Chapter 3

Arithmetic for Computers

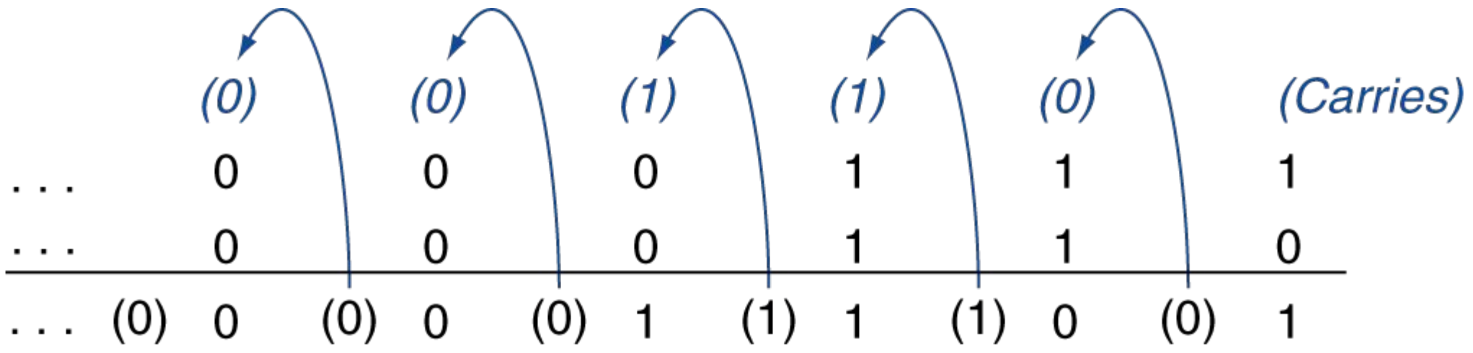
Outline

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Integer Addition

- Example: 7 + 6



- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

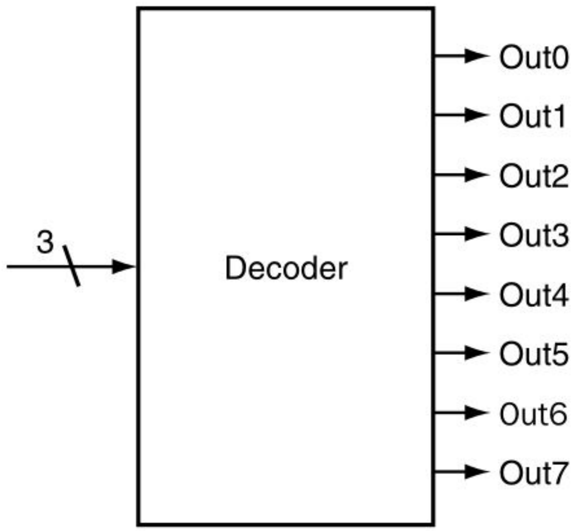
Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

```
+7:  0000 0000 … 0000 0111
–6:  1111 1111 … 1111 1010
+1:  0000 0000 … 0000 0001
```

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

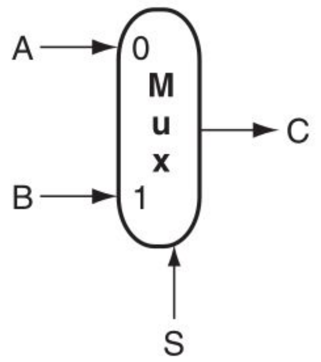
Decoder and Multiplexer

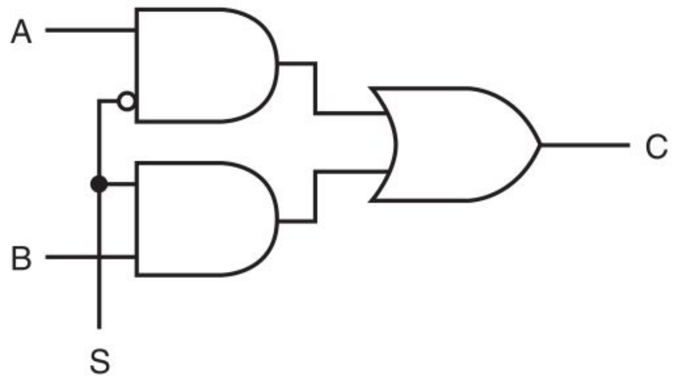


a. A 3-bit decoder

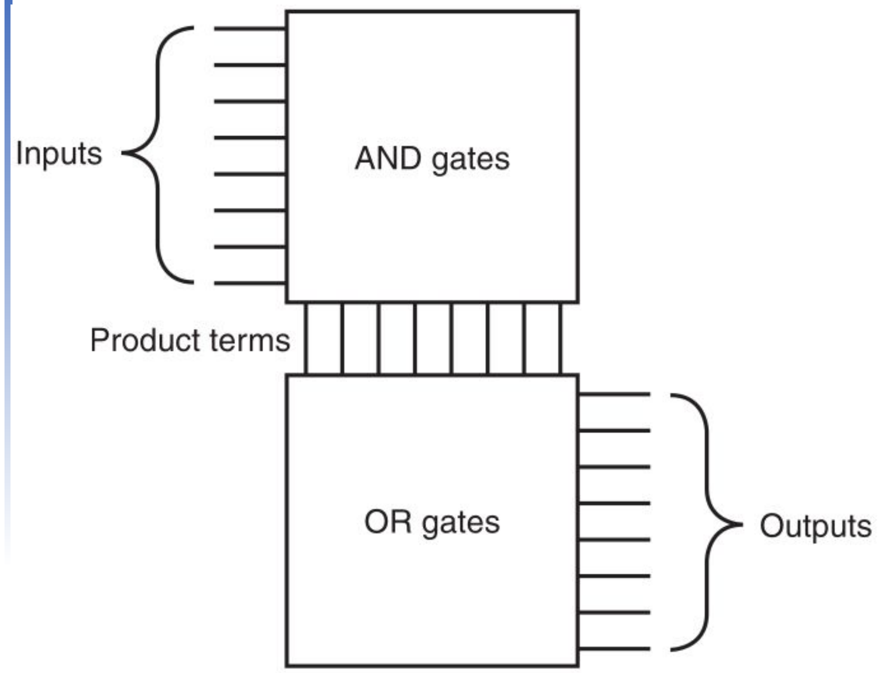
Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

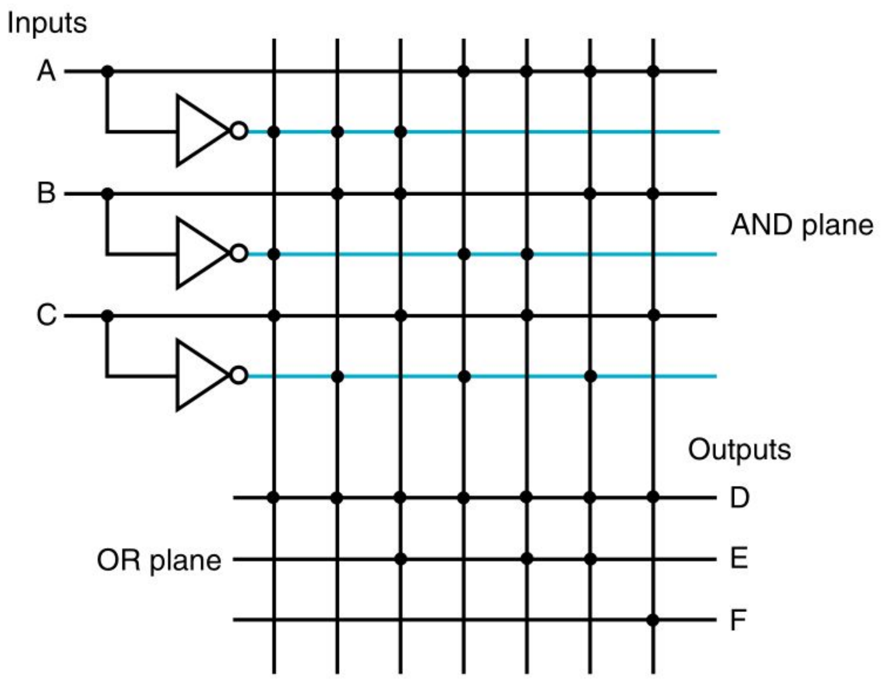
b. The truth table for a 3-bit decoder





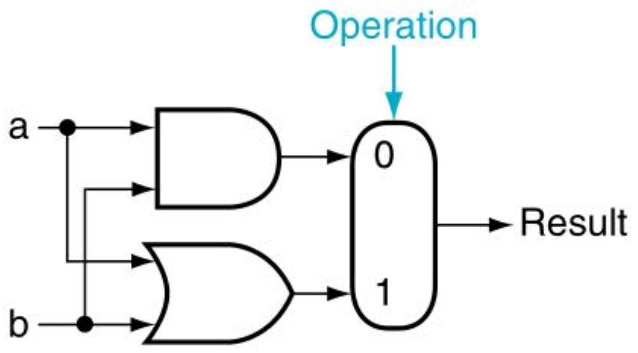
Programmable Logic Array

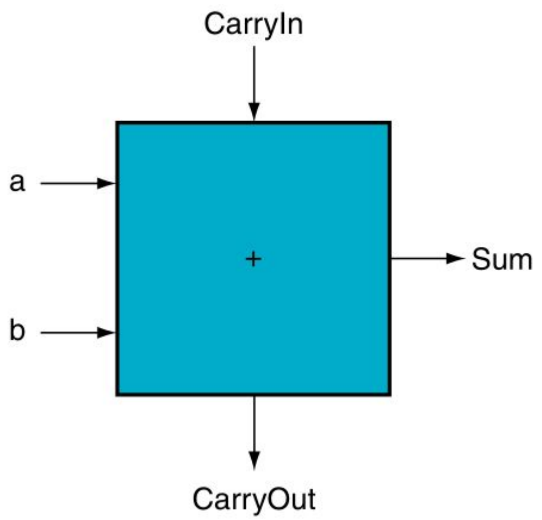


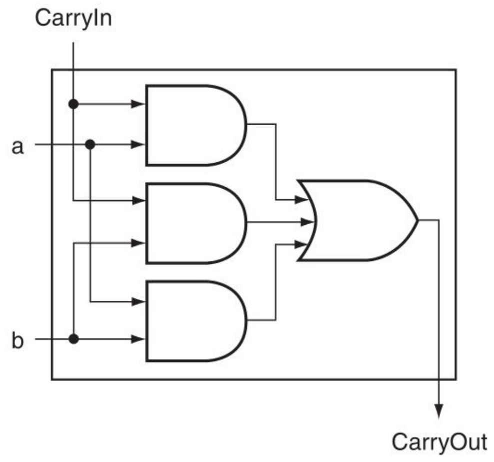


Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Arithmetic Logic Unit (ALU)





