Computer Architecture

COMP SCI 2GA3

Chapter 3 - Arithmetic for Computers

Based on: RISC-V Chapter 3 textbook slides
COMPSCI 2GA3 2016 fall - Chapter 3
SOFTENG 2GA3 2020 winter - Chapter 3

Dr. Bojan Nokovic, P.Eng.
McMaster University, Fall Term 2020/21

Arithmetic for Computers

- In this chapter we will investigate how integer arithmetic operations are carried out

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow

- Floating-point (real) numbers
  - Representation and operations

Integer Addition

- Example: 7 + 6

		(0)		*(0)*		*(1)*		*(1)*		*(0)*		*(Carries)*
. . .		0		0		0		1		1		1
. . .		0		0		0		1		1		0
. . .	(0)	0	(0)	0	(0)	1	(1)	1	(1)	0	(0)	1

- 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
```

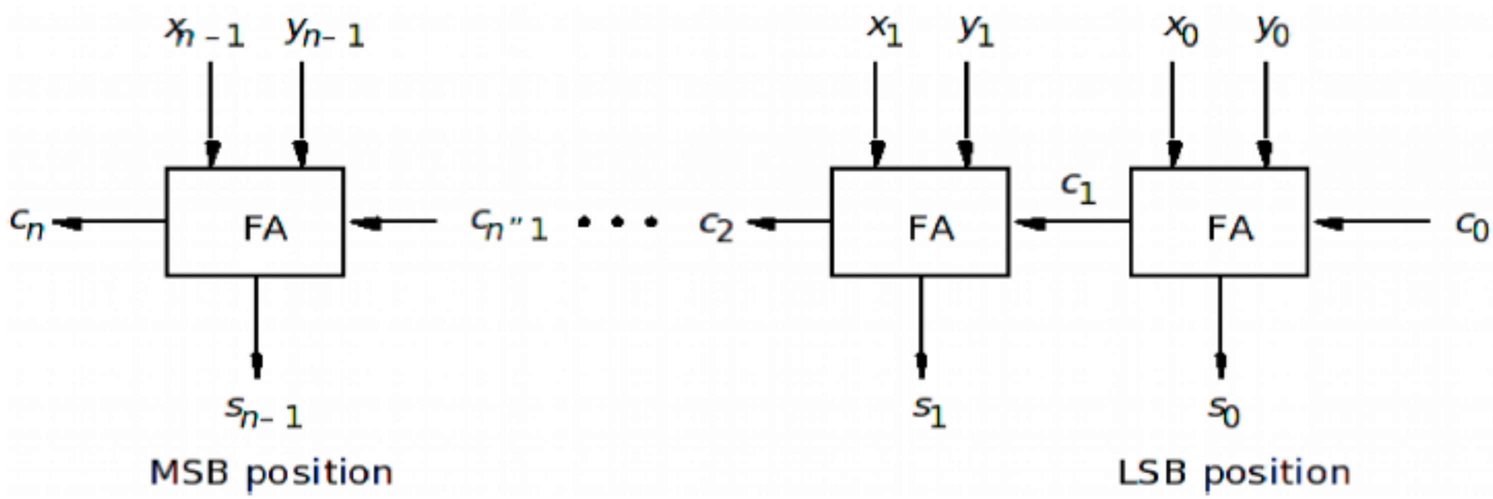
- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

Arithmetic for Multimedia

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data) - a class of parallel computation

- Saturating operations
  - On overflow, result is largest representable value
    - c.f. 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video

n- Bit Ripple Carry Adder



For each stage (called a full adder), we are adding $x_i+y_i+c_i$ to get s_i and c_{i+1}

