CS 31007 Autumn 2021

COMPUTER ORGANIZATION AND ARCHITECTURE

Instructors

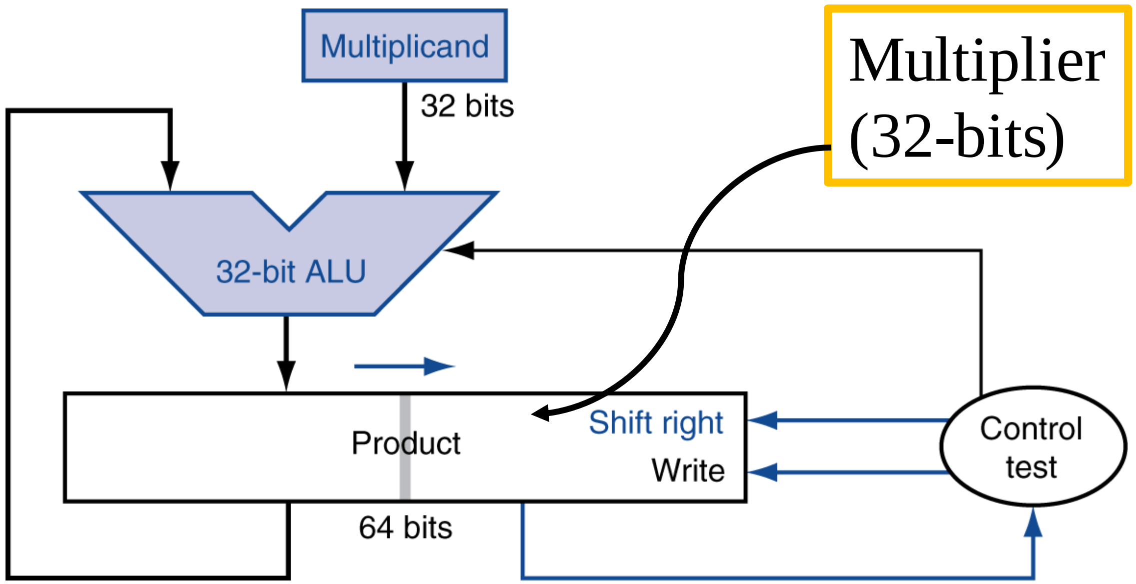
Rajat Subhra Chakraborty (*RSC*)
Bhargab B. Bhattacharya (*BBB*)

Lecture #22: Computer Arithmetic

20 September 2021

Indian Institute of Technology Kharagpur
Computer Science and Engineering

Recap: Integer Multiplier using Repeated Add/Shift



- Can we expedite multiplication?
- Handling negative numbers?
- - Booth's Algorithm

Recall: Multiplication Example

0010 × 0110 = ? 0010 (+ 2, multiplicand); 0110 (+ 6, multiplier)

Itera-tion	multi-plicand	Orignal algorithm: Step	Orignal algorithm: Product
0	0010	Initial values	0000 0110
1	0010	1:0 ⇒ no operation	0000 0110
	0010	2: Shift right Product logical shift	0000 0011
2	0010	1a:1⇒ prod = Prod + Mcand	0010 0011
	0010	2: Shift right Product	0001 0001
3	0010	1a:1⇒ prod = Prod + Mcand	0011 0001
	0010	2: Shift right Product	0001 1000
4	0010	1:0 ⇒ no operation	0001 1000
	0010	2: Shift right Product +12	0000 1100

Booth’s Encoding
(valid for signed multiplication as well)

- Recall old trick

  Example: 123454×9

  $\Rightarrow$ six partial products plus addition of six numbers

  - $123454 \times 9 = 123454 \times (10 - 1) = 1234540 - 123454$
  - Transform addition of six partial products to
  - one shift and one subtraction!

- Booth’s algorithm applies the same principle
  - in binary we have just ‘1’ and ‘0’