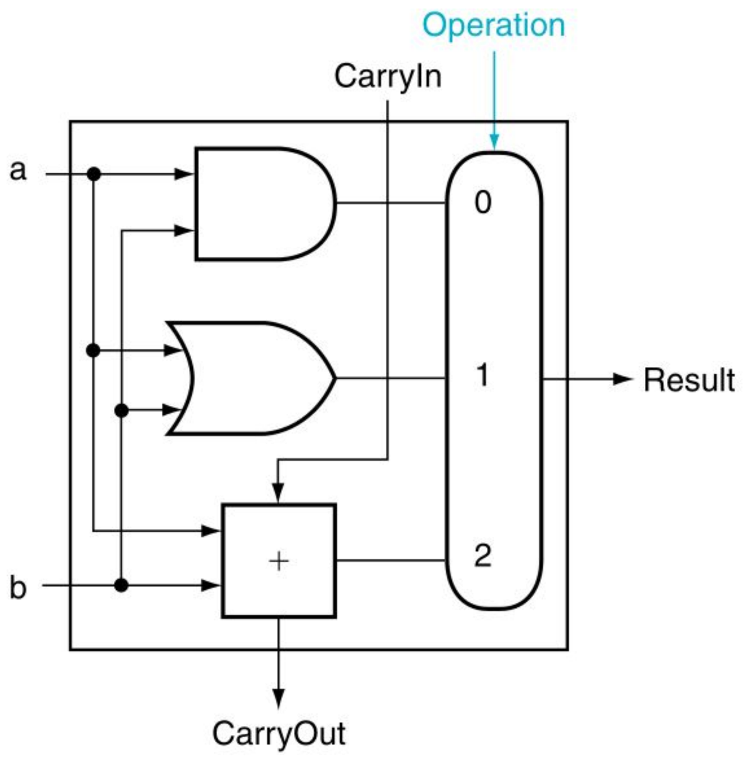


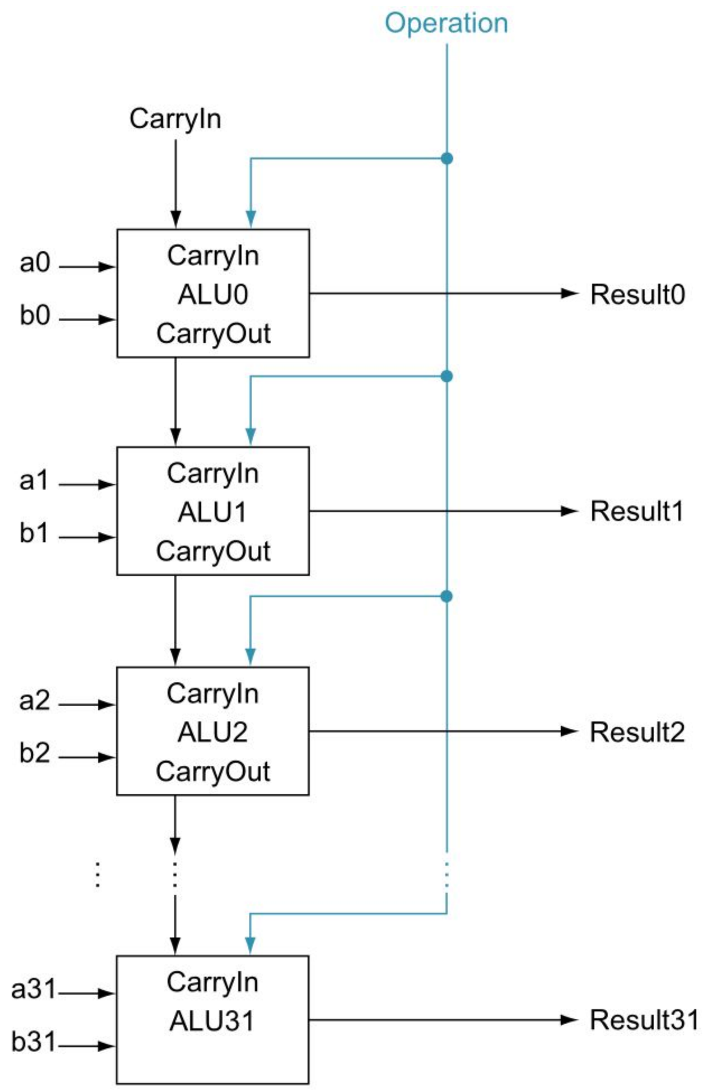
Inputs			Outputs		
a	b	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$

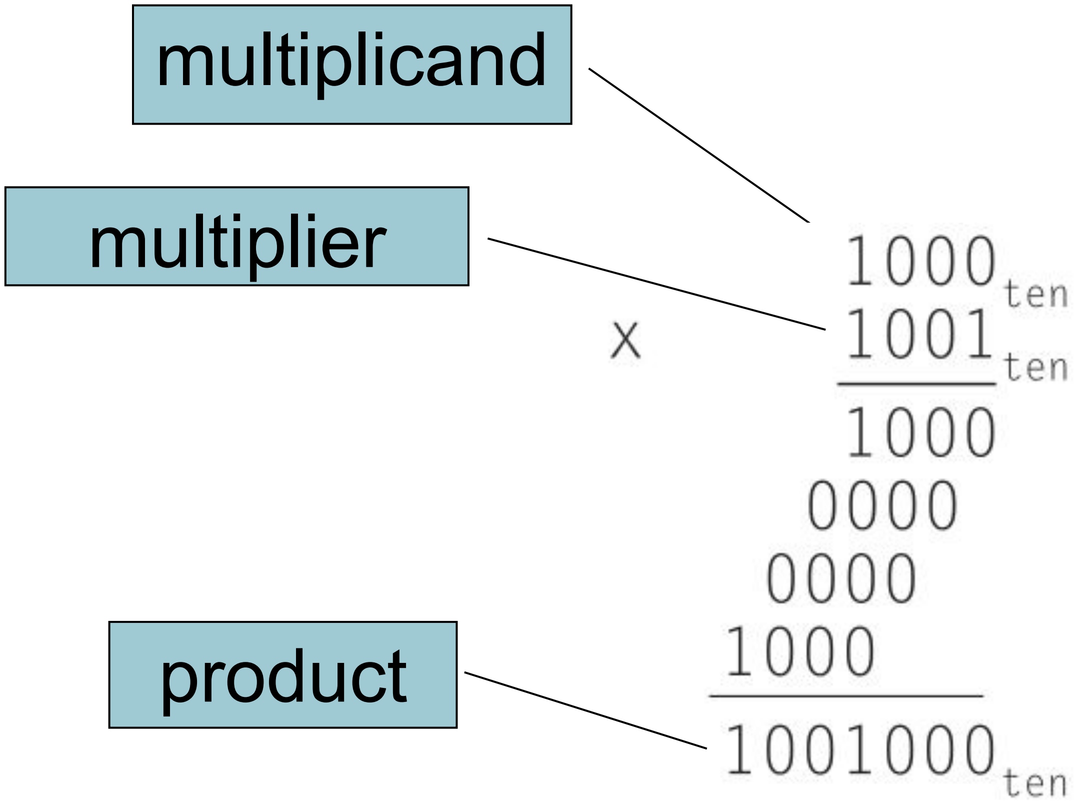
$\text{Sum} = (a \cdot \underline{b} \cdot \underline{\text{CarryIn}}) + (\underline{a} \cdot b \cdot \underline{\text{CarryIn}}) + (\underline{a} \cdot \underline{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$

Arithmetic Logic Unit (ALU)





Multiplication (Example)



Multiplication (Ideas$_1$)

$$\begin{array}{r}
1\,1\,0\,1 \\
\times\ 1\,0\,1\,\mathbf{1} \\
\hline
0\,0\,0\,0 \\
+\ 1\,1\,0\,1 \\
\hdashline
1\,1\,0\,1
\end{array}
\qquad
\begin{array}{r}
1\,1\,0\,1 \\
\times\ 1\,0\,\mathbf{1}\,1 \\
\hline
0\,0\,0\,0 \\
+\ 1\,1\,0\,1 \\
\hdashline
1\,1\,0\,1 \\
+\,1\,1\,0\,1 \leftarrow \\
\hdashline
1\,0\,0\,1\,1\,1
\end{array}
\qquad
\begin{array}{r}
1\,1\,0\,1 \\
\times\ 1\,\mathbf{0}\,1\,1 \\
\hline
0\,0\,0\,0 \\
+\ 1\,1\,0\,1 \\
\hdashline
1\,1\,0\,1 \\
+\,1\,1\,0\,1 \leftarrow \\
\hdashline
1\,0\,0\,1\,1\,1 \\
+\,0\,0\,0\,0 \leftarrow \leftarrow \\
\hdashline
1\,0\,0\,1\,1\,1
\end{array}
\qquad
\begin{array}{r}
1\,1\,0\,1 \\
\times\ \mathbf{1}\,0\,1\,1 \\
\hline
0\,0\,0\,0 \\
+\ 1\,1\,0\,1 \\
\hdashline
1\,1\,0\,1 \\
+\,1\,1\,0\,1 \leftarrow \\
\hdashline
1\,0\,0\,1\,1\,1 \\
+\,0\,0\,0\,0 \leftarrow \leftarrow \\
\hdashline
1\,0\,0\,1\,1\,1 \\
+\,1\,1\,0\,1 \leftarrow \leftarrow \leftarrow \\
\hline
1\,0\,0\,0\,1\,1\,1\,1
\end{array}$$

- Multiplier decides the addition by one bit and the deciding bit moves left.
- Multiplicand always shift left

Multiplication (Ideas$_2$)