Normally the first carry in (c_0) is set to zero

- could also be connected to the carry out (c_n) from another n-bit adder, to form a $2_{n\text{-bit}}$ adder

Full Adder

A full adder takes as input a carry from the previous stage (c_i) and a bit from each n-bit number being added (x_i and y_i).

It produces a sum bit (s_i) and a carry out to next stage (c_{i+1})

c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) Truth table

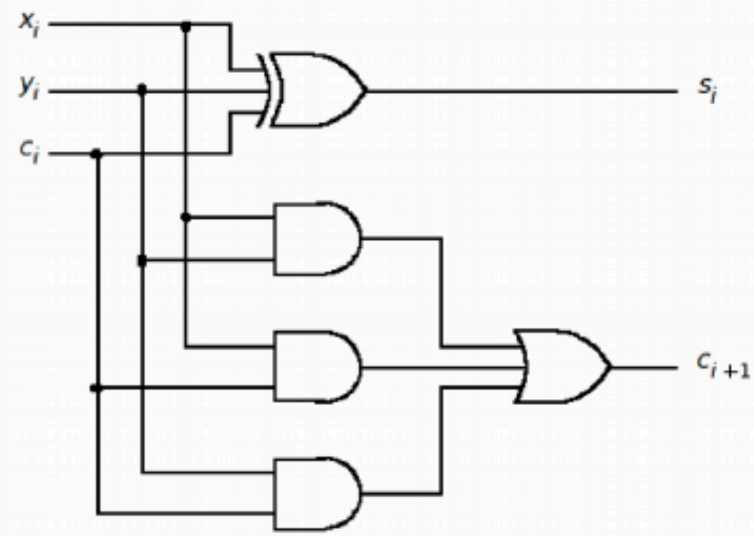
c_i \ x_iy_i	00	01	11	10
0		1		1
1	1		1	

$$s_i = x_i \oplus y_i \oplus c_i$$

c_i \ x_iy_i	00	01	11	10
0			1	
1		1	1	1

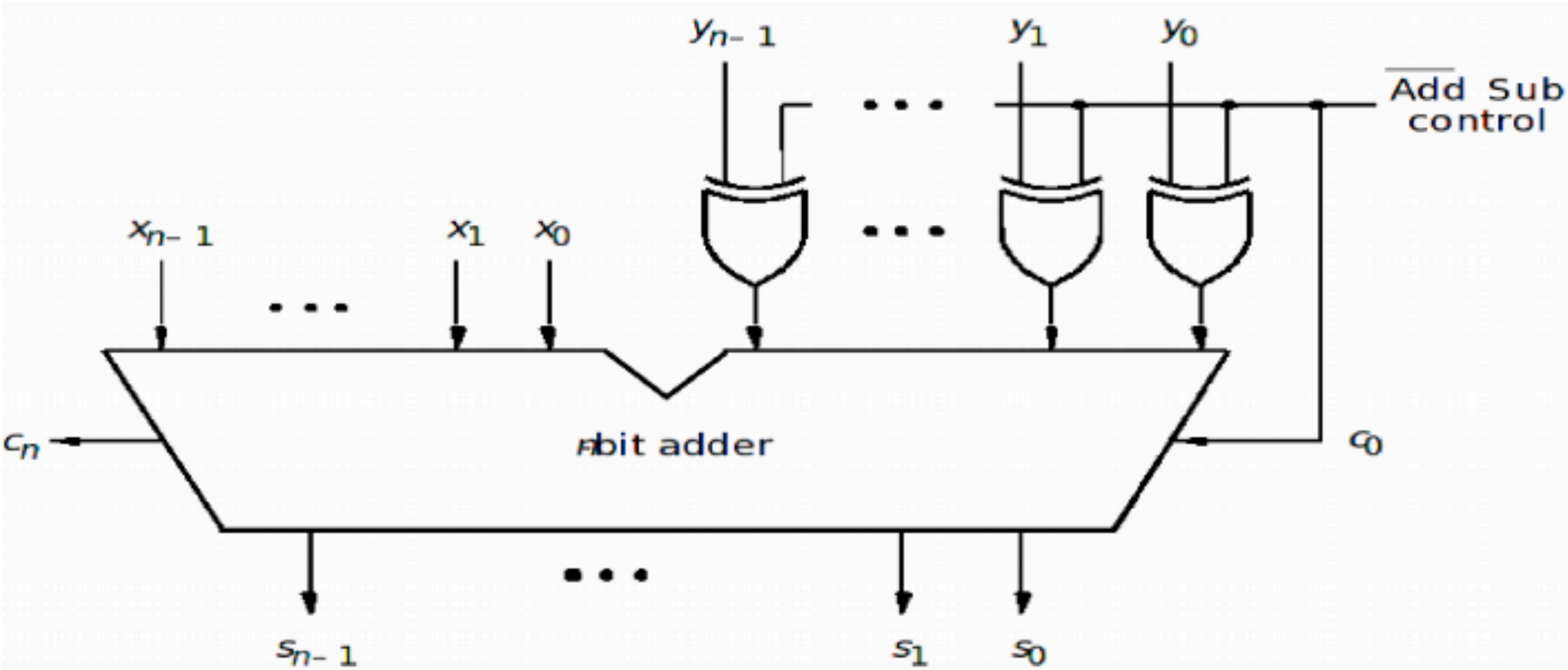
$$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$$

(b) Karnaugh maps



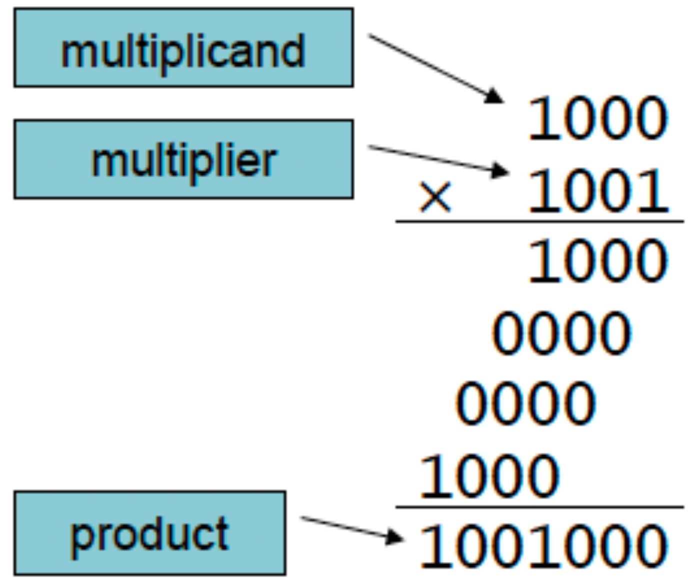
(c) Circuit

Adder/ Subtractor



Integer Multiplication

Consider the calculation on right which we limited to only 0 and 1 digits



For base-2, digits can be only 0 or 1

Means that at each step, we either get zero, or a shifted copy of multiplicand

Final step is to add values from all steps together

Length of product is the sum of operand lengths

The hardware that performs addition, subtraction and logical operations such as AND and OR is called an Arithmetic Logic Unit (ALU)

