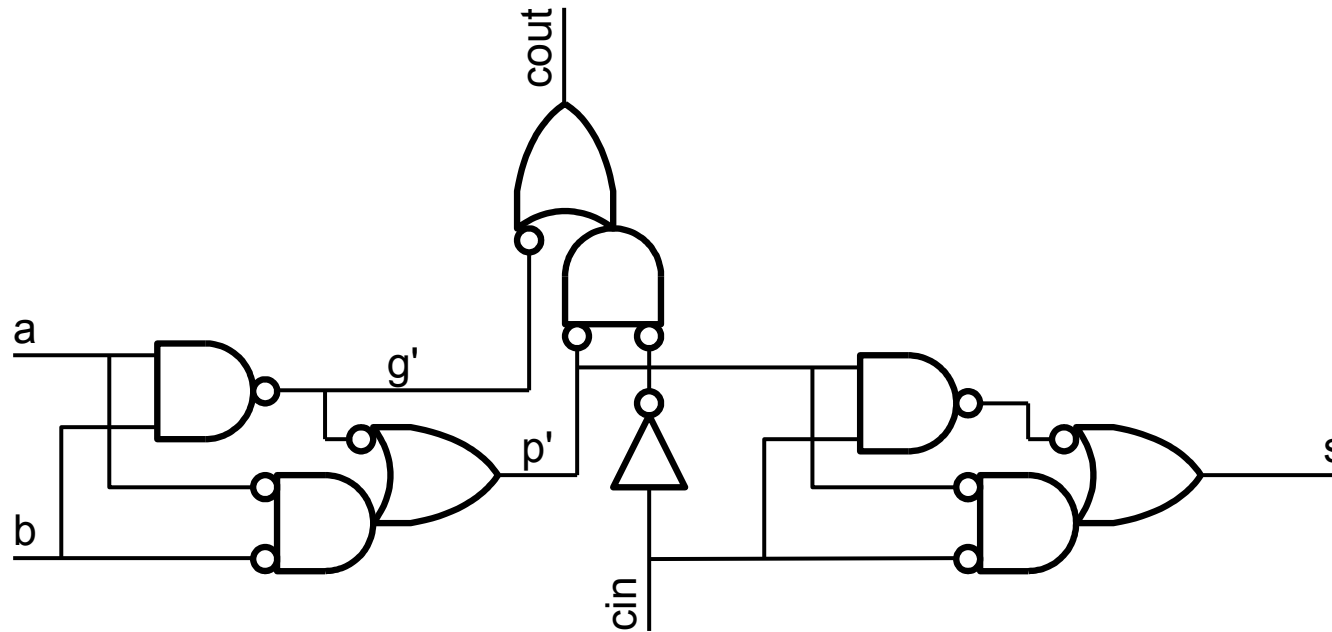
| c[i] | b[i] | a[i] | count | c[i+1] | s[i] |
|------|------|------|-------|--------|------|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 2 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 2 | 1 | 0 |
| 1 | 1 | 0 | 2 | 1 | 0 |
| 1 | 1 | 1 | 3 | 1 | 1 |



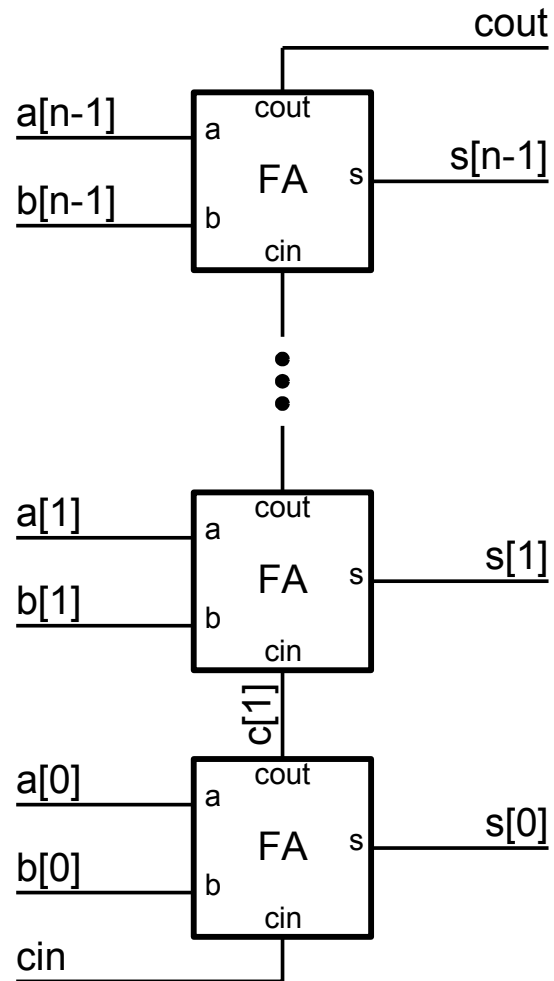
Full adder from truth table

```
// full adder - logical
module FullAdder2(a,b,cin,cout,s) ;
    input a,b,cin ;
    output cout,s ; // carry and sum
    wire s = a ^ b ^ cin ;
    wire cout = (a & b) | (a & cin) | (b & cin) ; // majority
endmodule
```

CMOS Version of Full Adder



Multi-bit Adder



Adder in verilog - behavioral

```
// multi-bit adder - behavioral
module Adder1(a,b,cin,cout,s) ;
    parameter n = 8 ;
    input [n-1:0] a, b ;
    input cin ;
    output [n-1:0] s ;
    output cout ;
    wire [n-1:0] s;
    wire cout ;

    assign {cout, s} = a + b + cin ;
endmodule
```