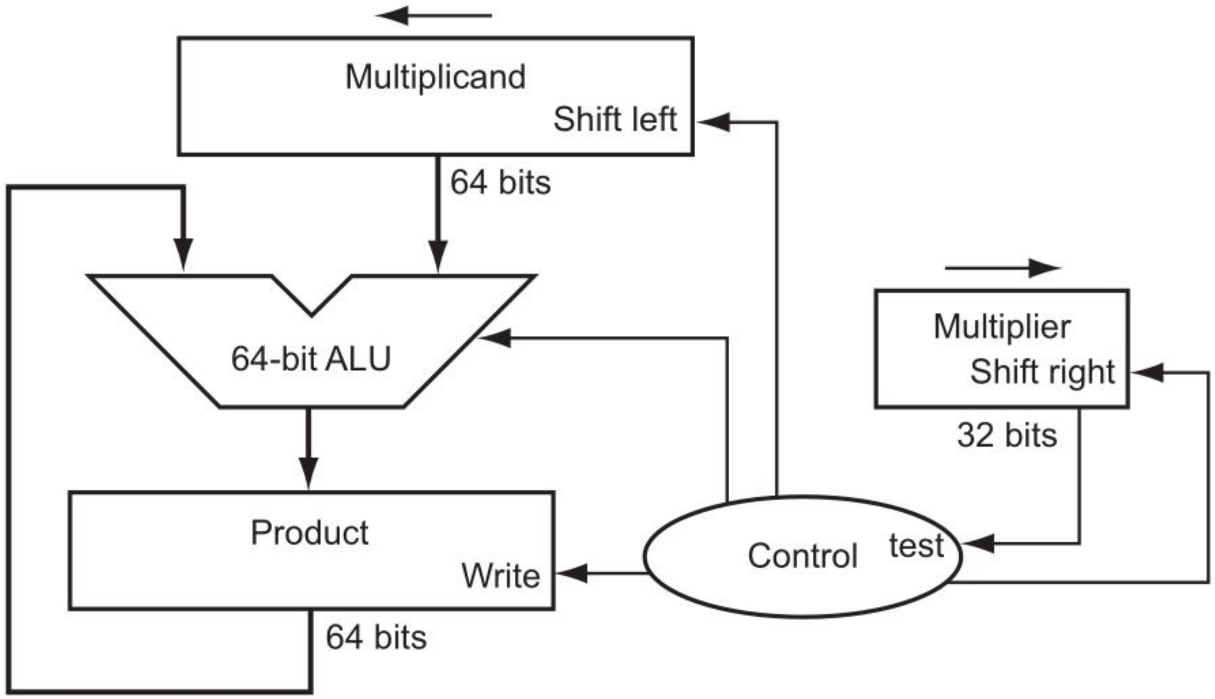
Multiplier	1 0 1 **1**	1 0 **1**	1 **0**	**1**
Multiplicand	1 1 0 1	1 1 0 1 0	1 1 0 1 0 0	1 1 0 1 0 0 0
Product	+ 0 0 0 0	+ 1 1 0 1	1 0 0 1 1 1	+ 1 0 0 1 1 1
	1 1 0 1	1 0 0 1 1 1	→ 1 0 0 1 1 1	1 0 0 0 1 1 1 1

- Look “last bit” of Multiplier
- Shift Multiplier right
- Shift Multiplicand Left

Multiplication

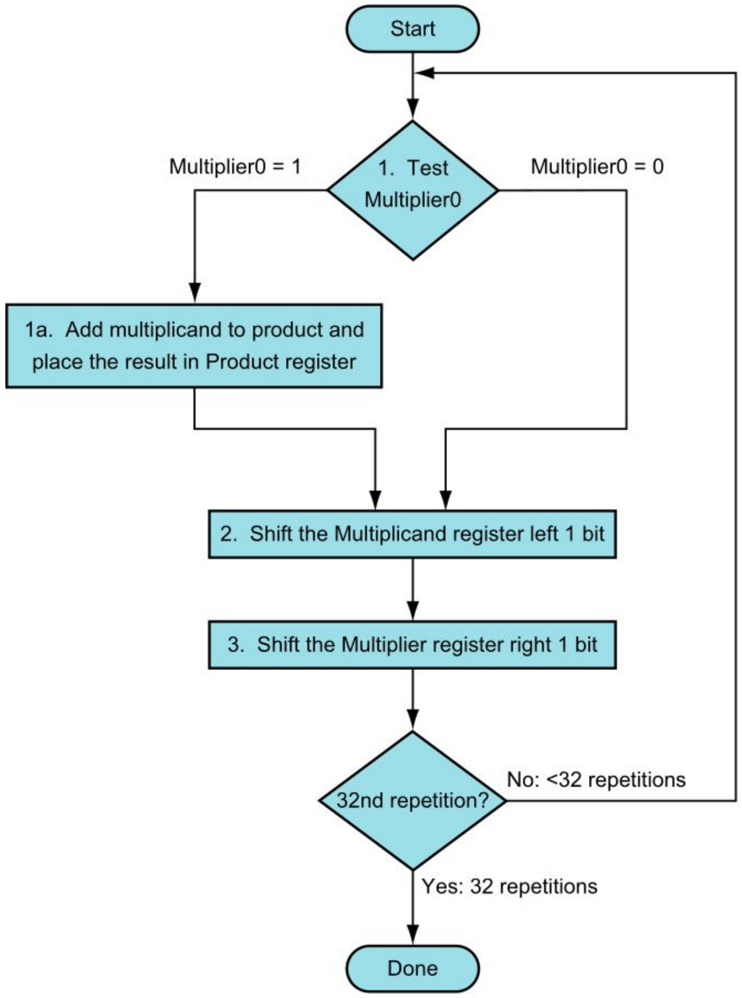
- Start with long-multiplication approach

```
       1011
    .......
       1101
   +   0000
   --------
       1101
        101
    .......
      11010
   +   1101
   --------
     100111
         10
    .......
     110100
     100111
   --------
     100111
          1
    .......
    1101000
   + 100111
   --------
   10001111
```

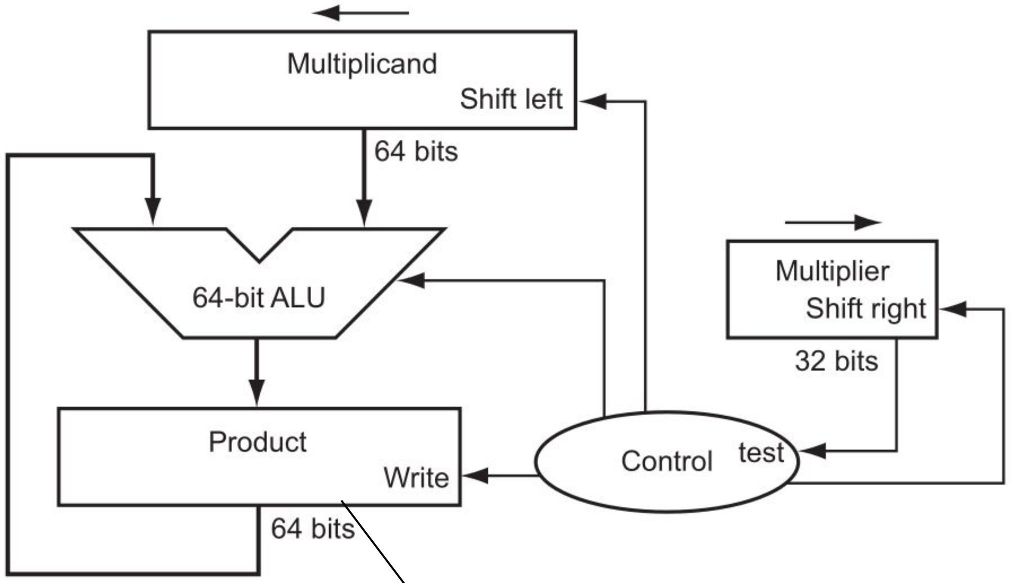


Length of product is the sum of operand lengths

Multiplication Hardware

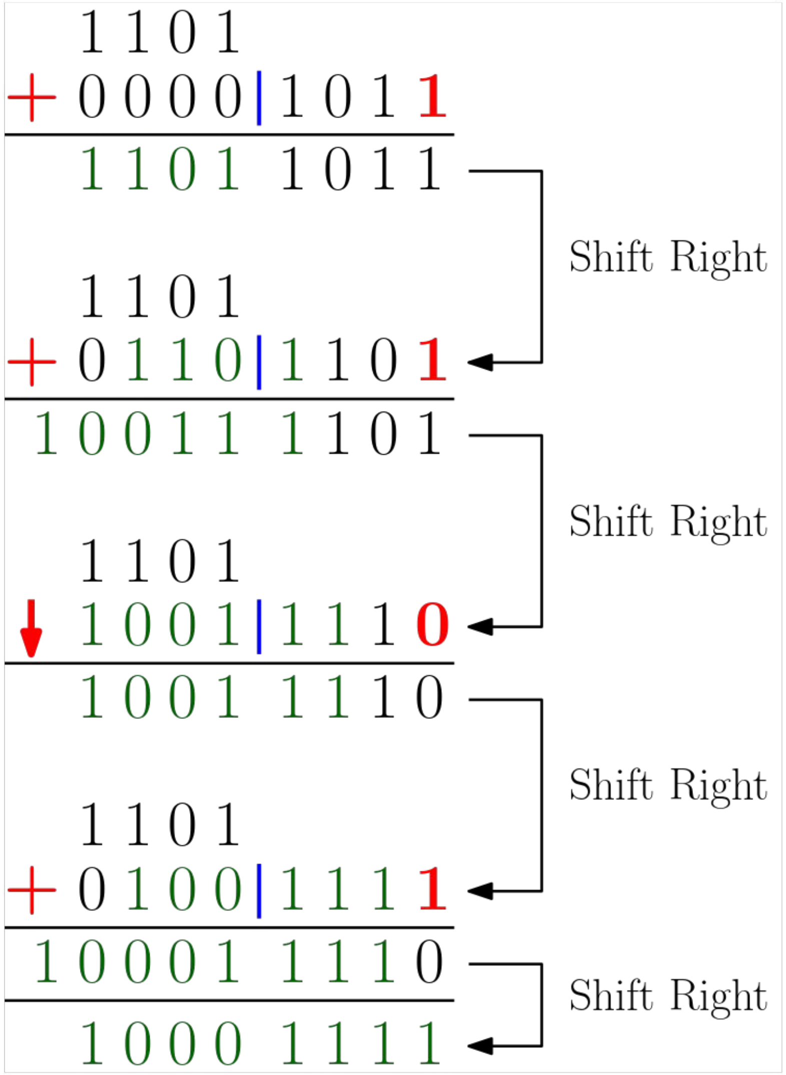


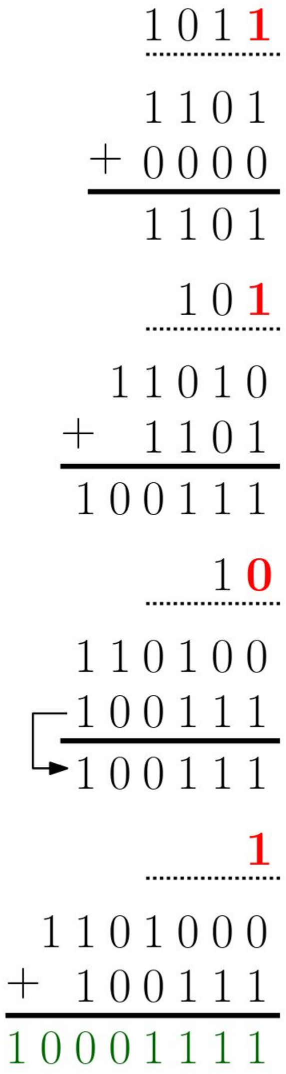
Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: <32 repetitions
Yes: 32 repetitions
Done