Multiplication

- Start with long-multiplication approach

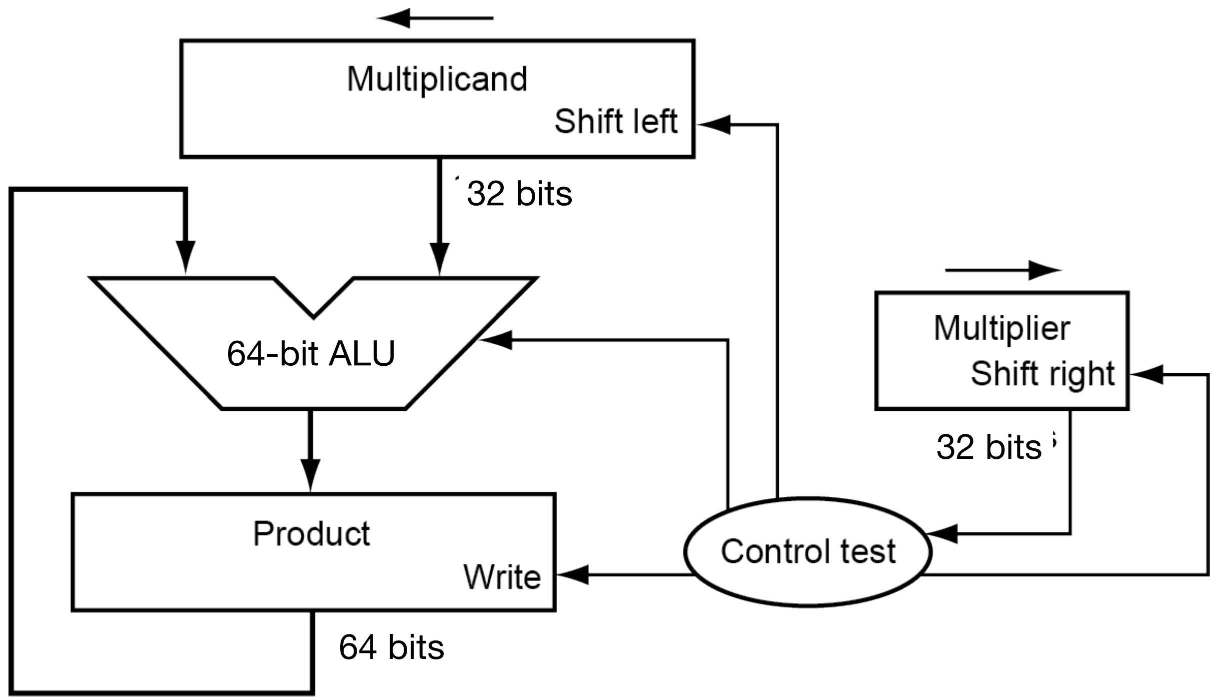
multiplicand → 1000

multiplier → × 1001

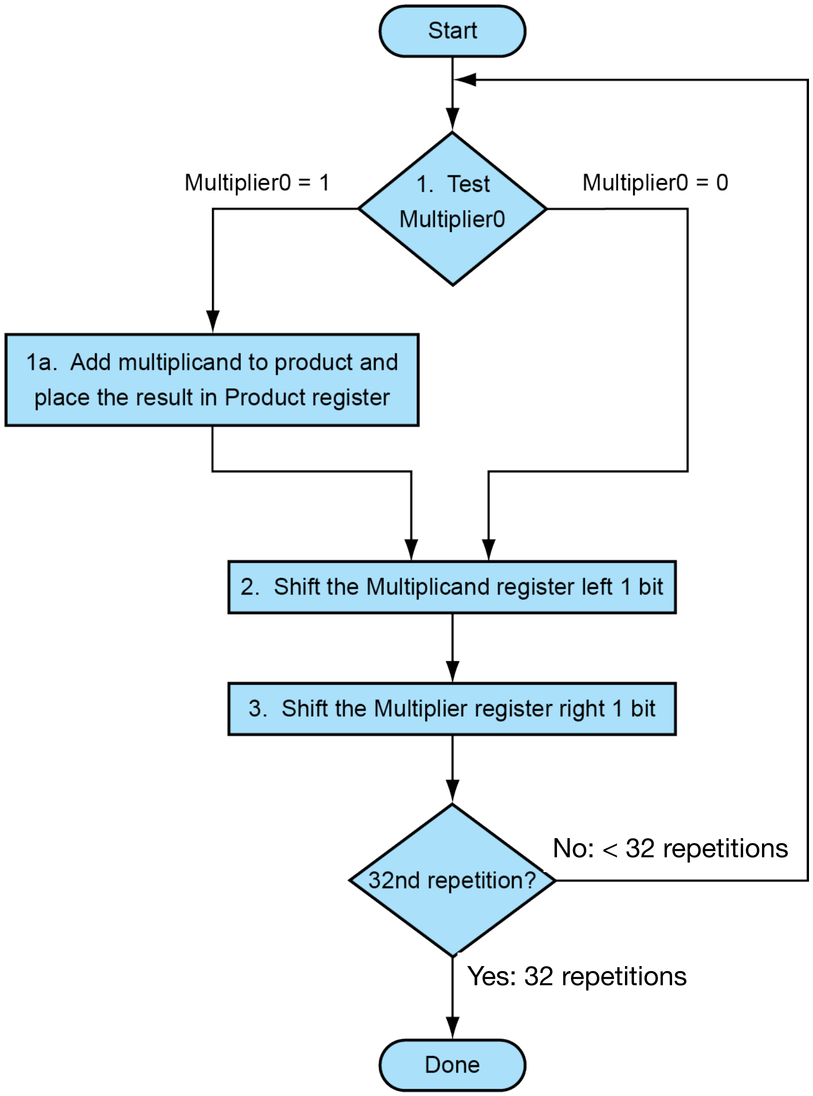
```
   1000
  0000
 0000
1000
1001000
```

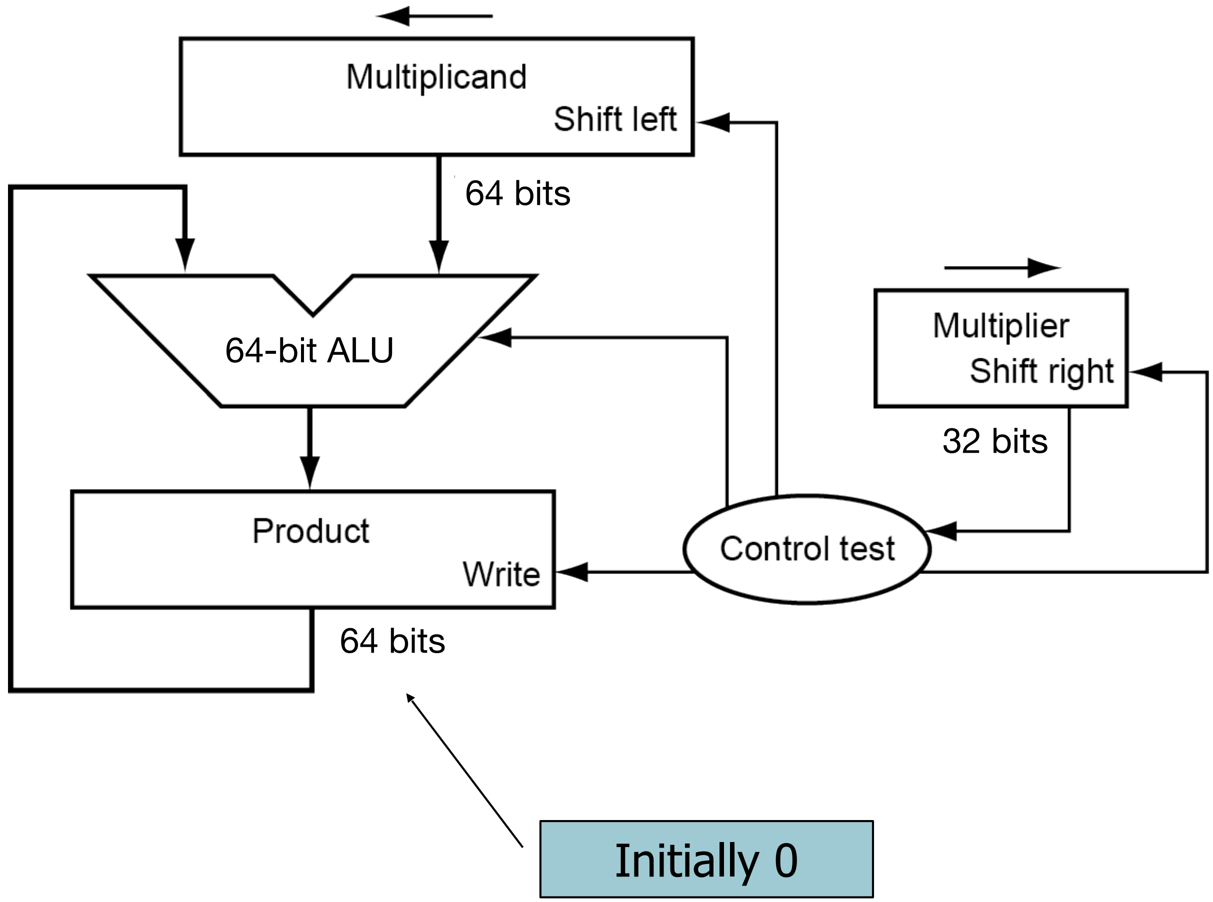
product → 1001000

Length of product is the sum of operand lengths



Multiplication Hardware





Multiplication Example

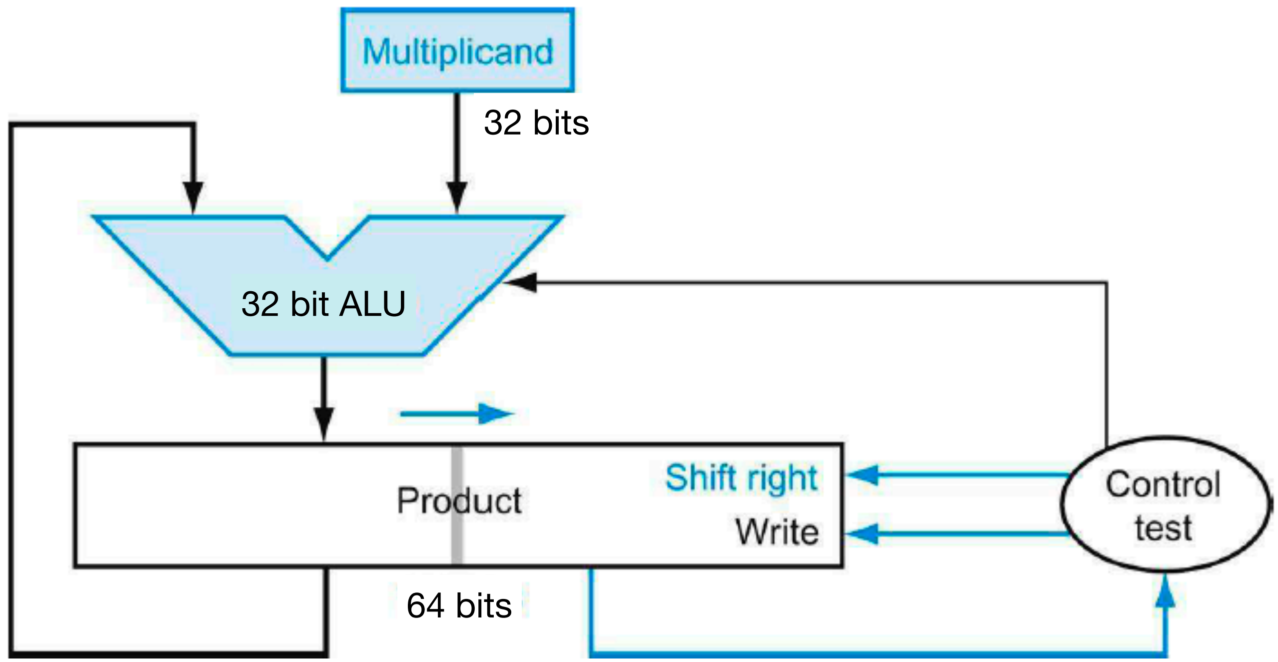
Example: 3 x 2

Multiplier = 3 (0011), Multiplicand= 2 (0010)

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⟹ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⟹ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⟹ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⟹ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Optimized Multiplier

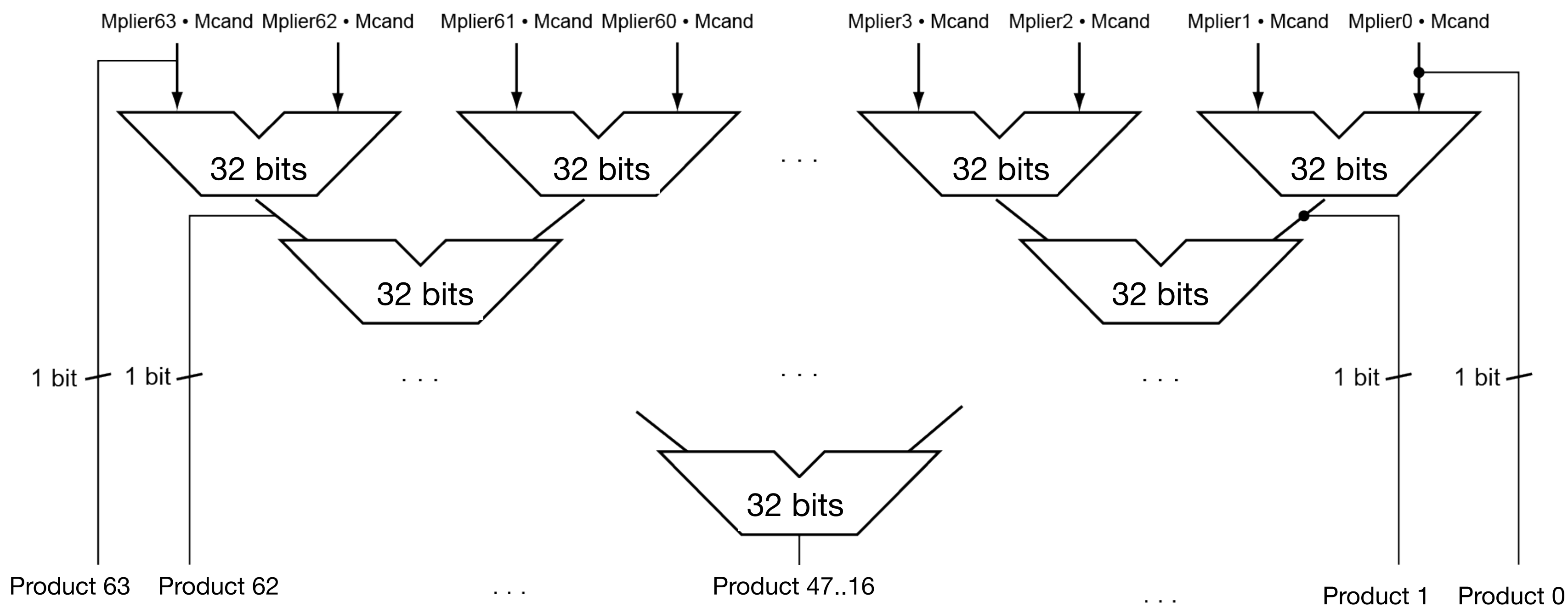
- Perform steps in parallel: add/shift



- One cycle per partial-product addition
  - That’s ok, if frequency of multiplications is low
- Multiplier is placed in the right half of the Product register

Faster Multiplier

- Uses multiple adders-cost/performance tradeoff (Moore's Law)

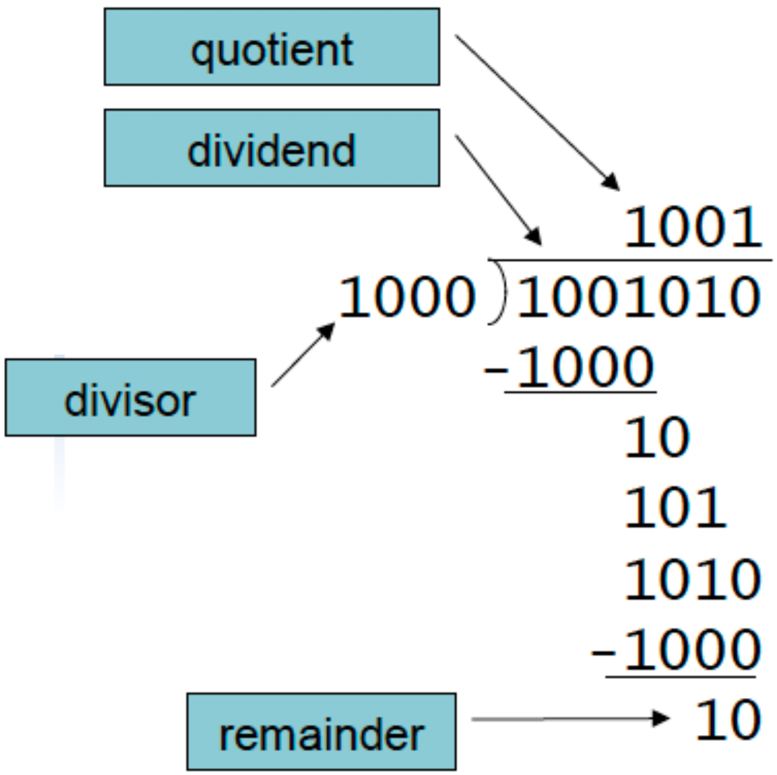


- Several multiplication performed in parallel, can be pipelined