Ripple-carry adder – bit-slice notation

```
// multi-bit adder - with vectors
module Adder2(a,b,cin,cout,s) ;
    parameter n = 8 ;
    input [n-1:0] a, b ;
    input cin ;
    output [n-1:0] s ;
    output cout ;

    wire [n-1:0] p = a ^ b ;
    wire [n-1:0] g = a & b ;
    wire [n:0]    c = {g | (p & c[n-1:0]), cin} ;
    wire [n-1:0] s = p ^ c[n-1:0] ;
    wire          cout = c[n] ;
endmodule
```

Negative Integers

- Thus far we have only addressed positive integers. What about negative numbers?
- 3 ways to represent negative integers in binary:
 - Sign Magnitude
 - One's complement
 - Two's complement
- Example: Consider

| | +23 | −23 |
|--------------------|---------|---------|
| – Sign Magnitude | 0 10111 | 1 10111 |
| – One's complement | 0 10111 | 1 01000 |
| – Two's complement | 0 10111 | 1 01001 |

Why do we all use 2's complement?

2's complement makes subtraction easy

Represent negative number, $-x$ as $2^n - x$

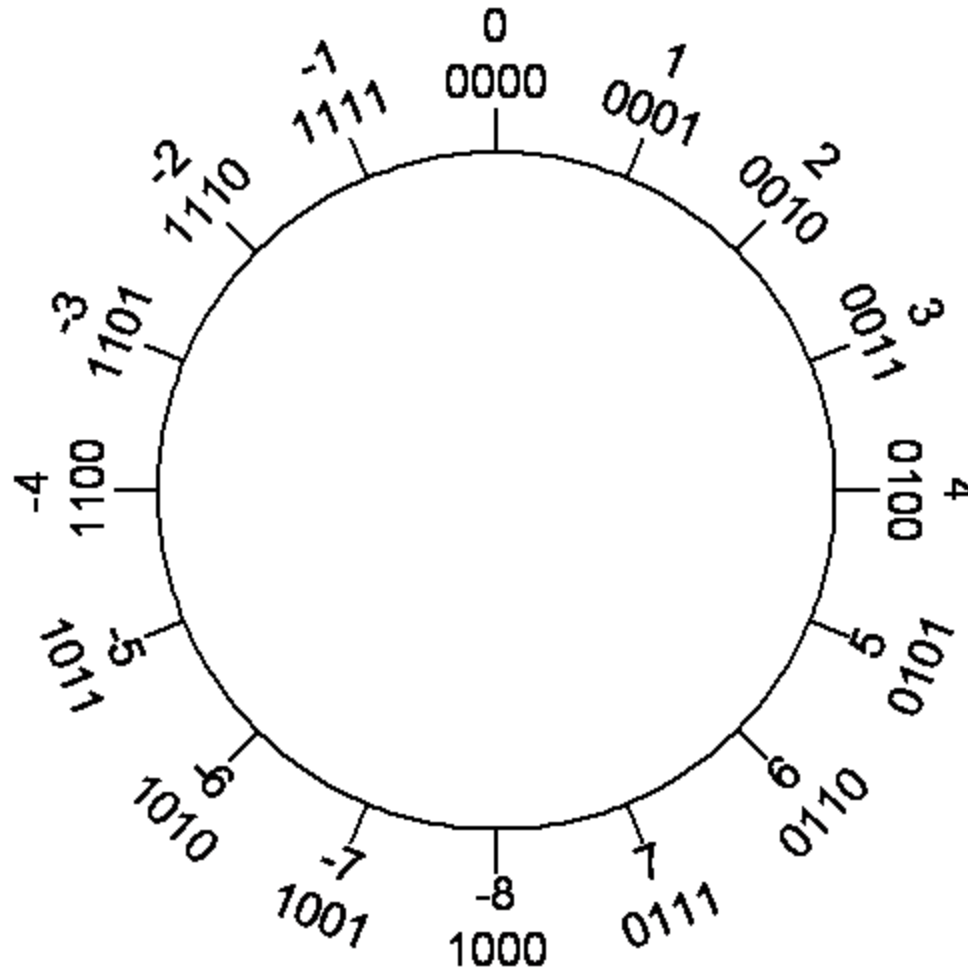
All arithmetic is done modulo 2^n so no adjustments are necessary