Initially 0

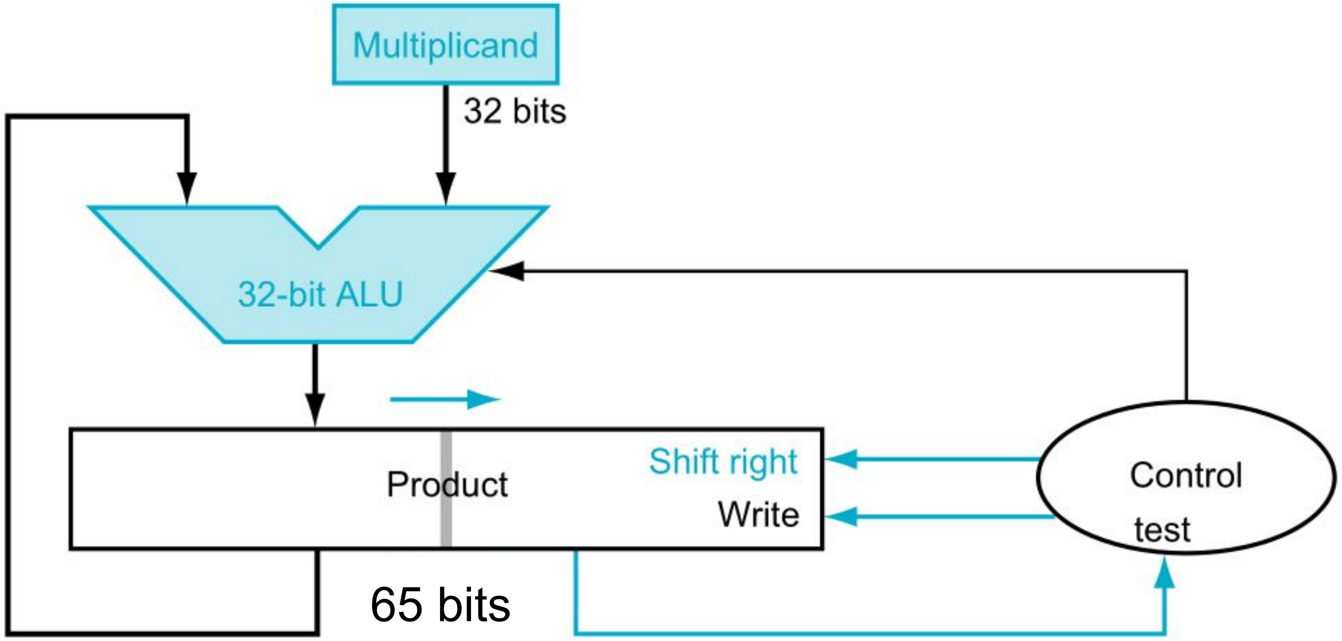
Optimized Multiplier (Ideas)





Optimized Multiplier

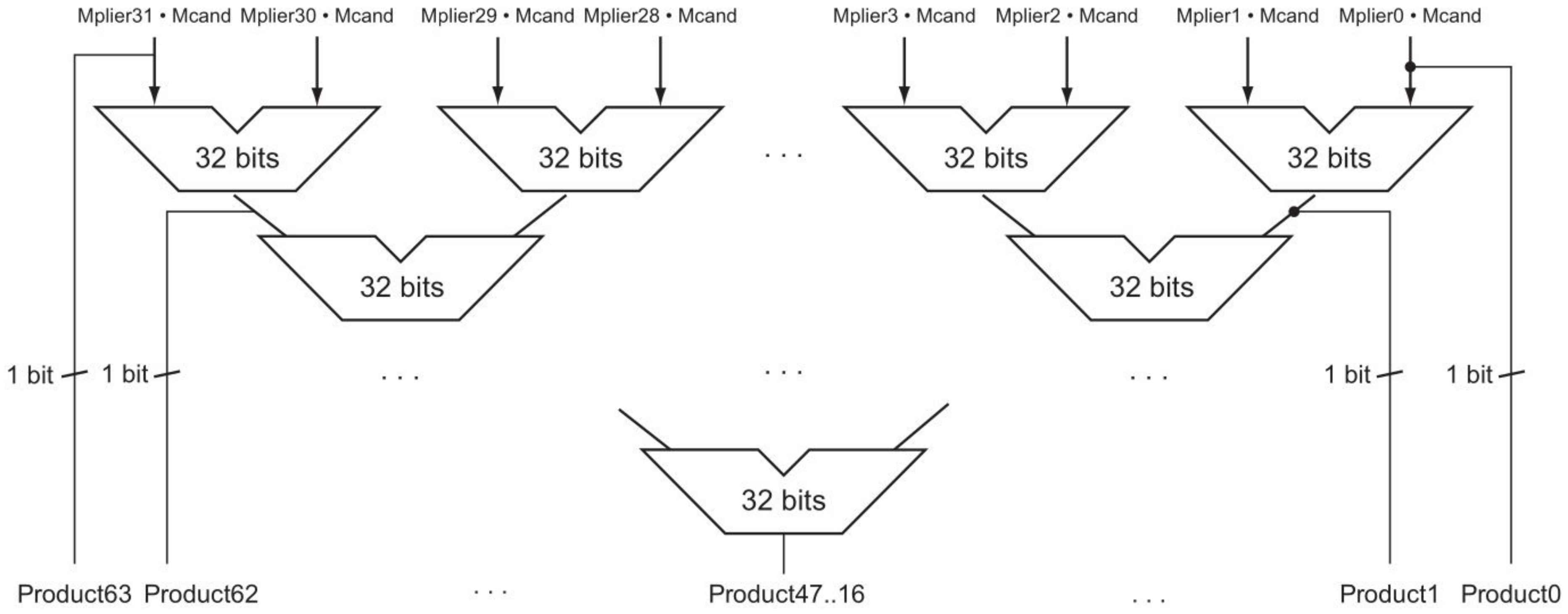
- Perform steps in parallel: add/shift



- One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low

Faster Multiplier

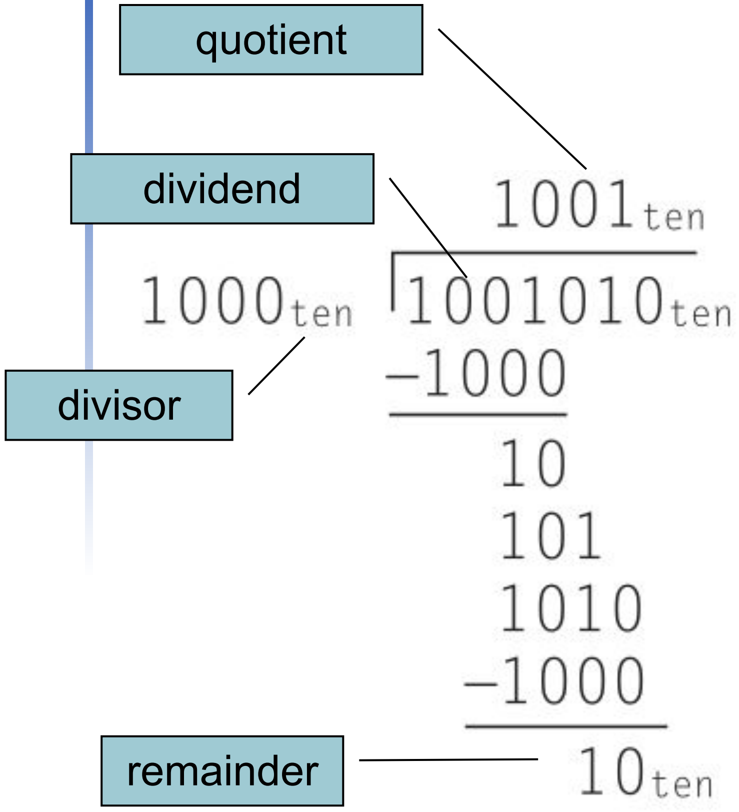
- Uses multiple adders
  - Cost-performance tradeoff



RISC-V Multiplication

- Four multiply instructions:
  - mul: multiply
    - Gives the lower 32 bits of the product
  - mulh: multiply high
    - Gives the upper 32 bits of the product, assuming the operands are signed
  - mulhu: multiply high unsigned
    - Gives the upper 32 bits of the product, assuming the operands are unsigned
  - mulhsu: multiply high signed/unsigned
    - Gives the upper 32 bits of the product, assuming one operand is signed and the other unsigned
  - Use mulh result to check for 32-bit overflow