- Instead of waiting for 32 add times, we wait just the $\log_2(32)$ or six 32-bit add times.

RISC-V Multiplication

- Four multiply instructions:
  - `mul`: multiply - Gives the lower 64 bits of the product
  - `mulh`: multiply high - Gives the upper 64 bits of the product, assuming the operands are signed
  - `mulhu`: multiply high unsigned - Gives the upper 64 bits of the product, assuming the operands are unsigned
  - `mulhsu`: multiply high signed/unsigned - Gives the upper 64 bits of the product, assuming one operand is signed and the other unsigned
  - Use `mulh` result to check for 64-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

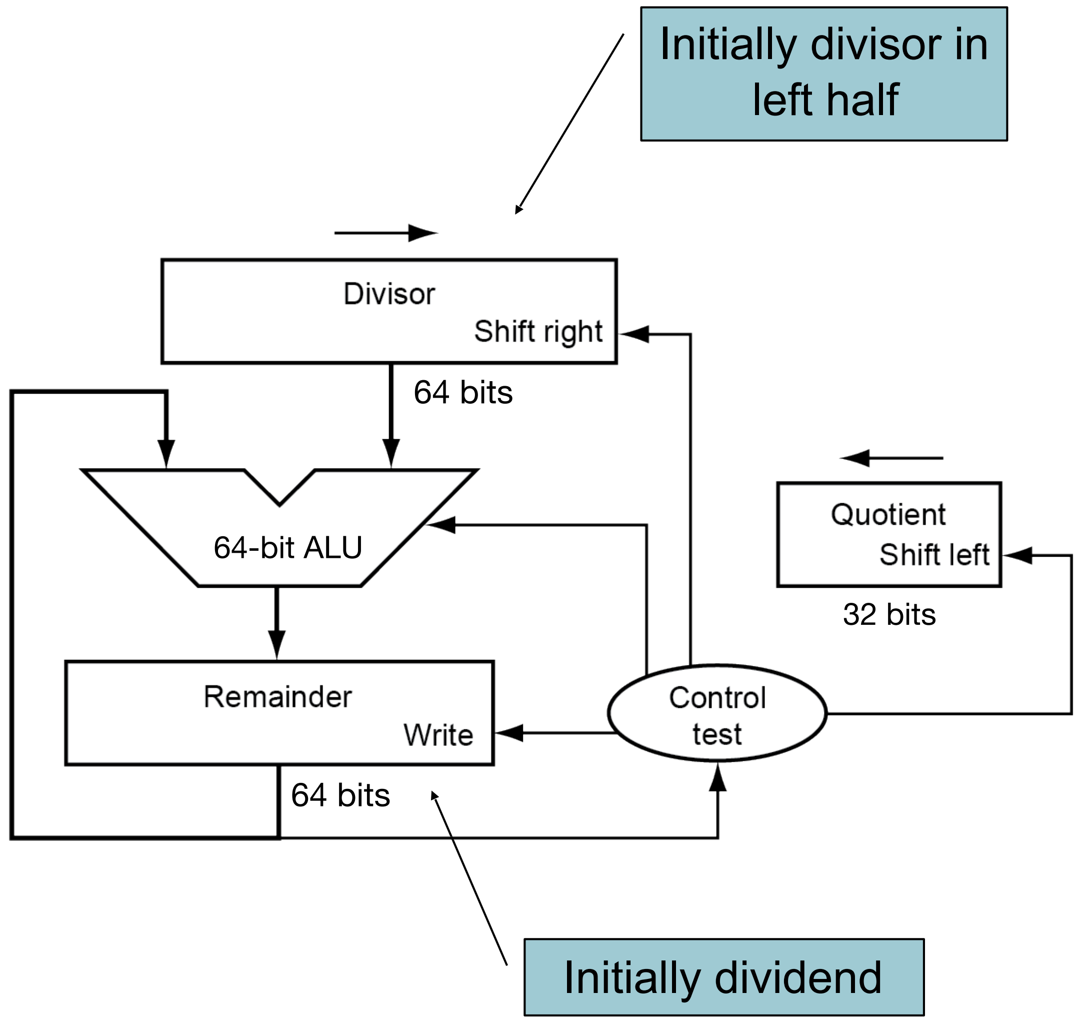
Division Hardware

We start with quotient set to zero

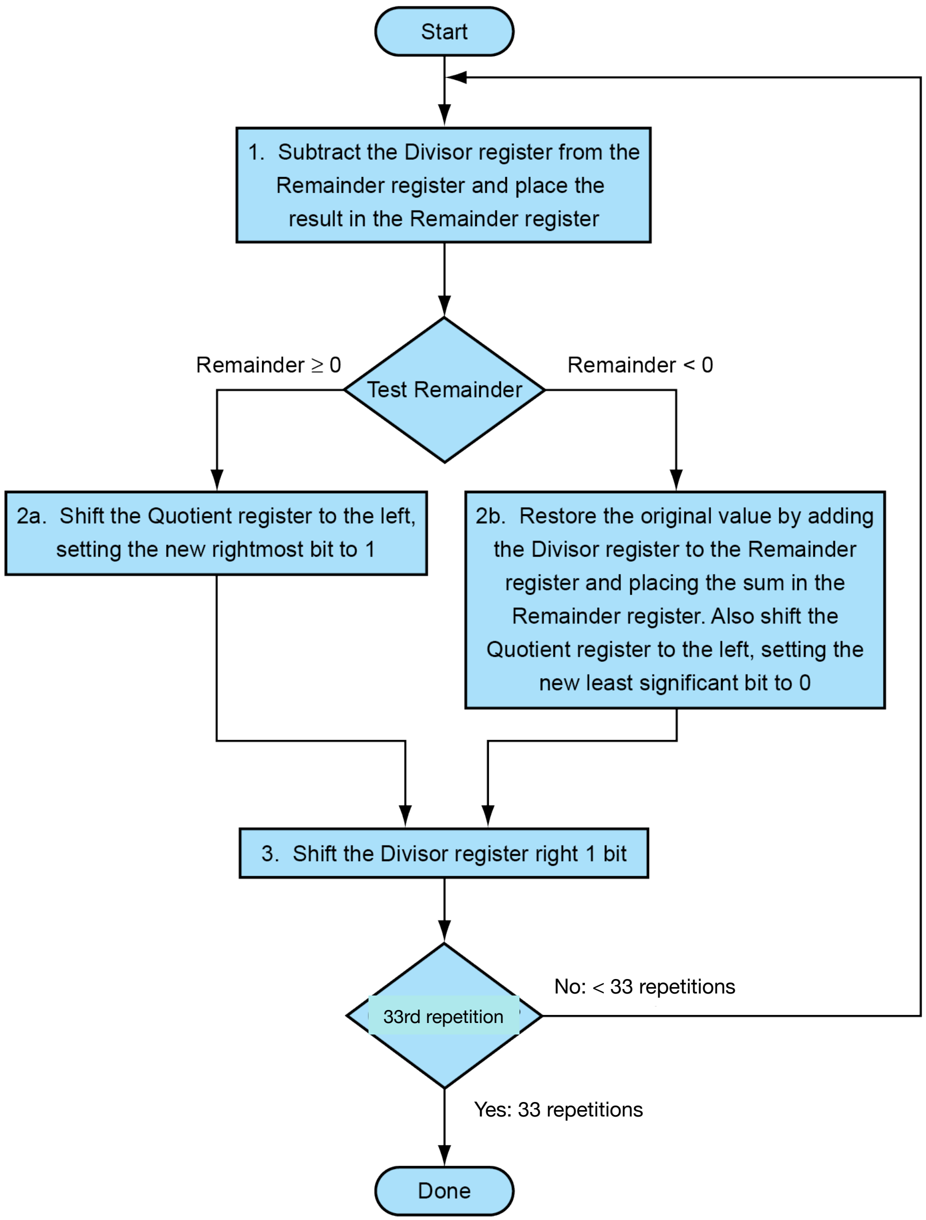
- Shifted to left one bit per iteration and new bit added

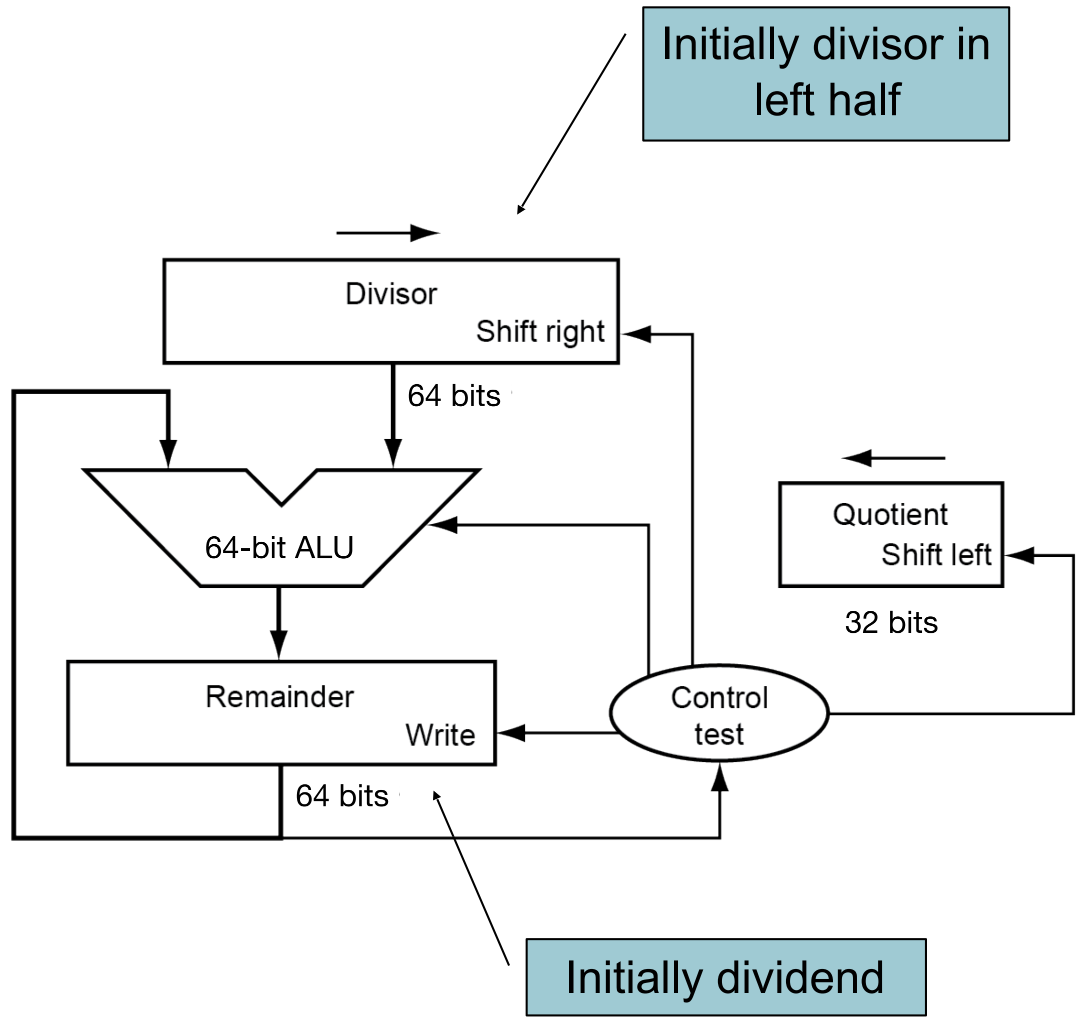
Divisor starts on left side of register and is shifted to the right one bit each iteration

Control decides when to shift divisor and quotient registers and when to store new value in remainder



Division Algorithm



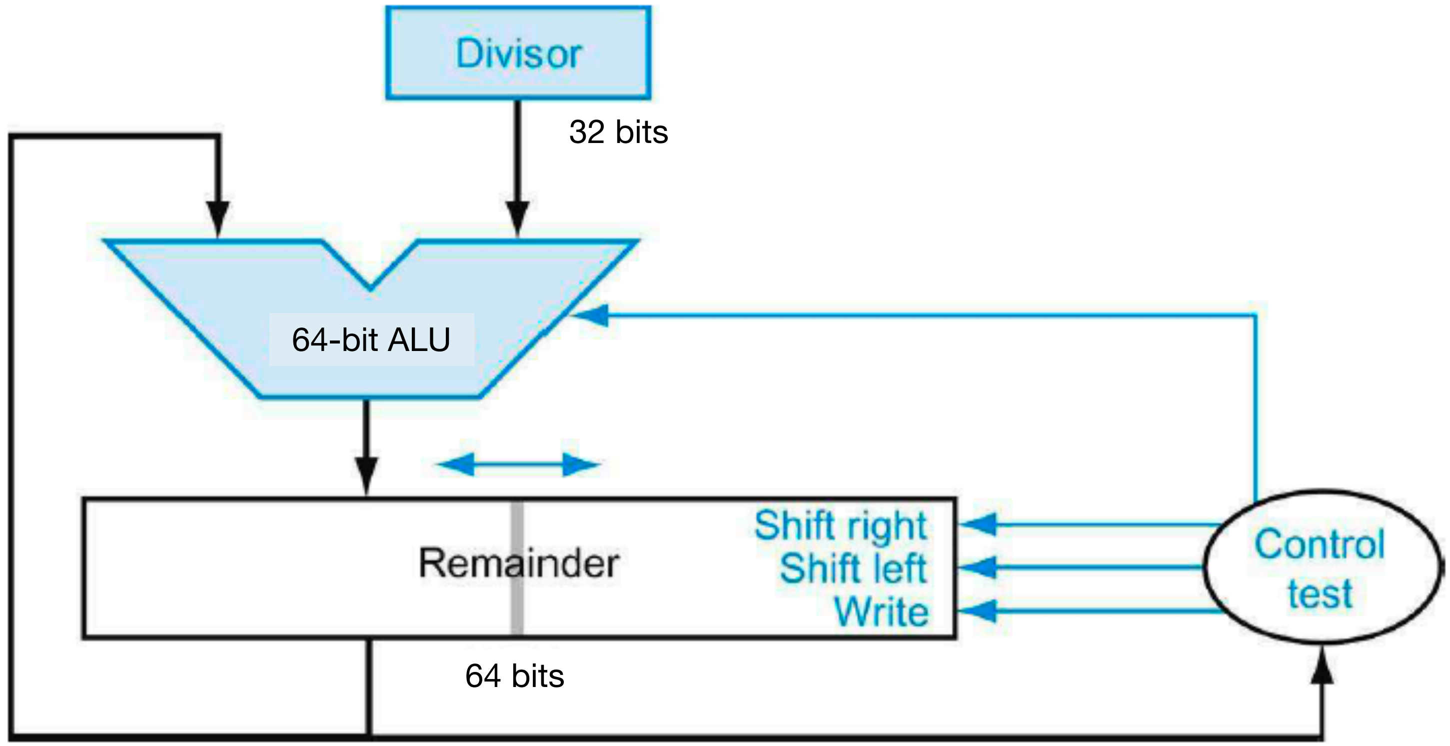


Division Example