$$x + (-y) = x + (2^n - y) \pmod{2^n}$$

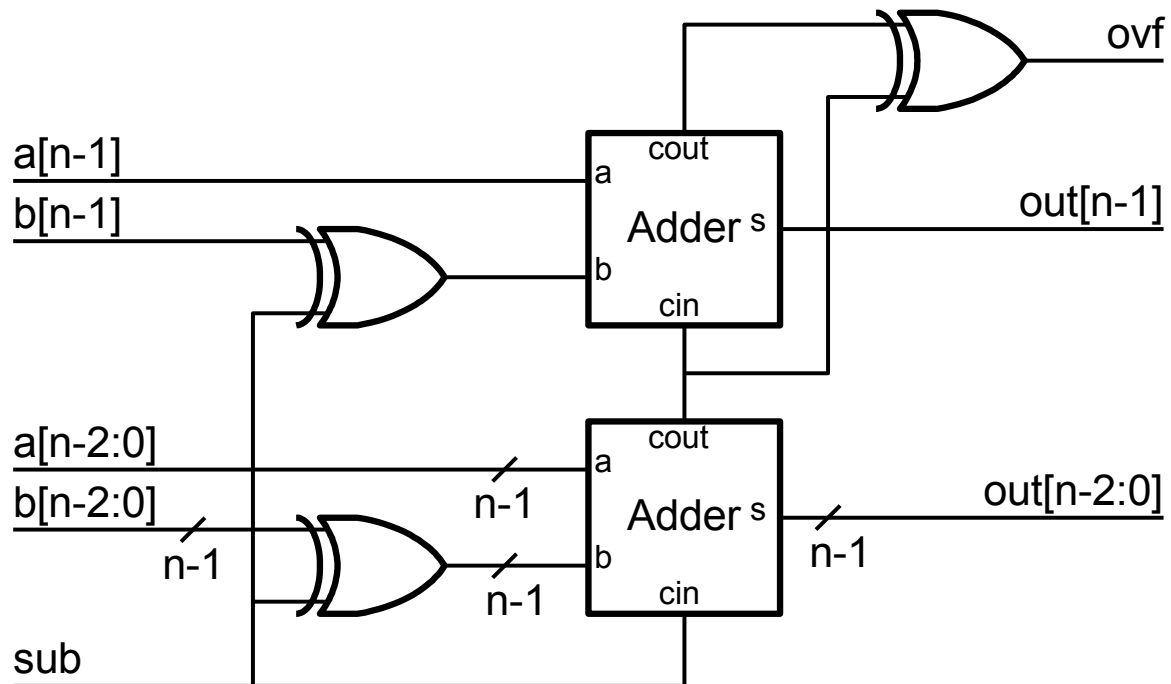
consider 4-bit numbers

$$\begin{aligned} 4 - 3 &= 4 + (16 - 3) \pmod{16} \\ &= 4 + (15 - 3 + 1) \\ &= 0100 + (1111 - 0011) + 0001 \\ &= 0100 + 1100 + 0001 \\ &= 0001 \end{aligned}$$

2's Complement



$$v = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$



```
// add a+b or subtract a-b, check for overflow
module AddSub(a,b,sub,s,ovf) ;
    parameter n = 8 ;
    input [n-1:0] a, b ;
    input sub ;           // subtract if sub=1, otherwise add
    output [n-1:0] s ;
    output ovf ;          // 1 if overflow
    wire c1, c2 ;         // carry out of last two bits
    wire ovf = c1 ^ c2 ;  // overflow if signs don't match

    // add non sign bits
    Adder1 #(n-1) ai(a[n-2:0],b[n-2:0]^{n-1{sub}},sub,c1,s[n-2:0]) ;
    // add sign bits
    Adder1 #(1) as(a[n-1],b[n-1]^sub,c1,c2,s[n-1]) ;
endmodule
```

```
// test script
module test2 ;

    reg [7:0] in0, in1 ;
    wire [7:0] out ;
    reg sub ;
    wire ovf ;

    AddSub #(8) a(in0,in1,sub,out,ovf) ;

    initial begin
        in0 = 8'd87 ; in1 = 8'd40 ; sub = 0 ;
        #100 $display("%03h + %03h = %03h ovf = %b", in0, in1, out, ovf) ;
        in0 = 8'd87 ; in1 = 8'd40 ; sub = 1 ;
        #100 $display("%03h - %03h = %03h ovf = %b", in0, in1, out, ovf) ;
        in0 = 8'd88 ; in1 = 8'd40 ; sub = 0 ;
        #100 $display("%03h + %03h = %03h ovf = %b", in0, in1, out, ovf) ;
        in0 = 8'he9 ; in1 = 8'h17 ; sub = 1 ; /* -23 - 23 */
        #100 $display("%03h - %03h = %03h ovf = %b", in0, in1, out, ovf) ;
        in0 = 8'he9 ; in1 = 8'h97 ; sub = 0 ; /* -23 - 105 = -128 no ovf */
        #100 $display("%03h + %03h = %03h ovf = %b", in0, in1, out, ovf) ;
        in0 = 8'he9 ; in1 = 8'd106 ; sub = 1 ; /* overflow */
        #100 $display("%03h - %03h = %03h ovf = %b", in0, in1, out, ovf) ;
    end
endmodule
```

```
# 57 + 28 = 7f ovf = 0
# 57 - 28 = 2f ovf = 0
# 58 + 28 = 80 ovf = 1
# e9 - 17 = d2 ovf = 0
# e9 + 97 = 80 ovf = 0
# e9 - 6a = 7f ovf = 1
```

```
reg [8:0] ref, in0, in1 ; // reference result, inputs kept longer
Reg rovf ;
```

```
...
initial begin
    // generate test cases

    // check each against reference
    if(sub) ref = in0-in1 ;
    else ref = in0+in1 ;
    rovf = (ref[8] != ref[7]) ;
    #100
    if ((out != ref[7:0]) || (ovf != rovf)) $display("ERROR") ;

    // more cases
end
```

Compare two numbers with subtraction

$$\text{diff} = a - b$$

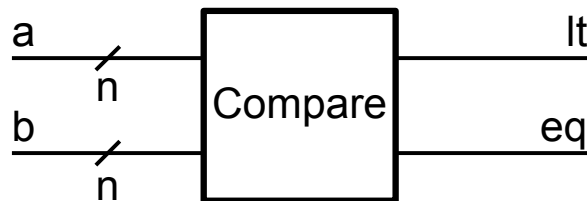
if diff is negative (sign bit = 1), $a < b$