Booth's Encoding

- Multiply x by 0111

x (multiplicand); 0111 (multiplier) => $x \times 7$

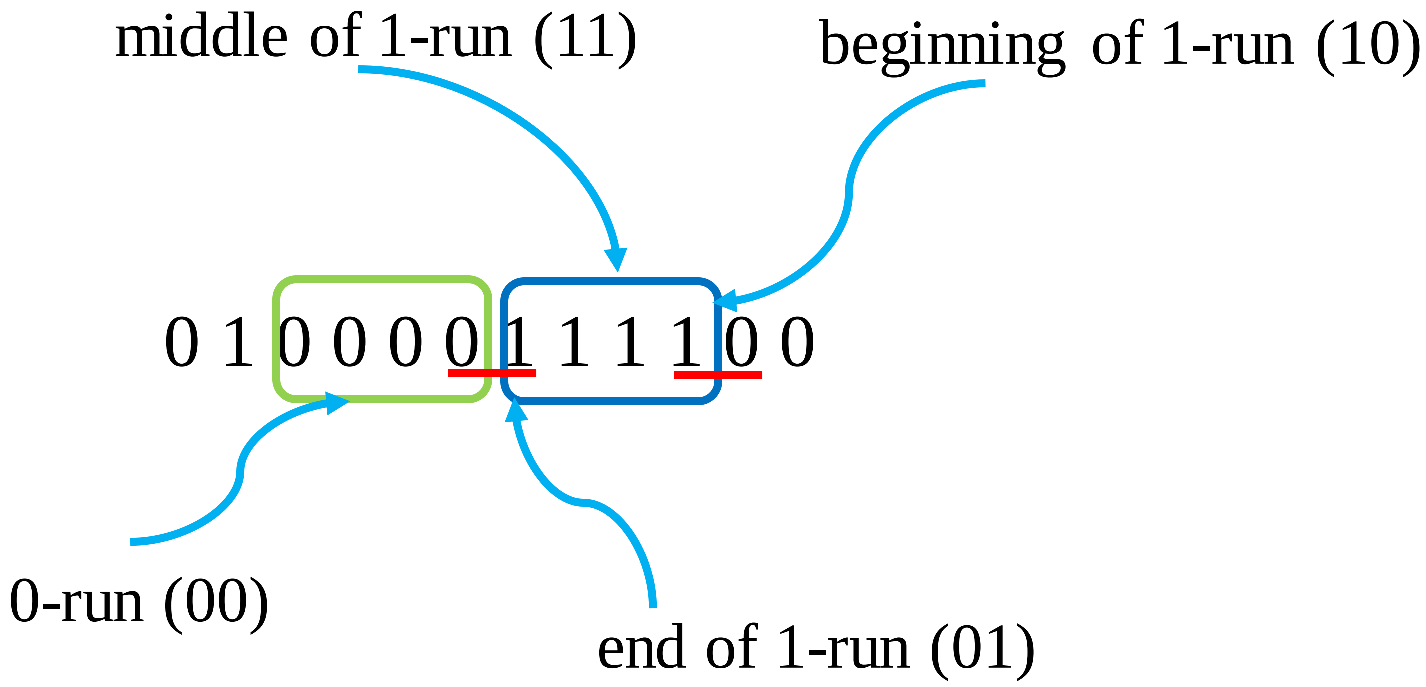
Search for a run of '1' bits in the multiplier

- – e.g. '0111' has a run of 3 consecutive '1' bits
- – Multiplying by '0111' (7 in decimal) is equivalent to multiplying by 8 and subtracting once, since

  $x \times 7 = x \times (8 - 1) = 8x - x$ => (shift-left x, three times) - x

- Hence, iterate right to left and look for "runs of 1":
- $x \times 111 = x \times (1000 - 1) = (x \times 2^3) - x$
  - – Subtract multiplicand from product at first '1'
  - – Shift multiplicand by 3-bits on left and add to the partial product after the last '1'
  - – Do nothing for the consecutive 1-bits/in the middle, or for 0-runs (actually, we keep on shifting the product register)

Booth's Encoding for Multiplier



Booth’s Algorithm

Current bit	Bit to right	Explanation	Example (multiplier)	Operation
1	0	Begins run of ‘1’	0000111**10**00	Subtract/shift
1	1	Middle of run of ‘1’	00001**11**1000	Nothing/shift
0	1	End of a run of ‘1’	000**01**111000	Add/shift
0	0	Middle of a run of ‘0’	0**00**01111000	Nothing/shift

Booth's algorithm: Example

0010 × 1101 = ? 0010 (+ 2, multiplicand); 1101 (- 3, multiplier)

arithmetic shift

op performed in the left half

initialization

Itera-tion	multi-plicand	Booth's algorithm: Step	Booth's algorithm: Product
0	0010	Initial values	0000 1101 0
1	0010	1c: 10⇒ prod = Prod - Mcand	1110 1101 0
	0010	2: Shift right Product	1111 0110 1
2	0010	1b: 01⇒ prod = Prod + Mcand	0001 0110 1
	0010	2: Shift right Product	0000 1011 0
3	0010	1c: 10⇒ prod = Prod - Mcand	1110 1011 0
	0010	2: Shift right Product	1111 0101 1
4	0010	1d: 11 ⇒ no operation	1111 0101 1
	0010	2: Shift right Product	1111 1010 1

start of 1-run

end of 1-run

-6

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt  /  multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd  /  mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