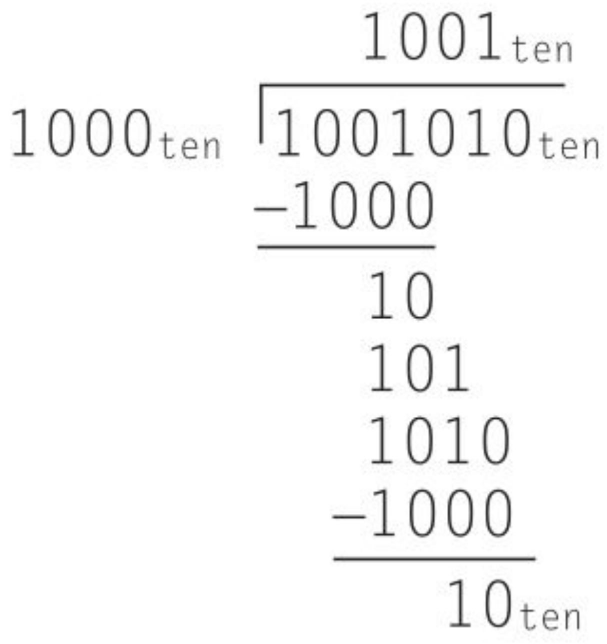
Division

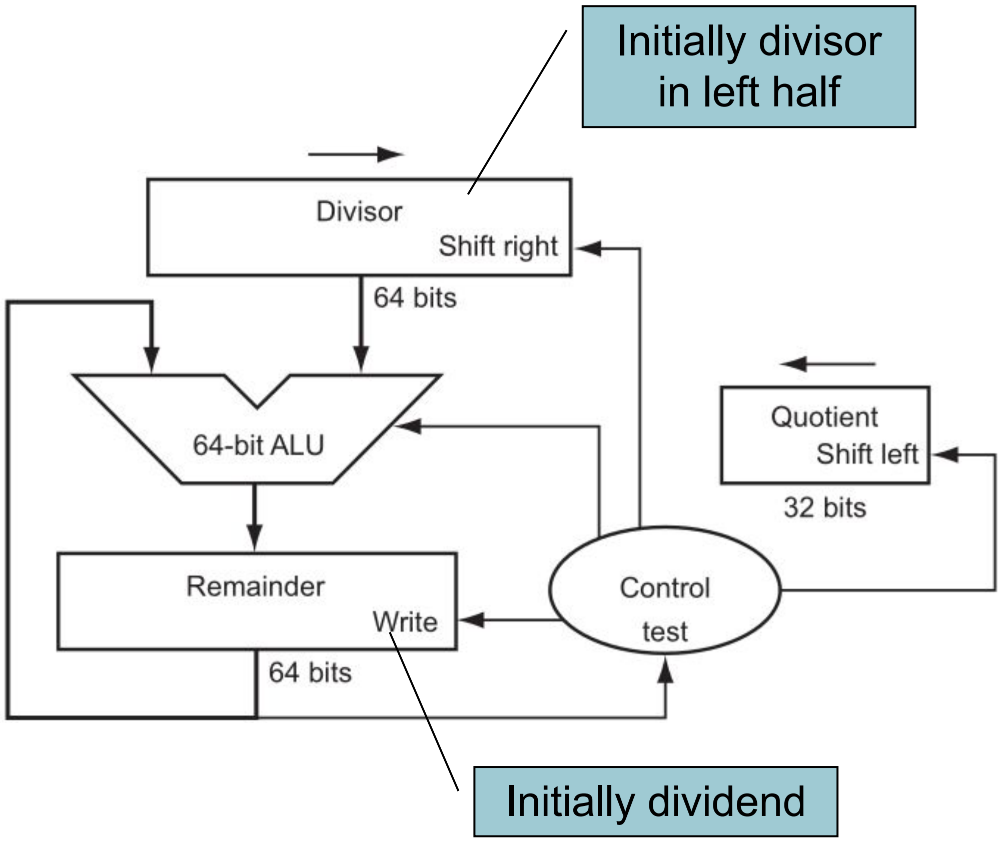


n-bit operands yield *n*-bit quotient and remainder

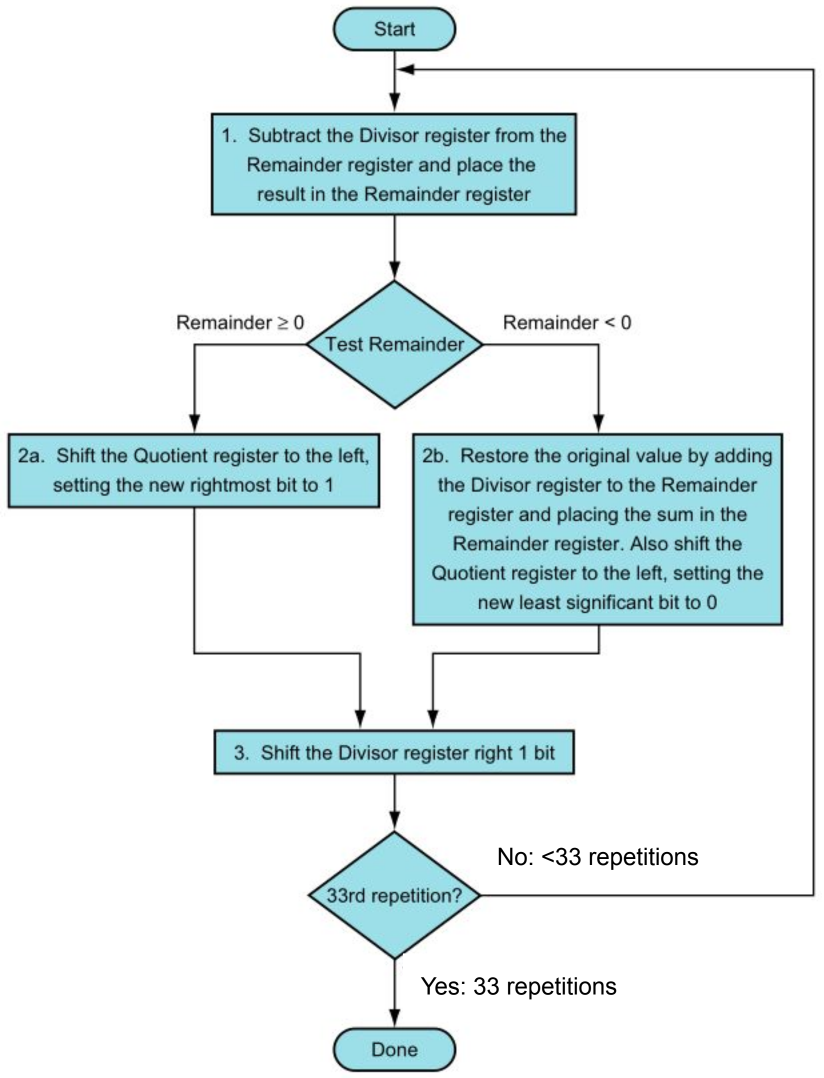
- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

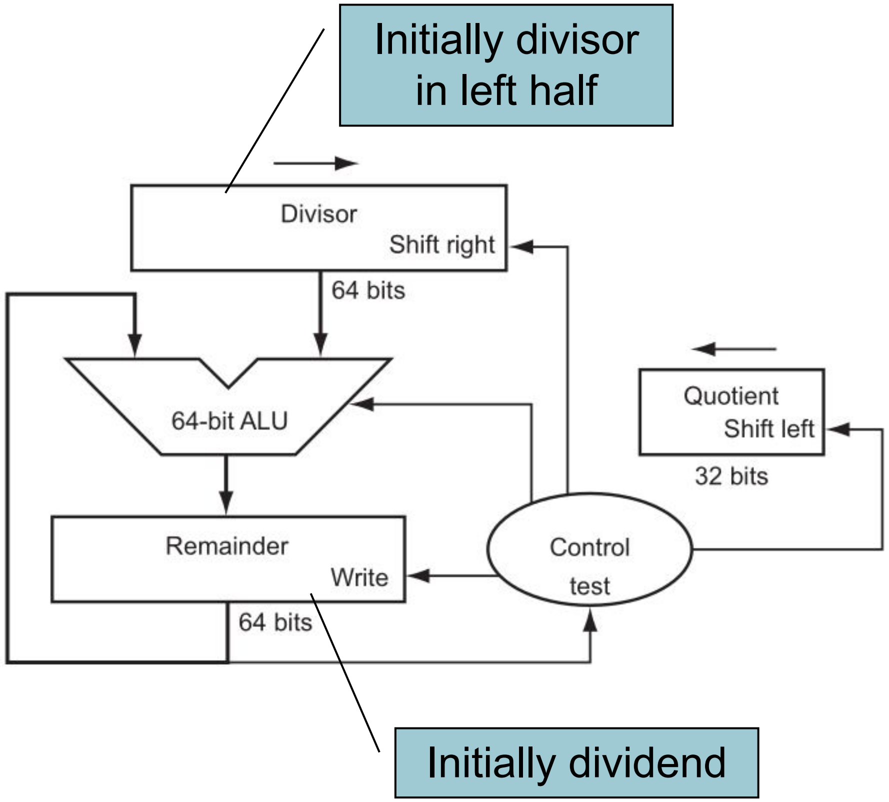
Division Hardware



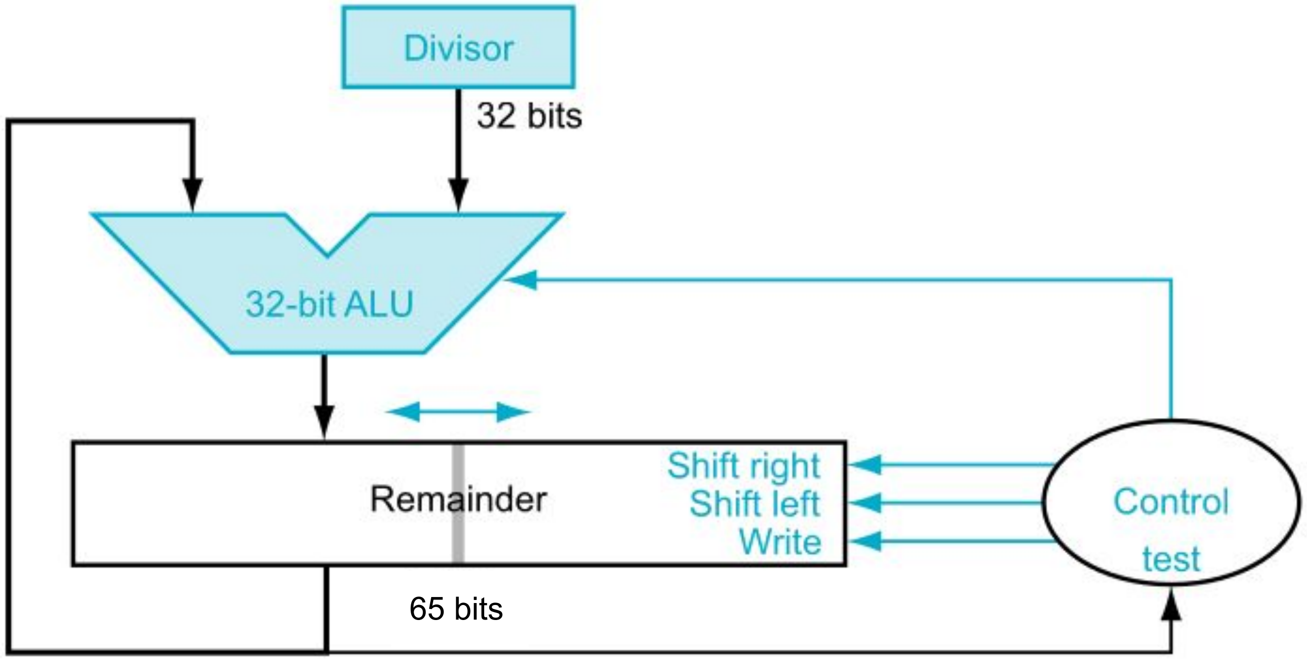


Division Hardware


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: <33 repetitions
Yes: 33 repetitions
Done