If diff is zero, $a = b$

Example, compare $a = 0101$ and $b = 0110$

$$\text{diff} = a - b = 1111 \Rightarrow a < b$$

Compare $a = 0111$ and $b = 0111$, $\text{diff} = 0 \Rightarrow a = b$



Multiplication

Multiplication

Shifting left multiplies by 2

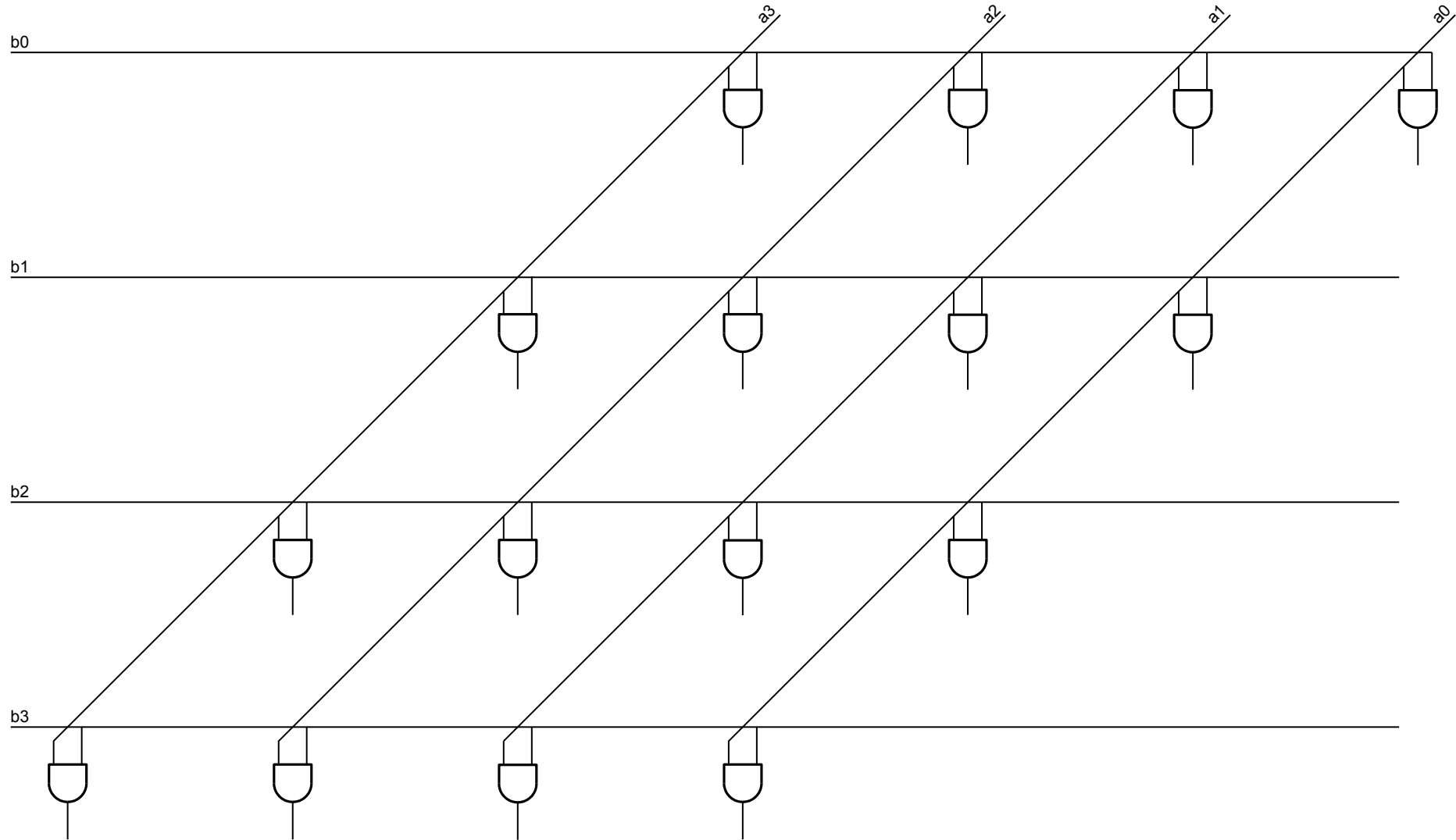
e.g., $5 = 101$, $10 = 1010$, $20 = 10100$