So far covered in computer arithmetic…

- ❖ Integer Number Systems and Overflow
- ❖ Ripple-Carry Adder (RCA)
- ❖ Carry-Lookahead Adder (CLA)
- ❖ Hybrid Adder, CLT
- ❖ Carry-Select Adder (CSA)
- ❖ Brent-Kung's Parallel Prefix Adder (PPA)
- ❖ Carry-Save Adders (for adding multiple operands)
- ❖ Integer Multiplication

- ❖ Integer Division: Reading Assignment
- ❖ Floating-Point Arithmetic and Hardware

Division

- Implemented by successive subtractions
- Result must verify the equality

  Dividend = (Multiplier × Quotient) + Remainder

Another powerful division method (Goldschmidt's algorithm):

In computer science, division can be accomplished by multiplying a number with the reciprocal of the multiplier!

MK MORGAN KAUFMANN

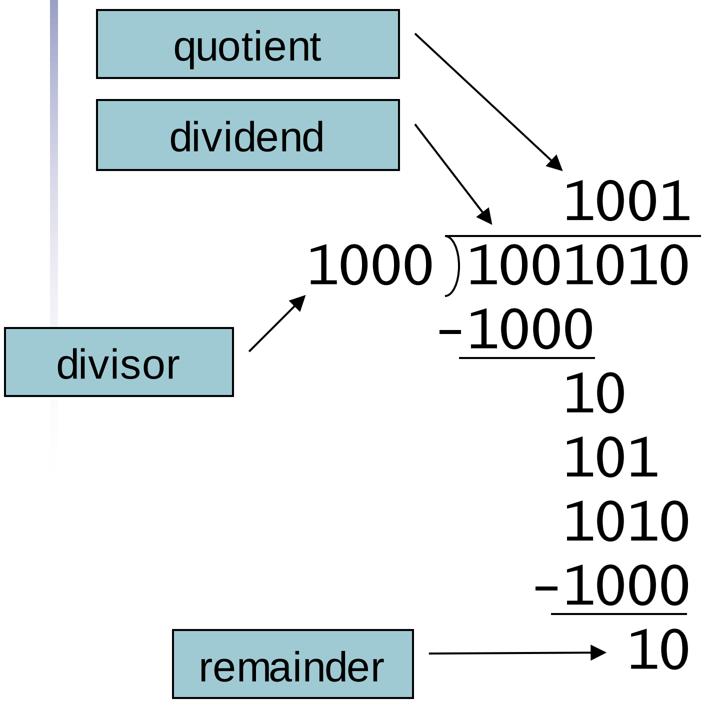
In biology, multiplication is accomplished by division!

Decimal division

```
     303
  ┌───────
7 │  2126
   -210
   ────
      26
     -21
     ───
       5
```

- What are the rules?
  - Repeatedly try to subtract a multiple of divisor from dividend
  - Record multiple (or zero)
  - At each step, repeat with a lower power of ten
  - Stop when remainder is smaller than divisor