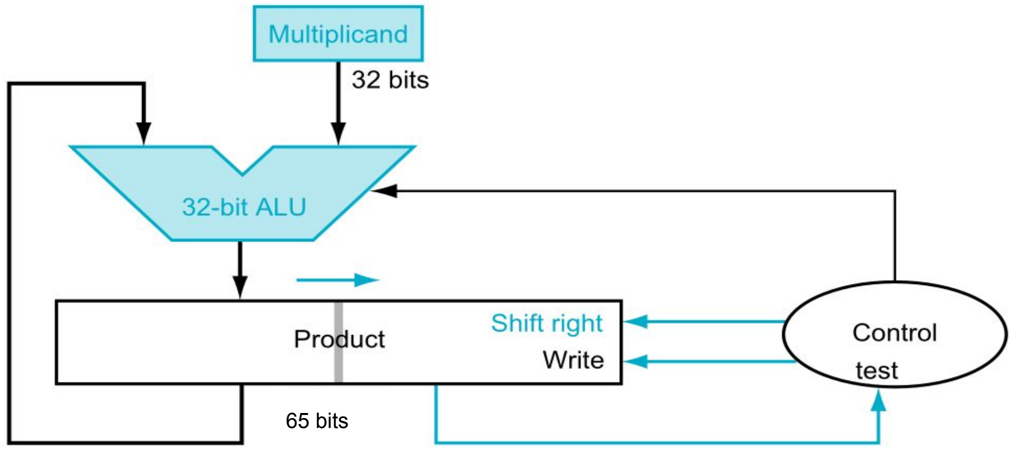


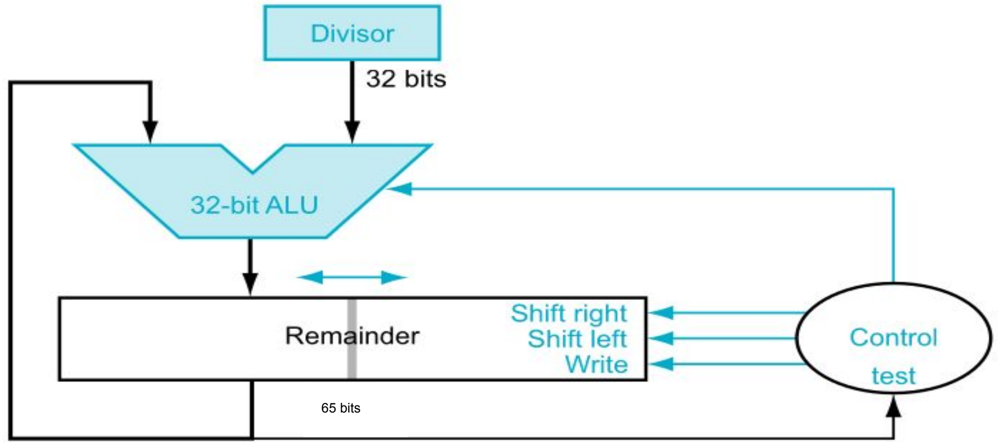
Optimized Divider



- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

Optimized Multiplier & Divider





RISC-V Division

- Four instructions:
  - div, rem: signed divide, remainder
  - divu, remu: unsigned divide, remainder
- Overflow and division-by-zero don’t produce errors
  - Just return defined results
  - Faster for the common case of no error

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$

normalized

not normalized

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
  - $a.bcd = a \times 2^0 + b \times 2^{-1} + c \times 2^{-2} + d \times 2^{-3}$
  - Types `float` and `double` in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the "1." restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Floating-Point Example$_1$

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example$_2$

- What number is represented by the single-precision float

  11000000101000…00
  - S = 1
  - Fraction = $01000…00_2 = .01_2 = 0.25$
  - Exponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + .01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Denormal Numbers

- Exponent = 000...0 ⇒ hidden bit is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Smaller than normal numbers
  - allow for gradual underflow, with diminishing precision
- Denormal with fraction = 000...0

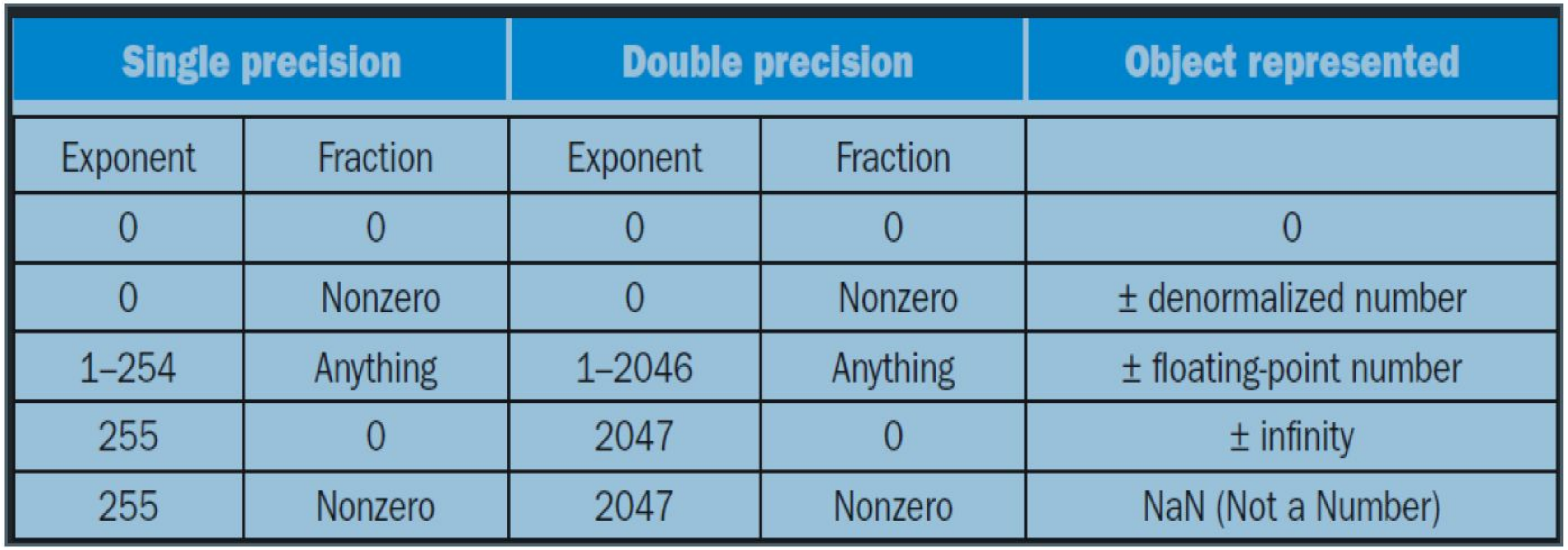
$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

IEEE Floating-Point Encoding



Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- 2. Add significands
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- 3. Normalize result & check for over/underflow
  - 1.0015×10^{2}
- 4. Round and renormalize if necessary
  - 1.002×10^{2}

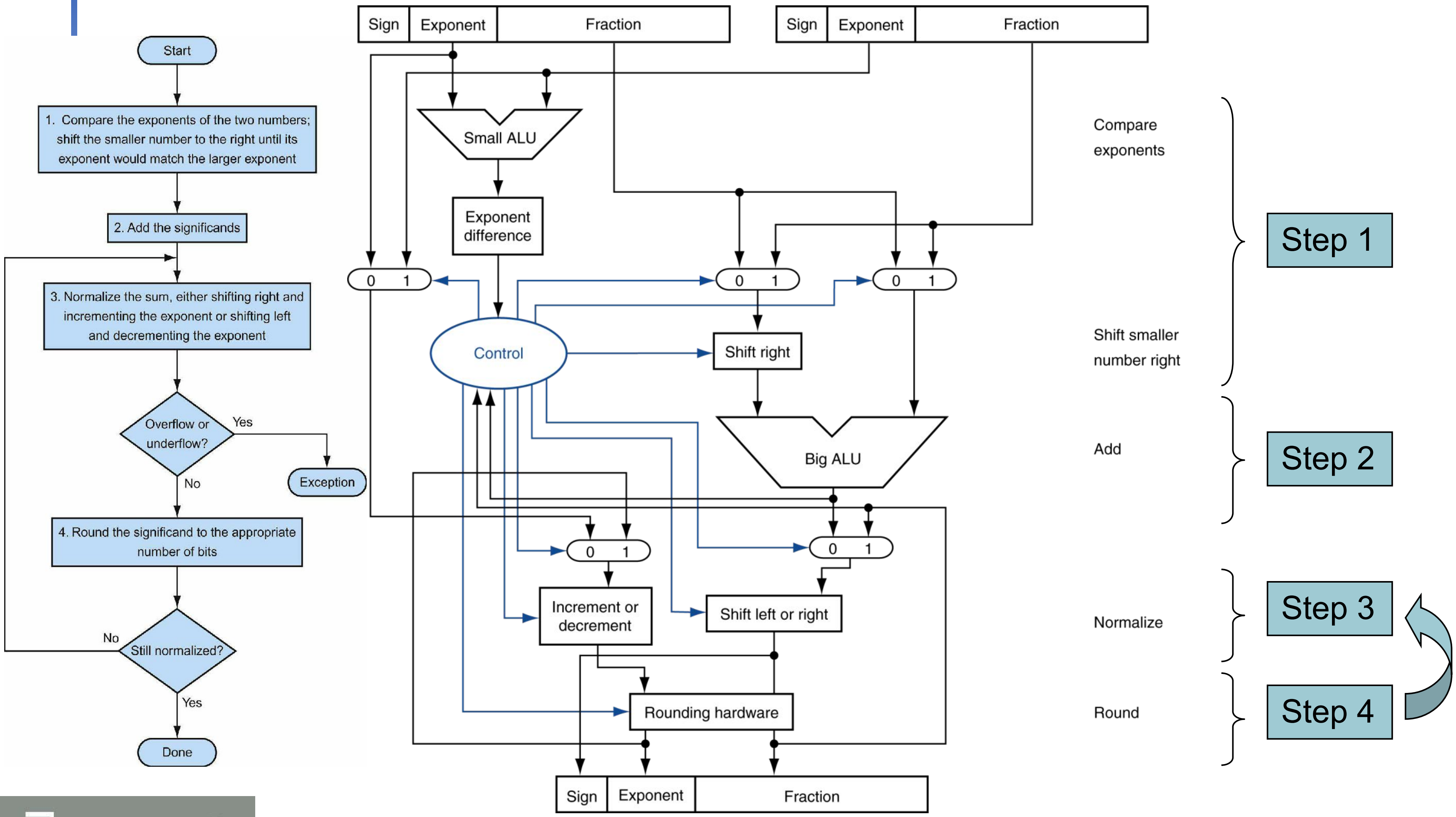
Floating-Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-}1 = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

FP Adder Hardware


Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
No
Exception
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done
Sign
Exponent
Fraction
Sign
Exponent
Fraction
Small ALU
Exponent difference
0
1
Control
Shift right
Big ALU
Increment or decrement
Shift left or right
Rounding hardware
Sign
Exponent
Fraction
Compare exponents
Shift smaller number right
Add
Normalize
Round
Step 1
Step 2
Step 3
Step 4