To multiply by 3, compute $3x = x + 2x$

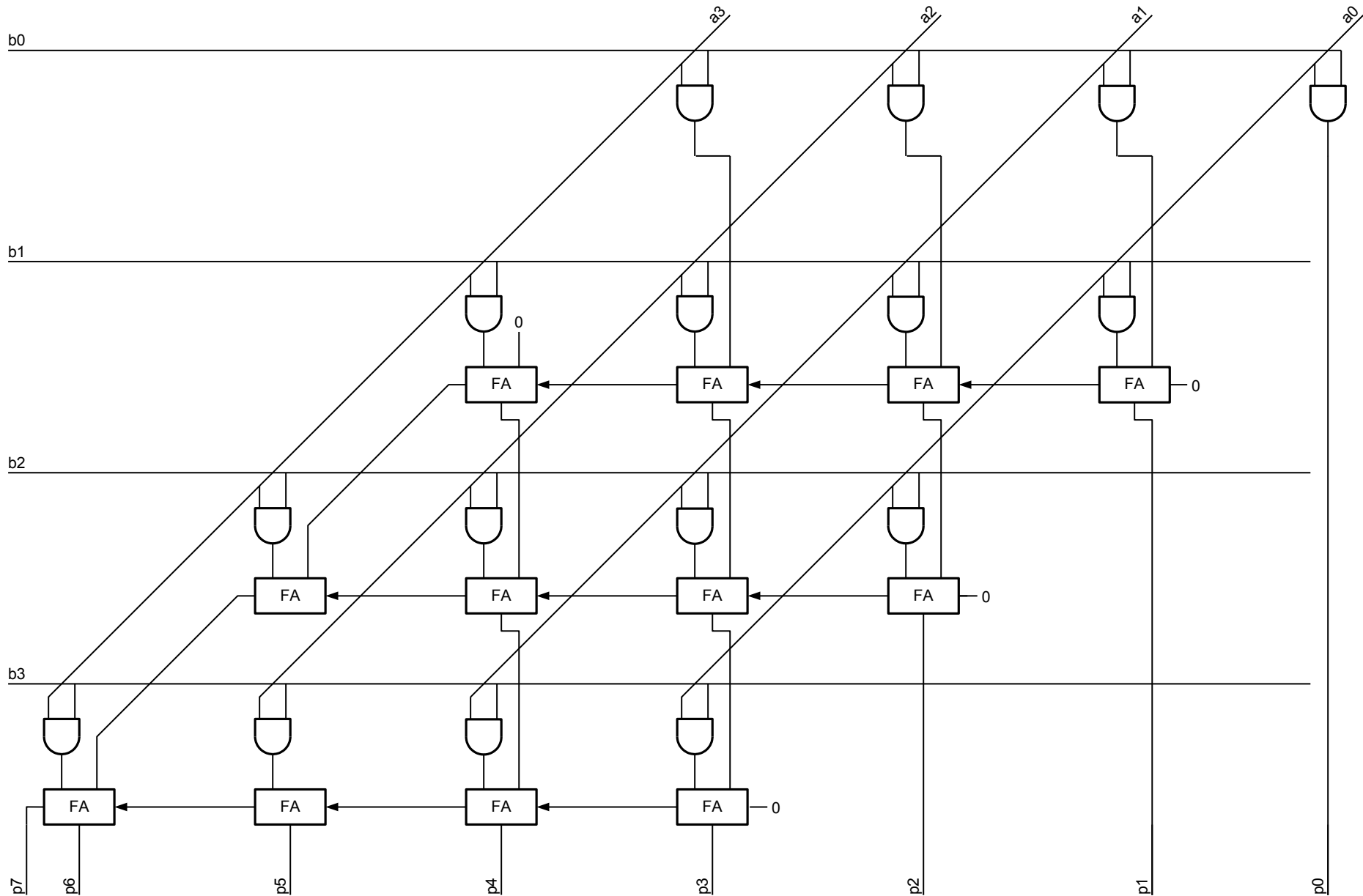
Example, 7×5

$$\begin{array}{r} 111 \\ x 101 \\ \hline 111 \\ 000 \\ 111 \\ \hline 100011 \end{array}$$

First generate partial products: $p_{ij} = a_i \wedge b_j$ has weight $i+j$



Then sum partial products to get sum



```
// 4-bit multiplier
module mul4(a,b,p) ;
  input [3:0] a,b ;
  output [7:0] p ;

  // form partial products
  wire [3:0] pp0 = a & {4{b[0]}} ; // x1
  wire [3:0] pp1 = a & {4{b[1]}} ; // x2
  wire [3:0] pp2 = a & {4{b[2]}} ; // x4
  wire [3:0] pp3 = a & {4{b[3]}} ; // x8

  // sum up partial products
  wire cout1, cout2, cout3 ;
  wire [3:0] s1, s2, s3 ;
  Adder1 #(4) a1(pp1, {0,pp0[3:1]}, 0, cout1, s1) ;
  Adder1 #(4) a2(pp2, {cout1,s1[3:1]}, 0, cout2, s2) ;
  Adder1 #(4) a3(pp3, {cout2,s2[3:1]}, 0, cout3, s3) ;

  // collect the result
  wire [7:0] p = {cout3, s3, s2[0], s1[0], pp0[0]} ;
endmodule
```

```
// test script
module test3 ;

    reg [3:0] in0, in1 ;
    wire [7:0] out ;

    Mul4 mul(in0,in1,out) ;

    initial begin
        in0 = 0 ;
        repeat (16) begin
            in1 = 0 ;
            repeat (in0+1) begin
                #100 $display("%03d * %03d = %03d",in0,in1,out) ;
                in1 = in1 + 1 ;
            end
            in0 = in0 + 1 ;
        end
    end
endmodule
```