Using a 4-bit version of the algorithm to save pages, let's try dividing 7 by 2, or 0000 0111 by 0010

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 $\Longrightarrow$ +Div, SLL Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 $\Longrightarrow$ +Div, SLL Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 $\Longrightarrow$ +Div, SLL Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 $\Longrightarrow$ SLL Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 $\Longrightarrow$ SLL Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Optimized Divider



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

Faster Division

- Can’t use parallel hardware as in multiplier
- Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT devision) generate multiple quotient bits per step
- Still require multiple steps

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

Fixed Point Numbers

Method to represent real numbers in digital hardware

Number represented as an *n-bit* integer part, and a *k-bit* fractional part

This means the decimal point is fixed

For binary number:
$B = b_{n-1} \ldots b_0.b_{-1}b_{-2} \ldots b_{-k}$
its base-10 value is:
$V(B) = \Sigma_{i=-k \text{ to } n-1} (b_i \times 2^i)$

For B = 000.0001001
if n = 4, and k= 3, we would get 0000.000 !?

Floating Point

- Representation for non-integral numbers
- Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- 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
  - IEEE Single precision (32-bit)
  - IEEE 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 ($0 \Rightarrow$ non-negative, $1 \Rightarrow$ negative)
- Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
  - 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 = 1203

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
  - $\pm 1.00000000000000000000000 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - $\pm 1.11111111111111111111111 \times 2^{+127}$
  - 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 ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
- 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

- 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 (32bits)

- Double: 1011111111101000…00 (64bits)

Floating-Point Example

- What number is represented by the single-precision float

  1100000001010000...00
  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = $10000001_2 = 129$

- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$
  $= (-1) \times 1.25 \times 2^2$
  $= -5.0$