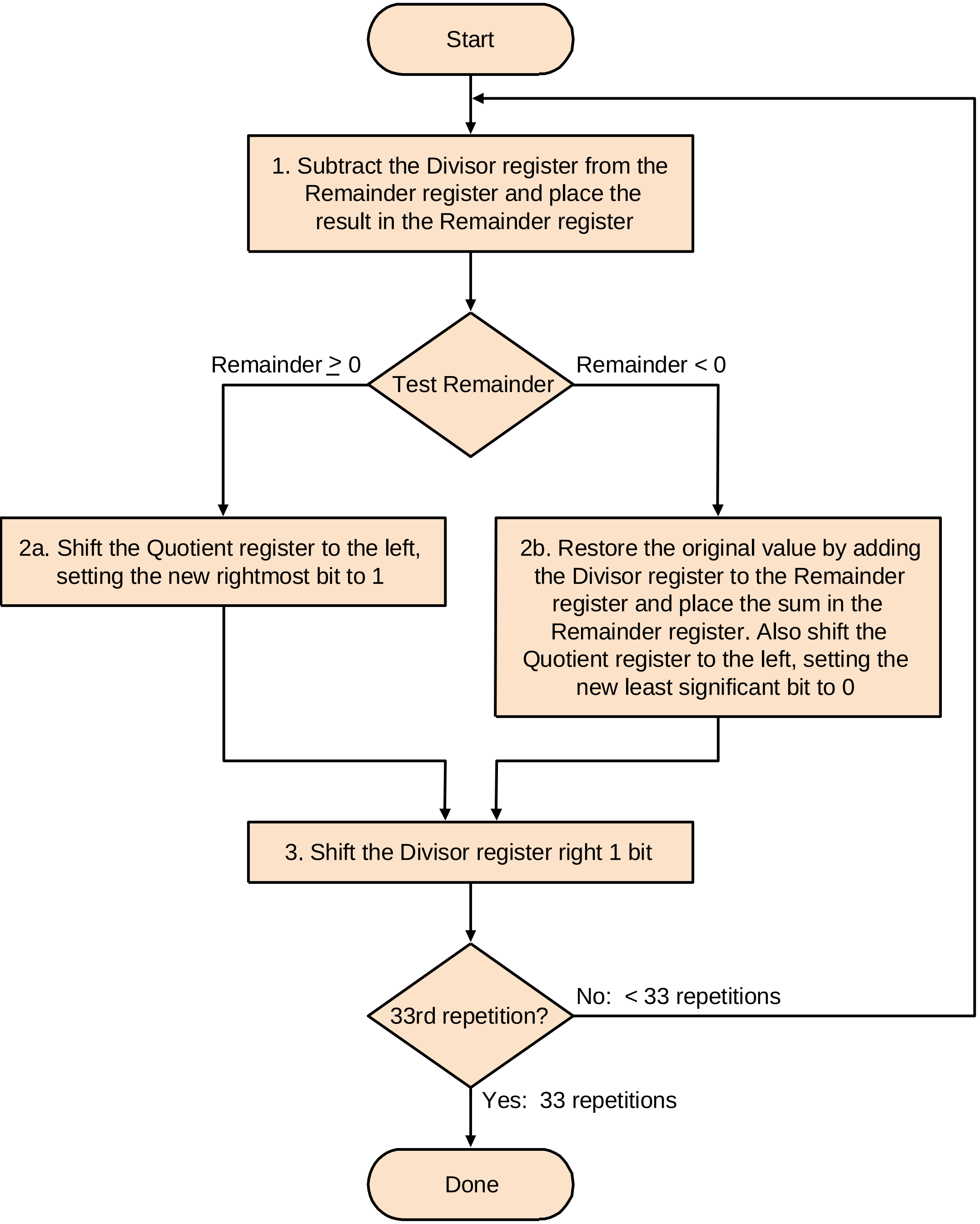
Integer Division in Binary

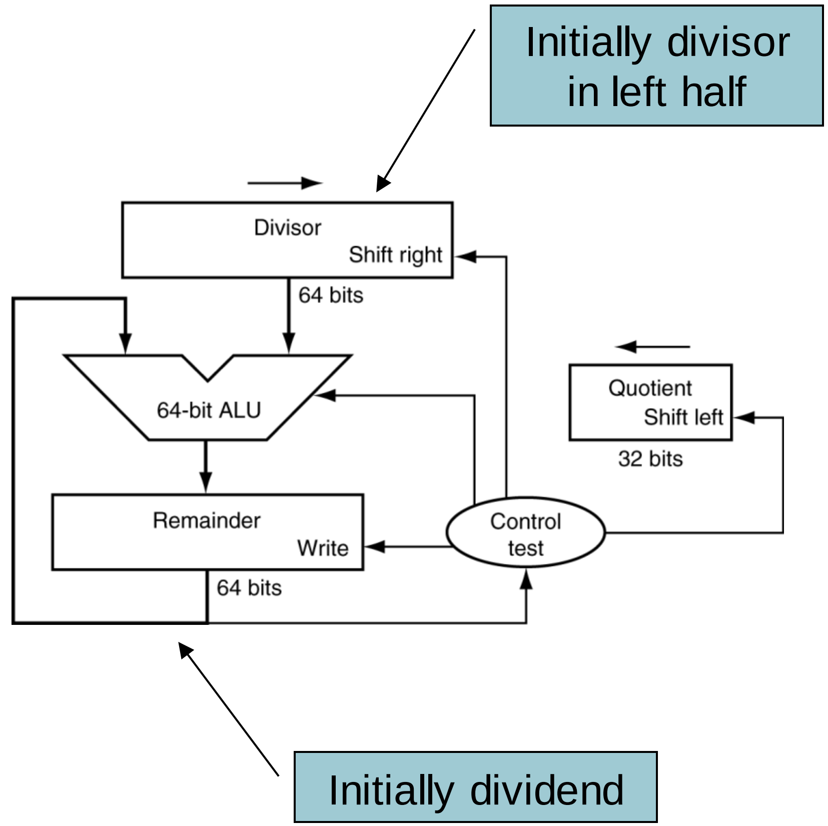


n-bit operands yield *n*-bit quotient and remainder

- 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

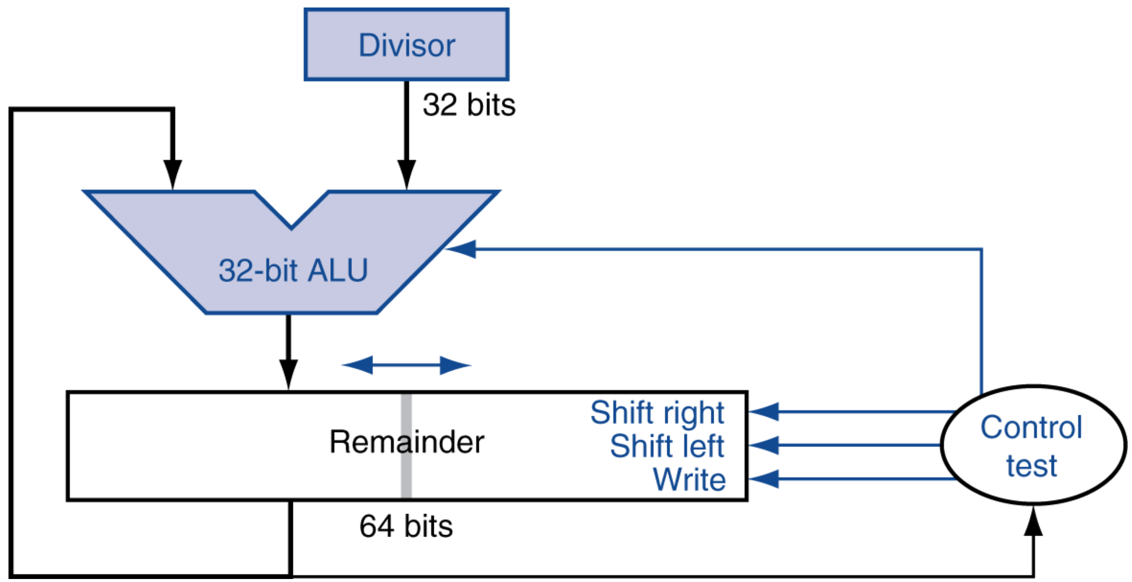




Restoring Division

Iteration	Divisor	Divide algorithm	
		Step	Remainder
0	0010	Initial values	0000 0111
	0010	Shift Rem left 1	0000 1110
1	0010	2: Rem = Rem - Div	1110 1110
	0010	3b: Rem < 0 $\Rightarrow$ + Div, sll R, R0 = 0	0001 1100
2	0010	2: Rem = Rem - Div	1111 1100
	0010	3b: Rem < 0 $\Rightarrow$ + Div, sll R, R0 = 0	0011 1000
3	0010	2: Rem = Rem - Div	0001 1000
	0010	3a: Rem $\geq$ 0 $\Rightarrow$ sll R, R0 = 1	0011 0001
4	0010	2: Rem = Rem - Div	0001 0001
	0010	3a: Rem $\geq$ 0 $\Rightarrow$ sll R, R0 = 1	0010 0011
Done	0010	shift left half of Rem right 1	0001 0011