| | | |
|--------------|-----------------|-----------------|
| # 0 * 0 = 0 | # 9 * 0 = 0 | # 13 * 0 = 0 |
| # 1 * 0 = 0 | # 9 * 1 = 9 | # 13 * 1 = 13 |
| # 1 * 1 = 1 | # 9 * 2 = 18 | # 13 * 2 = 26 |
| # 2 * 0 = 0 | # 9 * 3 = 27 | # 13 * 3 = 39 |
| # 2 * 1 = 2 | # 9 * 4 = 36 | # 13 * 4 = 52 |
| # 2 * 2 = 4 | # 9 * 5 = 45 | # 13 * 5 = 65 |
| # 3 * 0 = 0 | # 9 * 6 = 54 | # 13 * 6 = 78 |
| # 3 * 1 = 3 | # 9 * 7 = 63 | # 13 * 7 = 91 |
| # 3 * 2 = 6 | # 9 * 8 = 72 | # 13 * 8 = 104 |
| # 3 * 3 = 9 | # 9 * 9 = 81 | # 13 * 9 = 117 |
| # 4 * 0 = 0 | # 10 * 0 = 0 | # 13 * 10 = 130 |
| # 4 * 1 = 4 | # 10 * 1 = 10 | # 13 * 11 = 143 |
| # 4 * 2 = 8 | # 10 * 2 = 20 | # 13 * 12 = 156 |
| # 4 * 3 = 12 | # 10 * 3 = 30 | # 13 * 13 = 169 |
| # 4 * 4 = 16 | # 10 * 4 = 40 | # 14 * 0 = 0 |
| # 5 * 0 = 0 | # 10 * 5 = 50 | # 14 * 1 = 14 |
| # 5 * 1 = 5 | # 10 * 6 = 60 | # 14 * 2 = 28 |
| # 5 * 2 = 10 | # 10 * 7 = 70 | # 14 * 3 = 42 |
| # 5 * 3 = 15 | # 10 * 8 = 80 | # 14 * 4 = 56 |
| # 5 * 4 = 20 | # 10 * 9 = 90 | # 14 * 5 = 70 |
| # 5 * 5 = 25 | # 10 * 10 = 100 | # 14 * 6 = 84 |
| # 6 * 0 = 0 | # 11 * 0 = 0 | # 14 * 7 = 98 |
| # 6 * 1 = 6 | # 11 * 1 = 11 | # 14 * 8 = 112 |
| # 6 * 2 = 12 | # 11 * 2 = 22 | # 14 * 9 = 126 |
| # 6 * 3 = 18 | # 11 * 3 = 33 | # 14 * 10 = 140 |
| # 6 * 4 = 24 | # 11 * 4 = 44 | # 14 * 11 = 154 |
| # 6 * 5 = 30 | # 11 * 5 = 55 | # 14 * 12 = 168 |
| # 6 * 6 = 36 | # 11 * 6 = 66 | # 14 * 13 = 182 |
| # 7 * 0 = 0 | # 11 * 7 = 77 | # 14 * 14 = 196 |
| # 7 * 1 = 7 | # 11 * 8 = 88 | # 15 * 0 = 0 |
| # 7 * 2 = 14 | # 11 * 9 = 99 | # 15 * 1 = 15 |
| # 7 * 3 = 21 | # 11 * 10 = 110 | # 15 * 2 = 30 |
| # 7 * 4 = 28 | # 11 * 11 = 121 | # 15 * 3 = 45 |
| # 7 * 5 = 35 | # 12 * 0 = 0 | # 15 * 4 = 60 |
| # 7 * 6 = 42 | # 12 * 1 = 12 | # 15 * 5 = 75 |
| # 7 * 7 = 49 | # 12 * 2 = 24 | # 15 * 6 = 90 |
| # 8 * 0 = 0 | # 12 * 3 = 36 | # 15 * 7 = 105 |
| # 8 * 1 = 8 | # 12 * 4 = 48 | # 15 * 8 = 120 |
| # 8 * 2 = 16 | # 12 * 5 = 60 | # 15 * 9 = 135 |
| # 8 * 3 = 24 | # 12 * 6 = 72 | # 15 * 10 = 150 |
| # 8 * 4 = 32 | # 12 * 7 = 84 | # 15 * 11 = 165 |
| # 8 * 5 = 40 | # 12 * 8 = 96 | # 15 * 12 = 180 |
| # 8 * 6 = 48 | # 12 * 9 = 108 | # 15 * 13 = 195 |
| # 8 * 7 = 56 | # 12 * 10 = 120 | # 15 * 14 = 210 |
| # 8 * 8 = 64 | # 12 * 11 = 132 | # 15 * 15 = 225 |
| | # 12 * 12 = 144 | |

Accuracy

Resolution

Value and Representation functions

- $R(x)$ is binary number representing real number x
- $V(b)$ is value of binary number b
- Absolute error
 - $e_a = |V(R(x)) - x|$
- Relative error
 - $e_r = |(V(R(x)) - x)/x|$

Example

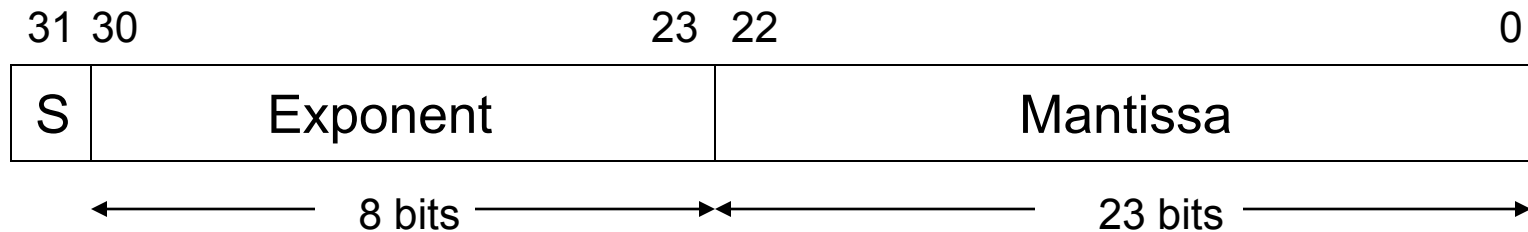
- Suppose you need to represent temperatures from 0 to 100C to 0.1C
- What representation would you use?
 - What is the resolution of this representation?
 - What is its maximum absolute error?