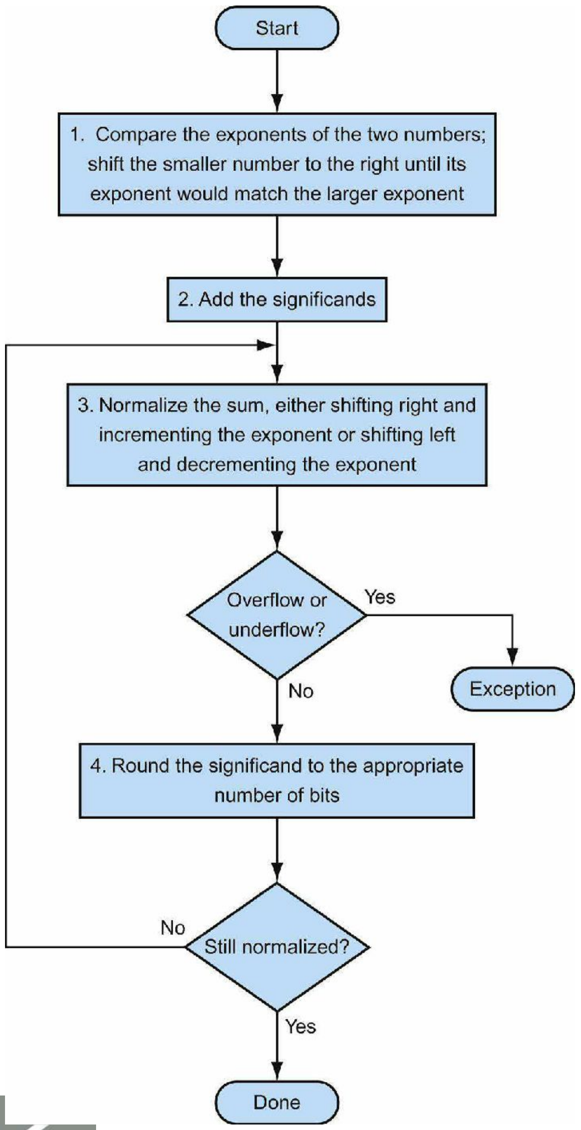
Floating-Point Multiplication

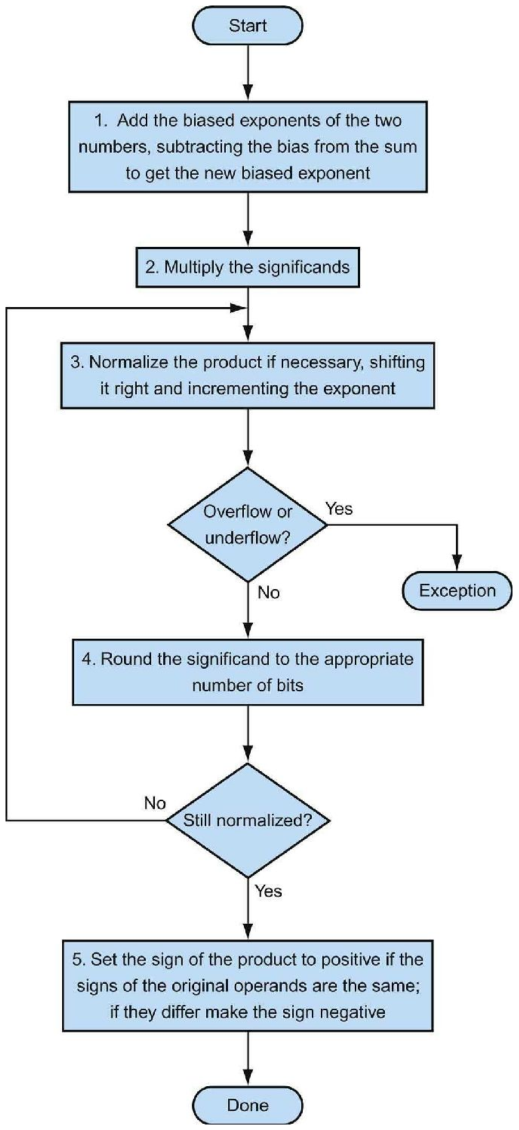
- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = $10 + -5 = 5$
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- 3. Normalize result & check for over/underflow
  - 1.0212×10^{6}
- 4. Round and renormalize if necessary
  - 1.021×10^{6}
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)
- 1. Add exponents
  - Unbiased: –1 + –2 = –3
  - Biased: –3 + 127
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

FP Multiplication & Addition


Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done


Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

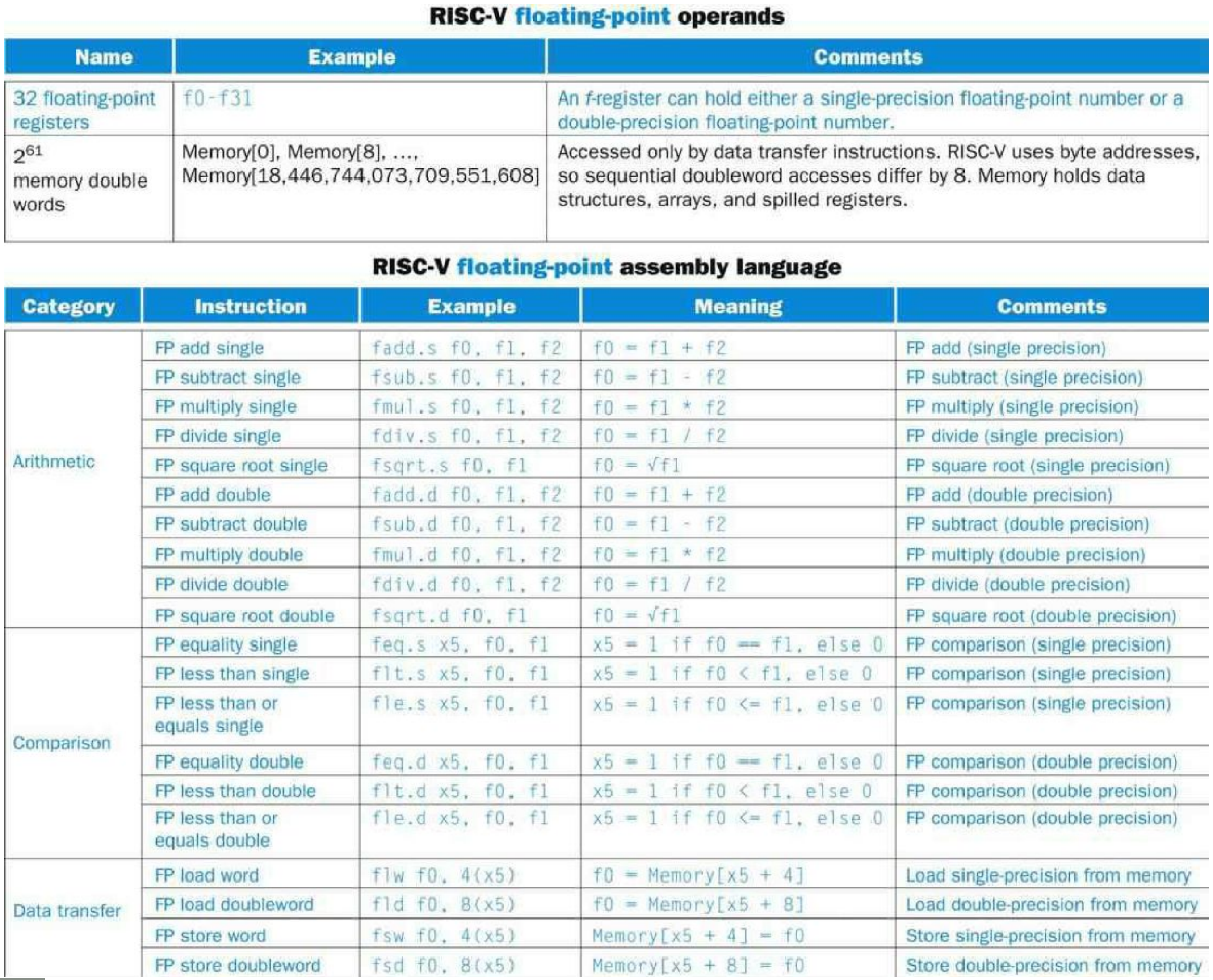
FP Instructions in RISC-V

- Separate FP registers: f0, …, f31
  - double-precision
  - single-precision values stored in the lower 32 bits
- FP instructions operate only on FP registers
  - Programs generally don’t do integer ops on FP data, or vice versa
- FP load and store instructions
  - `flw, fld`
  - `fsw, fsd`

FP Instructions in RISC-V

- Single-precision arithmetic
  - `fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s`
    - e.g., `fadds.s f2, f4, f6`
- Double-precision arithmetic
  - `fadd.d, fsub.d, fmul.d, fdiv.d, fsqrt.d`
    - e.g., `fadd.d f2, f4, f6`
- Single- and double-precision comparison
  - `feq.s, flt.s, fle.s`
  - `feq.d, flt.d, fle.d`
  - Result is 0 or 1 in integer destination register
    - Use beq, bne to branch on comparison result
- Branch on FP condition code true or false
  - `b.cond`

FP Instructions in RISC-V



RISC-V floating-point operands

Name	Example	Comments
32 floating-point registers	f0–f31	An f-register can hold either a single-precision floating-point number or a double-precision floating-point number.
2^{61} memory double words	Memory[0], Memory[8], …, Memory[18,446,744,073,709,551,608]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential doubleword accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

RISC-V floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`fadd.s f0, f1, f2`	`f0 = f1 + f2`	FP add (single precision)
	FP subtract single	`fsub.s f0, f1, f2`	`f0 = f1 - f2`	FP subtract (single precision)
	FP multiply single	`fmul.s f0, f1, f2`	`f0 = f1 * f2`	FP multiply (single precision)
	FP divide single	`fdiv.s f0, f1, f2`	`f0 = f1 / f2`	FP divide (single precision)
	FP square root single	`fsqrt.s f0, f1`	`f0 = √f1`	FP square root (single precision)
	FP add double	`fadd.d f0, f1, f2`	`f0 = f1 + f2`	FP add (double precision)
	FP subtract double	`fsub.d f0, f1, f2`	`f0 = f1 - f2`	FP subtract (double precision)
	FP multiply double	`fmul.d f0, f1, f2`	`f0 = f1 * f2`	FP multiply (double precision)
	FP divide double	`fdiv.d f0, f1, f2`	`f0 = f1 / f2`	FP divide (double precision)
	FP square root double	`fsqrt.d f0, f1`	`f0 = √f1`	FP square root (double precision)
Comparison	FP equality single	`feq.s x5, f0, f1`	`x5 = 1 if f0 == f1, else 0`	FP comparison (single precision)
	FP less than single	`flt.s x5, f0, f1`	`x5 = 1 if f0 < f1, else 0`	FP comparison (single precision)
	FP less than or equals single	`fle.s x5, f0, f1`	`x5 = 1 if f0 <= f1, else 0`	FP comparison (single precision)
	FP equality double	`feq.d x5, f0, f1`	`x5 = 1 if f0 == f1, else 0`	FP comparison (double precision)
	FP less than double	`flt.d x5, f0, f1`	`x5 = 1 if f0 < f1, else 0`	FP comparison (double precision)
	FP less than or equals double	`fle.d x5, f0, f1`	`x5 = 1 if f0 <= f1, else 0`	FP comparison (double precision)
Data transfer	FP load word	`flw f0, 4(x5)`	`f0 = Memory[x5 + 4]`	Load single-precision from memory
	FP load doubleword	`fld f0, 8(x5)`	`f0 = Memory[x5 + 8]`	Load double-precision from memory
	FP store word	`fsw f0, 4(x5)`	`Memory[x5 + 4] = f0`	Store single-precision from memory
	FP store doubleword	`fsd f0, 8(x5)`	`Memory[x5 + 8] = f0`	Store double-precision from memory