Optimized Divider



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

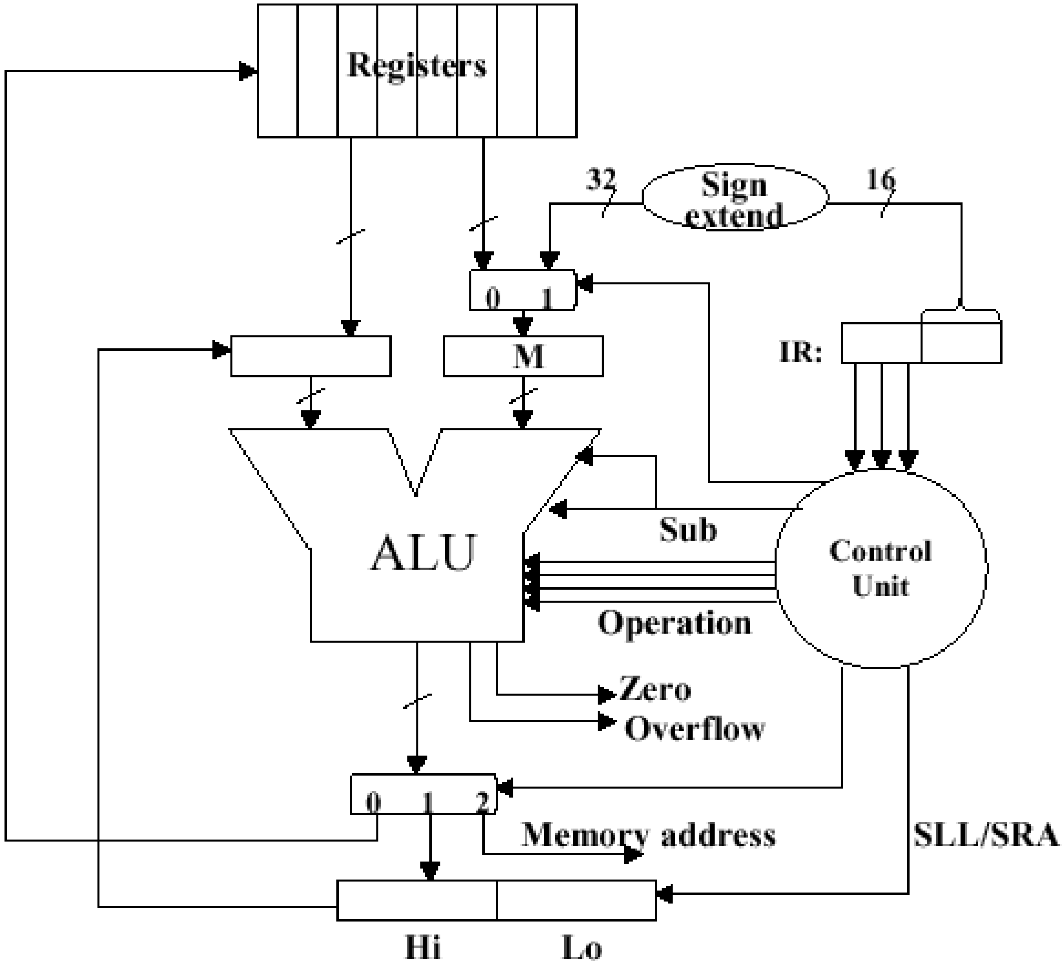
Faster Division

- 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

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt` / `divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result

MIPS Architecture for Integer Arithmetic: Multiplication and Division



CS 31007 Autumn 2021

COMPUTER ORGANIZATION AND ARCHITECTURE

Instructors

Rajat Subhra Chakraborty (*RSC*)

Bhargab B. Bhattacharya (*BBB*)