Example

- Suppose you need to represent distances from 1mm to 1m with an accuracy of 0.1%
- How many bits are required to do this with a binary fixed-point number?
- How many bits are needed with binary floating point?
- What floating-point representation is needed?

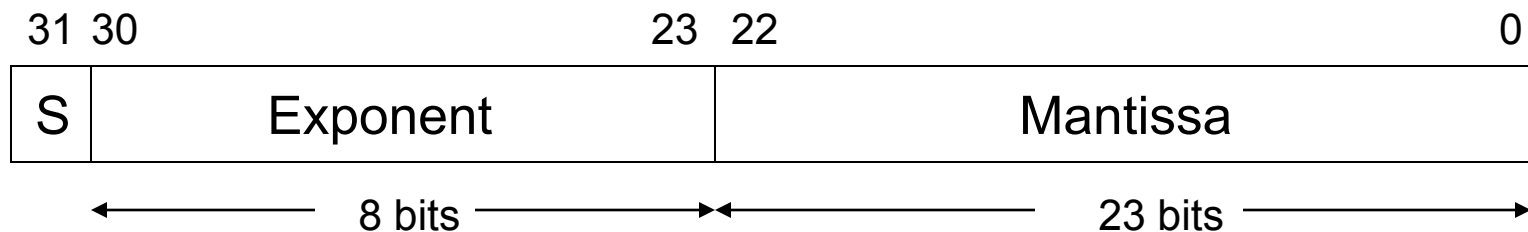
Binary Floating Point

Floating-Point Representation



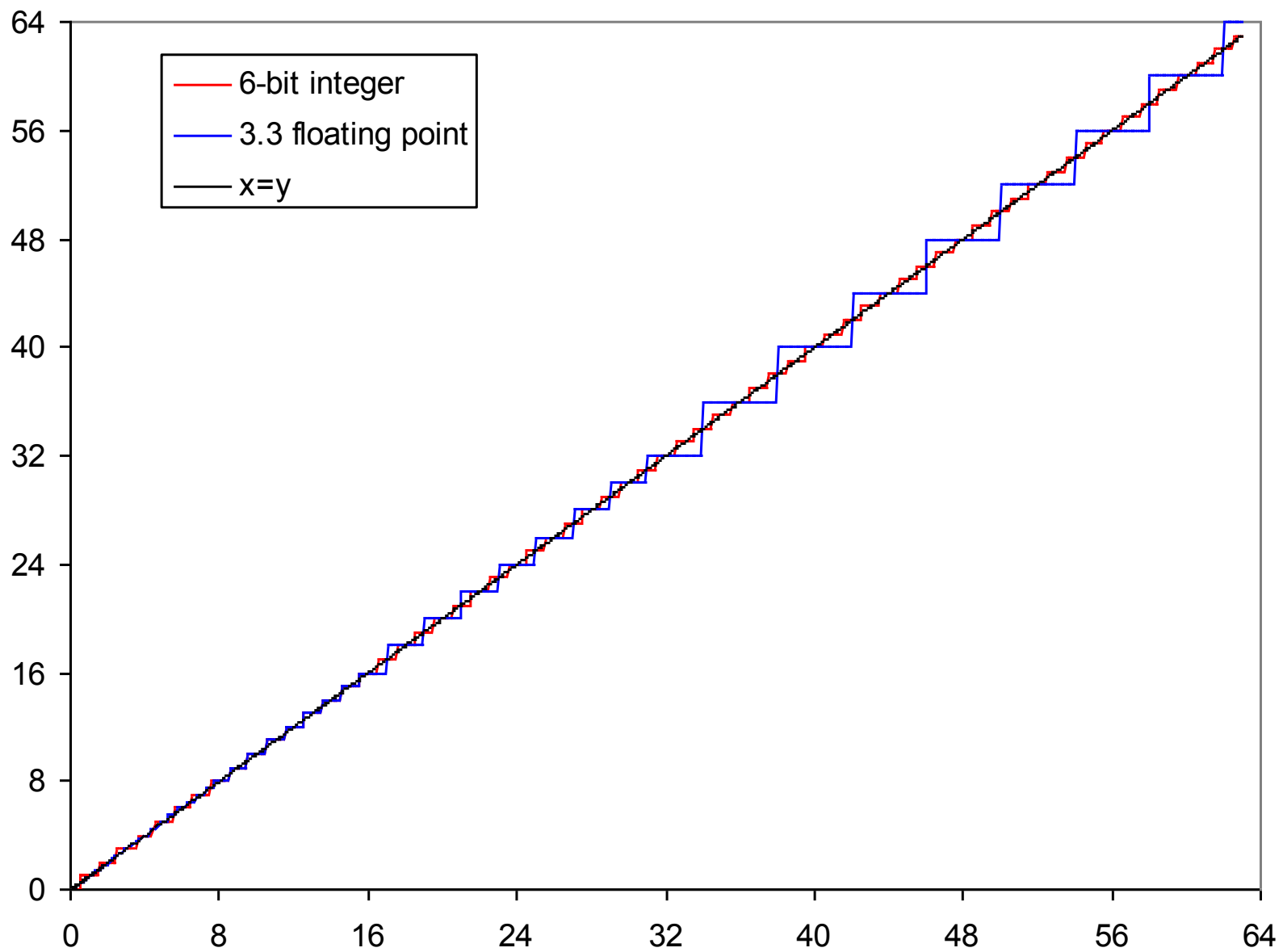
$$\text{Floating-Point Format: } (-1)^S \times M \times 2^E$$