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - fahr in f10, result in f10, literals in global memory space

- Compiled RISC-V code:

```
f2c:
  flw    f0,const5(x3)   // f0 = 5.0f
  flw    f1,const9(x3)   // f1 = 9.0f
  fdiv.s f0, f0, f1      // f0 = 5.0f / 9.0f
  flw    f1,const32(x3)  // f1 = 32.0f
  fsub.s f10,f10,f1      // f10 = fahr – 32.0
  fmul.s f10,f0,f10    // f10 = (5.0f/9.0f) * (fahr–32.0f)
  jalr   x0,0(x1)      // return
```

FP Example: Matrix Multiplication$_1$

- C = C + A × B
  - All 32 × 32 matrices, 64-bit double-precision elements
- C code:

```
void mm (double c[][],
         double a[][], double b[][]) {
  size_t i, j, k;
  for (i = 0; i < 32; i = i + 1)
    for (j = 0; j < 32; j = j + 1)
      for (k = 0; k < 32; k = k + 1)
        c[i][j] = c[i][j]
                  + a[i][k] * b[k][j];
}
```

  - Addresses of `c`, `a`, `b` in x10, x11, x12, and `i`, `j`, `k` in x5, x6, x7

FP Example: Matrix Multiplication$_2$

- RISC-V code:

```
mm:...
        li    x28,32        // x28 = 32 (row size/loop end)
        li    x5,0          // i = 0; initialize 1st for loop
    L1: li    x6,0          // j = 0; initialize 2nd for loop
    L2: li    x7,0          // k = 0; initialize 3rd for loop
        slli  x30,x5,5      // x30 = i * 2**5 (size of row of c)
        add   x30,x30,x6    // x30 = i * size(row) + j
        slli  x30,x30,3     // x30 = byte offset of [i][j]
        add   x30,x10,x30   // x30 = byte address of c[i][j]
        fld   f0,0(x30)     // f0 = c[i][j]
    L3: slli  x29,x7,5      // x29 = k * 2**5 (size of row of b)
        add   x29,x29,x6    // x29 = k * size(row) + j
        slli  x29,x29,3     // x29 = byte offset of [k][j]
        add   x29,x12,x29   // x29 = byte address of b[k][j]
        fld   f1,0(x29)     // f1 = b[k][j]
```

FP Example: Array Multiplication

…

```
slli   x29,x5,5      // x29 = i * 2**5 (size of row of a)
add    x29,x29,x7    // x29 = i * size(row) + k
slli   x29,x29,3     // x29 = byte offset of [i][k]
add    x29,x11,x29   // x29 = byte address of a[i][k]
fld    f2,0(x29)     // f2 = a[i][k]
fmul.d f1, f2, f1    // f1 = a[i][k] * b[k][j]
fadd.d f0, f0, f1    // f0 = c[i][j] + a[i][k] * b[k][j]
addi   x7,x7,1       // k = k + 1
bltu   x7,x28,L3     // if (k < 32) go to L3
fsd    f0,0(x30)     // c[i][j] = f0
addi   x6,x6,1       // j = j + 1
bltu   x6,x28,L2     // if (j < 32) go to L2
addi   x5,x5,1       // i = i + 1
bltu   x5,x28,L1     // if (i < 32) go to L1
```

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

IEEE 754 Rounding Mode

- A 1-digit decimal example

	$1.40	$1.60	$1.50	$2.50	$-1.50
Round up	$2	$2	$2	$3	-$1
Round down	$1	$1	$1	$2	-$2
Truncate	$1	$1	$1	$2	-$1
Nearest even	$1	$2	$2	$2	-$2

Rounding to Nearest Even

- A 3-digit decimal example

Value_{10}	Rounded_{10}	Action
7.8949999	7.89	Less than 1/2
7.8950001	7.90	Greater than 1/2
7.8950000	7.90	1/2 Round up
7.8850000	7.88	1/2 Round down

- A 4-digit binary example

Value_{2}	Rounded_{2}	Action
10.00001	10.00	Less than 1/2
10.00110	10.01	Greater than 1/2
10.11100	11.00	1/2 round up
10.10100	10.10	1/2 round down

Example

- Use guard bit, round bit, and sticky bit
- Consider the following example

```
    S    E           F
    1   10000000   1.11000000000000000011111
  + 1   10000010   1.11100000000000000001001
  -------------------------------------------
```

- Shift the smaller to line up exponents, add significands, and normalize

```
    S    E           F                      grs
    1   10000010   0.01110000000000000000111110
  + 1   10000010   1.11100000000000000001001000
  ----------------------------------------------
    1   10000010  10.01010000000000000010000110 (add significands)
  ----------------------------------------------
    1   10000011   1.00101000000000000001000011 (normalize)
```

Example

```
  S    E          F                           grs
  1   10000010   0.011100000000000000000111110
+ 1   10000010   1.111000000000000000001001000
----------------------------------------------
  1   10000010  10.010100000000000000010000110 (add significands)
----------------------------------------------
  1   10000011   1.001010000000000000001000011 (normalize)
```

- Now let’s round the number in these modes

```
  1   10000011   1.001010000000000000001000011 (from the above)
----------------------------------------------
  1   10000011   1.00101000000000000001000    (round up)
  1   10000011   1.00101000000000000001001    (round down)
  1   10000011   1.00101000000000000001000    (truncate)
  1   10000011   1.00101000000000000001000    (nearest even)
```

Subword Parallellism

- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
  - Example: 128-bit adder:
    - Sixteen 8-bit adds
    - Eight 16-bit adds
    - Four 32-bit adds
- Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD)

x86 FP Architecture

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …
- FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
  - Result: poor FP performance

x86 FP Instructions

Data transfer	Arithmetic	Compare	Transcendental

`FILD mem/ST(i)`
`FISTP mem/ST(i)`
`FLDPI`
`FLD1`
`FLDZ`
	`FIADDP mem/ST(i)`
	`FISUBRP mem/ST(i)`
	`FIMULP mem/ST(i)`
	`FIDIVRP mem/ST(i)`
	`FSQRT`
	`FABS`
	`FRNDINT`
		`FICOMP`
		`FIUCOMP`
		`FSTSW AX/mem`
			`FPATAN`
			`F2XMI`
			`FCOS`
			`FPTAN`
			`FPREM`
			`FPSIN`
			`FYL2X`

- Optional variations
  - I: integer operand
  - P: pop operand from stack
  - R: reverse operand order
  - But not all combinations allowed

Streaming SIMD Extension 2 (SSE2)

- Adds 4 × 128-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 × 64-bit double precision
  - 4 × 32-bit double precision
  - Instructions operate on them simultaneously
    - <u>S</u>ingle-<u>I</u>nstruction <u>M</u>ultiple-<u>D</u>ata

Matrix Multiply

- Unoptimized code:

```
void dgemm (int n, double* A, double* B, double* C)
{
 for (int i = 0; i < n; ++i)
   for (int j = 0; j < n; ++j)
   {
    double cij = C[i+j*n]; /* cij = C[i][j] */
    for(int k = 0; k < n; k++ )
     cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
    C[i+j*n] = cij; /* C[i][j] = cij */
  }
}
```

Matrix Multiply

- x86 assembly code:

```
vmovsd (%r10),%xmm0   # Load 1 element of C into %xmm0
mov %rsi,%rcx         # register %rcx = %rsi
xor %eax,%eax         # register %eax = 0
vmovsd (%rcx),%xmm1   # Load 1 element of B into %xmm1
add %r9,%rcx          # register %rcx = %rcx + %r9
vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1, element of A
add $0x1,%rax         # register %rax = %rax + 1
cmp %eax,%edi         # compare %eax to %edi
vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
jg 30 <dgemm+0x30>    # jump if %eax > %edi
add $0x1,%r11d        # register %r11 = %r11 + 1
vmovsd %xmm0,(%r10)   # Store %xmm0 into C element
```

Matrix Multiply

- Optimized C code:

```
#include <x86intrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
    c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
             _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
             _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
  }
 }
```

Matrix Multiply

- Optimized x86 assembly code:

```
vmovapd (%r11),%ymm0        # Load 4 elements of C into %ymm0
mov %rbx,%rcx               # register %rcx = %rbx
xor %eax,%eax               # register %eax = 0
vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
add $0x8,%rax               # register %rax = %rax + 8
vmulpd (%rcx),%ymm1,%ymm1   # Parallel mul %ymm1,4 A elements
add %r9,%rcx                # register %rcx = %rcx + %r9
cmp %r10,%rax               # compare %r10 to %rax
vaddpd %ymm1,%ymm0,%ymm0    # Parallel add %ymm1, %ymm0
jne 50 <dgemm+0x50>         # jump if not %r10 != %rax
add $0x1,%esi               # register % esi = % esi + 1
vmovapd %ymm0,(%r11)        # Store %ymm0 into 4 C elements
```

Right Shift and Division

- Left shift by *i* places multiplies an integer by 2^i
- Right shift divides by 2^i?
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., $11111011_2 = -5$
    - $11111011_2 >> 2 = 11111110_2 = -2$
  - cf. $11111011_2 >>> 2 = 00111110_2 = +62$

Associativity

- Associativity may fail

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Parallel programs may interleave operations in unexpected orders
- Need to validate parallel programs under varying degrees of parallelism

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - “My bank balance is out by 0.0002¢!” ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

Concluding Remarks

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow

Concluding Remarks

RISC-V Instruction	Name	Frequency	Cumulative
Add immediate	addi	14.36%	14.36%
Load word	lw	12.65%	27.01%
Add registers	add	7.57%	34.58%
Load fl. pt. double	fld	6.83%	41.41%
Store word	sw	5.81%	47.22%
Branch if not equal	bne	4.14%	51.36%
Shift left immediate	slli	3.65%	55.01%
Fused mul-add double	fmadd.d	3.49%	58.50%
Branch if equal	beq	3.27%	61.77%
Add immediate word	addiw	2.86%	64.63%
Store fl. pt. double	fsd	2.24%	66.87%
Multiply fl. pt. double	fmul.d	2.02%	68.89%