Overflow and Underflow

With floating point numbers, overflow means that the exponent is too large to fit in the exponent field.

Underflow occurs when the negative exponent is too small to fit in the exponent field.

For single precision, exponents can go in range from -126 to 127

For double precision, the range is -1022 to 1023

Overflow and Underflow II

RISC-V does not cause exceptions on arithmetic errors: overflow, underflow.

Instead, both integer and floating-point arithmetic produce reasonable default values and set status bits.

The status bits can be tested by an operating system or periodic interrupt.

Denormalized Numbers

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

- They have the same exponent as zero but a nonzero fraction.
- Smaller than normal numbers
  - Allow for gradual underflow, with diminishing precision
  - Denormal with fraction = 000…0
  - Exponent = 000...0 $\Rightarrow$ hidden bit is 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

Floating-Point Addition: Decimal

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 significants

$9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$

3. Normalize result & check for over/underflow

1.0015×10^{2}

4. Round and renormalize if necessary

1.002×10^{2}

Floating-Point Addition: Binary

Now consider a 4-digit binary example

$1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ $(0.5_{10} + -0.4375_{10})$

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) $= 0.0625_{10}$

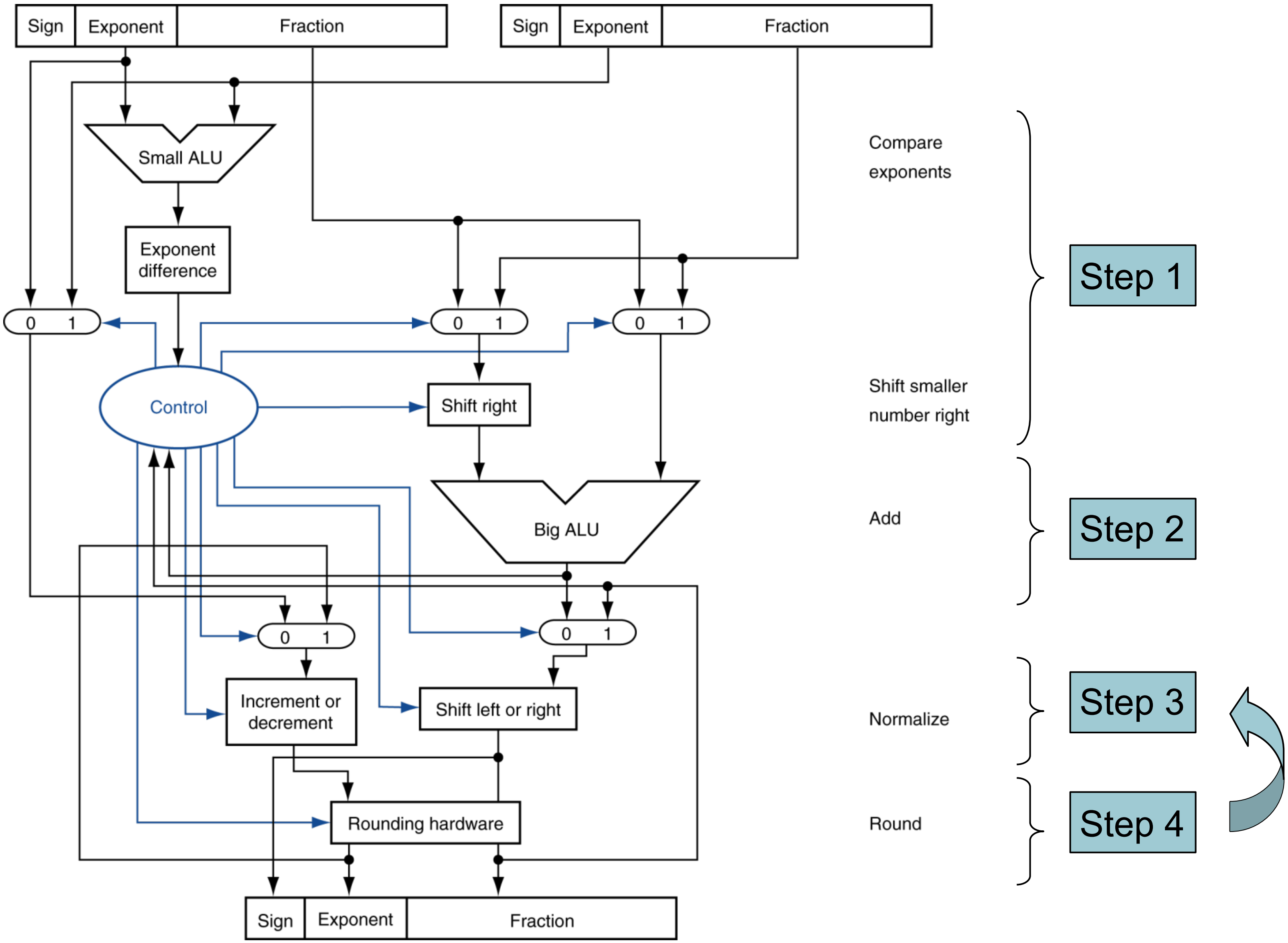
FP Adder Hardware

- 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



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 $\leftrightarrow$ 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
  - More registers with minimal code-size impact
- 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

FP Example: °F to °C

C code:

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

- `fahr` in `f10`, `result` should also go to `f10`
- we assume that the compiler places the floating point constants in memory with easy reach of register `x3`

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
```

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

REAL Stuff (Additional Reading)

3.6 Parallelism and Computer Arithmetic: Subword Parallelism
3.7 Real Stuff: Streaming SIMD Extensions and Advanced Vector Extensions in x86
3.9 Fallacies and Pitfalls
3.10 Concluding Remarks

Thank You