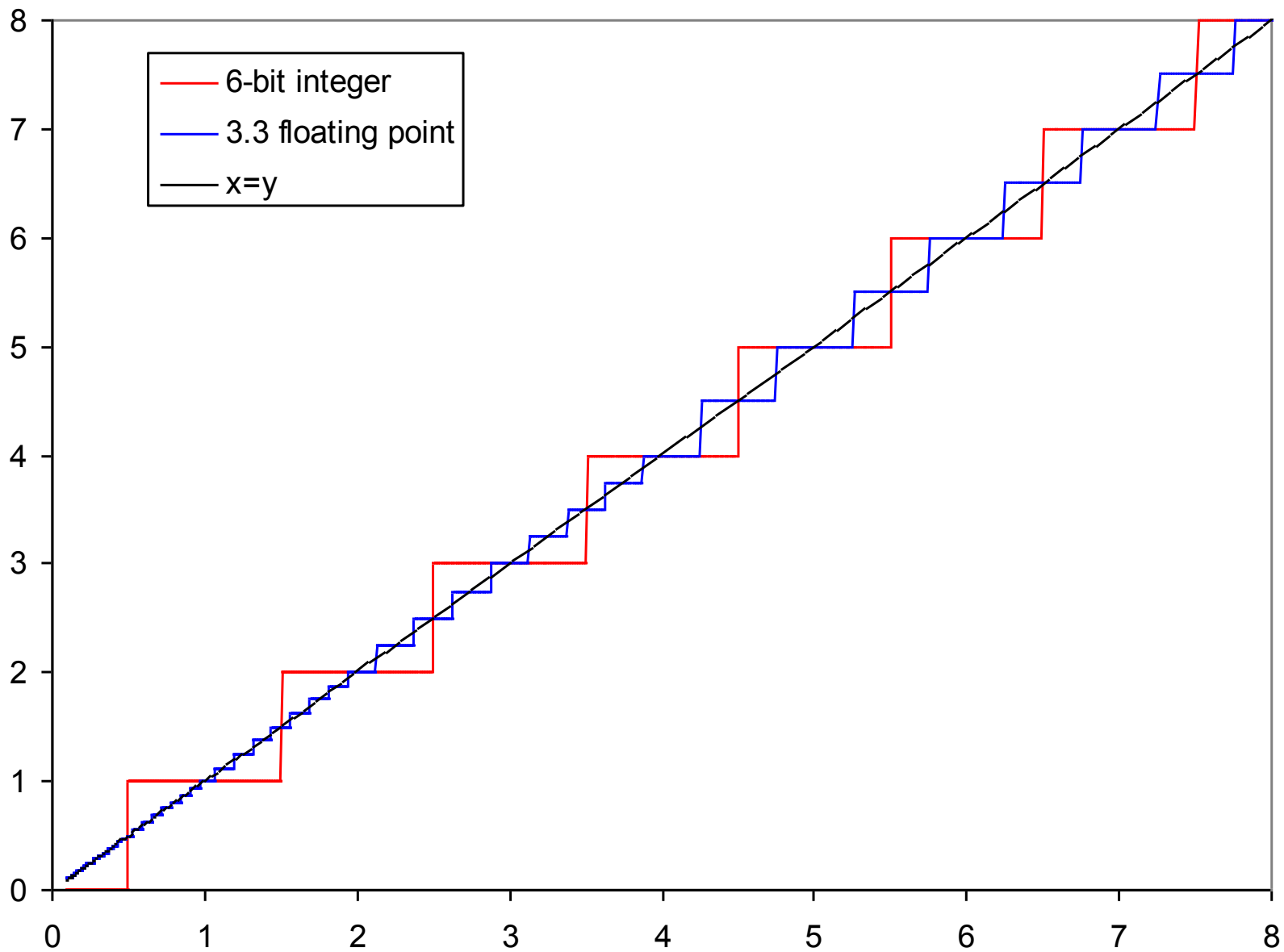
IEEE 754 Floating-Point Representation



$$\text{Floating-Point Format: } (-1)^S \times (1 + M) \times 2^{(E-\text{Bias})}$$

| | | | | 6.4 FP | | | | |
|--------|---|----------|---------|--------|----------|----------|-------------|-------|
| Value | 10-bit integer | Error | % | Exp | Mantissa | Value | Error | % |
| 477.49 | 0111011101 | 0.49 | 0.10% | 1110 | 1110111 | 476 | 1.49 | 0.31% |
| 47.749 | 0000110000 | 0.251 | 0.53% | 1011 | 1011111 | 47.5 | 0.249 | 0.52% |
| 4.7749 | 0000000101 | 0.2251 | 4.71% | 1000 | 1001100 | 4.75 | 0.0249 | 0.52% |
| 0.4775 | 0000000000 | 0.47749 | 100.00% | 0100 | 1111010 | 0.476563 | 0.0009275 | 0.19% |
| 0.0477 | 0000000000 | 0.047749 | 100.00% | 0001 | 1100010 | 0.047852 | 0.000102562 | 0.21% |
| | | | | | | | | |
| | | | | | | | | |
| | Shift mantissa by exp-12 | | | | | | | |
| | Binary point is shifted by exp-5 from left of implied one | | | | | | | |





Summary

- Binary number representation
- Add numbers a bit at a time
- 2's complement $-x = (2^n - x) = (2^n - 1) - x + 1 = \text{neg}(x) + 1$
- Subtract by 2's complement and add
- Multiply – form partial products p_{ij} and sum

- Number representation – accuracy and resolution.
- Fixed and floating point