Lecture #23, #24: Floating-Point Arithmetic

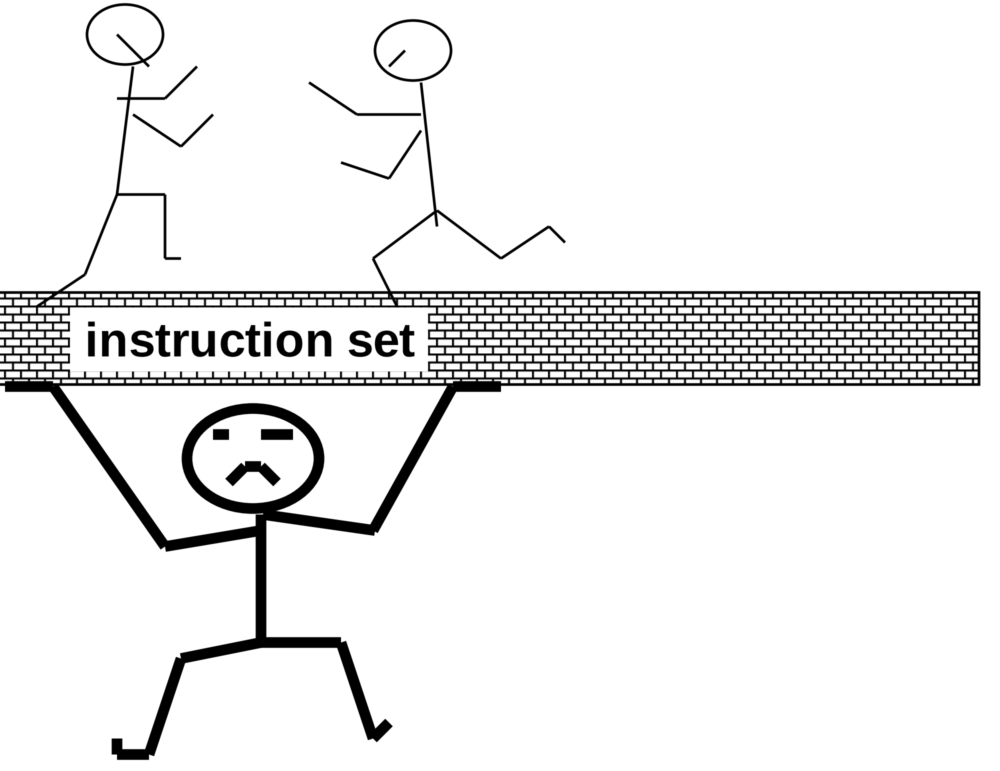
21 September 2021

Indian Institute of Technology Kharagpur

Computer Science and Engineering

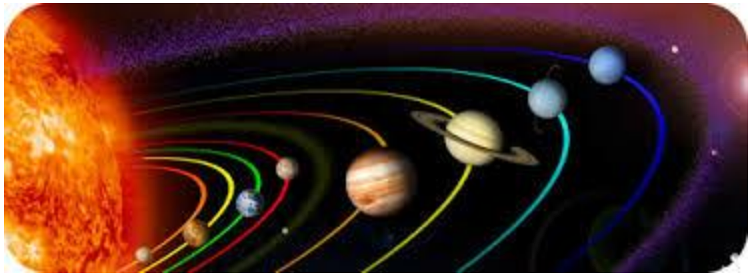
Floating Point:
Format, Arithmetic, and Hardware Implementation

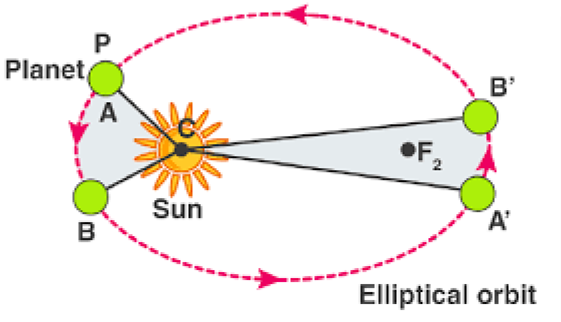
software



hardware

A fundamental theoretical question: Can we prove Kepler's Law of planetary motion by virtue of experiments?





The IEEE Floating-Point Standard

So far as the theories of mathematics are about reality, they are not certain; so far as they are certain, they are not about reality. - *Albert Einstein*

A computation error observed by a UG student led to the ACM Turing Award later ….

- 1953: Willian Kahan, a UG student of Math at the *University of Toronto* was simulating numerically the dynamics of the wing controller of an aircraft during take-off and landing
- Observed certain unexpected results due to errors in computation
- => concept of floating-point (FP) arithmetic
- => principal architect behind **IEEE 754 FP standard (1985)**
- => Kahan honored with ACM Turing Award (1989)

William Kahan
(1933 -)

Floating-Point Representation

- Used to represent *real numbers:* -34.986×10^{-22}, π, e, $\sqrt{2}$
- Defined by IEEE 754 Standard

  --- Kahan (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)

Infamous Intel Pentium Bug (1994) =>FDIV => loss of $300 Million

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent (E)	Mantissa (M)

Fraction
≡Mantissa
≡ Significand

$$N = (-1)^S \times (1.M) \times 2^{(\text{Exponent}-\text{Bias})}$$

- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ 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)
  - Normalized Significand is Mantissa with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023
  - Trade-off between range (E) and precision (M)

IEEE 754 Single-Precision Floating-Point

Representation of floating point numbers in IEEE 754 standard:

single precision

	1	8	23
sign	**S**	**E**	**M**

FP-exponent E:
excess 127 binary coding

Actual exponent is $e = E - 127$

Floating-point exponent $E = e + 127$

mantissa:
sign + magnitude, normalized binary significand w/ hidden integer bit: 1.M

FP-exponent (*E*)	**Actual exponent *e***
00000000 (0)	**Special use**
00000001 (1)	**- 126**
00000010 (2)	**- 125**
………….	**…….**
01111111 (127)	**0**
………….	**…….**
11111110 (254)	**+ 127**
11111111 (255)	**Special use**

IEEE 754 floating-point standard

- Leading “1” bit of significand is implicit

- Exponent is “biased” to make comparison easier
  - all 0s is smallest exponent; all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - summary: $(-1)^{\text{sign}} \times (1+\text{significand}) \times 2^{\text{exponent} - \text{bias}}$

- Example 1 (Encoding):

  For a given decimal number, construct its FP-representation

  - decimal: - 0.75

  → binary: - 0.11 = - 1.1 x 2^{-1} (normalized)

  → Floating-point exponent = - 1 +127 = 126 = 01111110

IEEE single precision:

- 0.75 = **1 01111110 10000000000000000000000**

Decoding a floating-point number

S	E	M

- Sign indicated by first bit $S \rightarrow (-1)^S$
- Subtract 127 from biased exponent E to obtain the actual exponent $e = E - 127$
- Number in binary = $(-1)^S\ 1.\mathrm{M} \times 2^e$

Example (Decoding)

$\boxed{0}\ \boxed{1000\ 0000}\ \boxed{1000\ 0000\ 0000\ 0000\ 0000\ 000}$

- Sign bit is zero:

  Number is positive
- Biased exponent $E = 1000\ 0000\ |_2 = 128$;
- Actual exponent $e = E - 127 = 1 \rightarrow 2^1$
- Significand $\rightarrow$ 1.1 (restored the hidden bit)
- The number $= + 1.1 \times 2^1 |_2 = + 11 |_2 = + 3.0|_{10}$

Example (Decoding)

1	0111 1110	11000000000000000000000

- Sign bit is one:

  Number is negative
- Biased exponent $E = 0111\ 1110\ |_2 = 126$;
- Actual exponent $e = E - 127 = -1 \rightarrow 2^{-1}$
- Significand $\rightarrow$ 1.11 (restored the hidden bit)
- The number $= -1.11 \times 2^{-1}\ |_2 = -0.111\ |_2 = -0.875|_{10}$

Example (Encoding)

- Represent –2 in FP-format
  - Convert to binary: 10
  - Normalize: 1.0×2^1
  - Sign bit is 1
  - FP-exponent is 127 + 1 = 128 = 10000000_{two}
  - Mantissa is 00…0

1	1000 0000	0000 0000 0000 0000 0000 000

0/1	0000 0000	0000 0000 0000 0000 0000 000
0	0	Zero
0	Nonzero	
1-254	Anything	
255	0	+ / - infinity
255	Nonzero	NaN like 0/0 o

* Any non-zero number that is smaller than the smallest normalized FP-number is a denormal number (consider magnitude only)

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Mantissa: 000…00 $\Rightarrow$ normalized 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
  - Mantissa: 111…11 $\Rightarrow$ normalized significand ≈ 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
  - Mantissa : 000…00 $\Rightarrow$ norm. 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
  - Mantissa: 111…11 $\Rightarrow$ norm significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all mantissa-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

Single-Precision Normalized Range

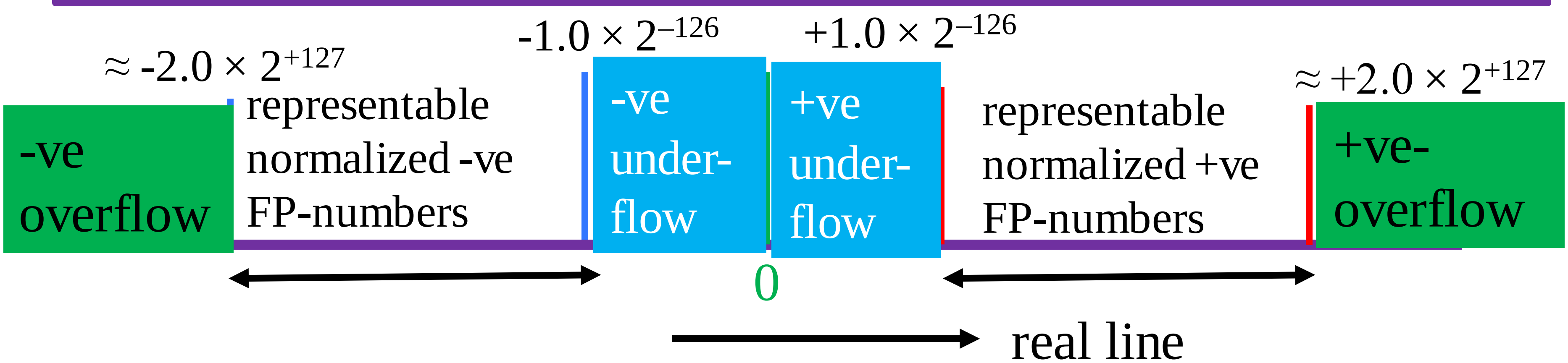
- Exponents 00000000 and 11111111 reserved
- Smallest value

0/1	0000 0001	0000 0000 0000 0000 0000 000

  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
  - Largest value

0/1	1111 1110	1111 1111 1111 1111 1111 111

  - $= \pm(2.0 - 2^{-23}) \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$



Single-Precision FP-Denormal Numbers

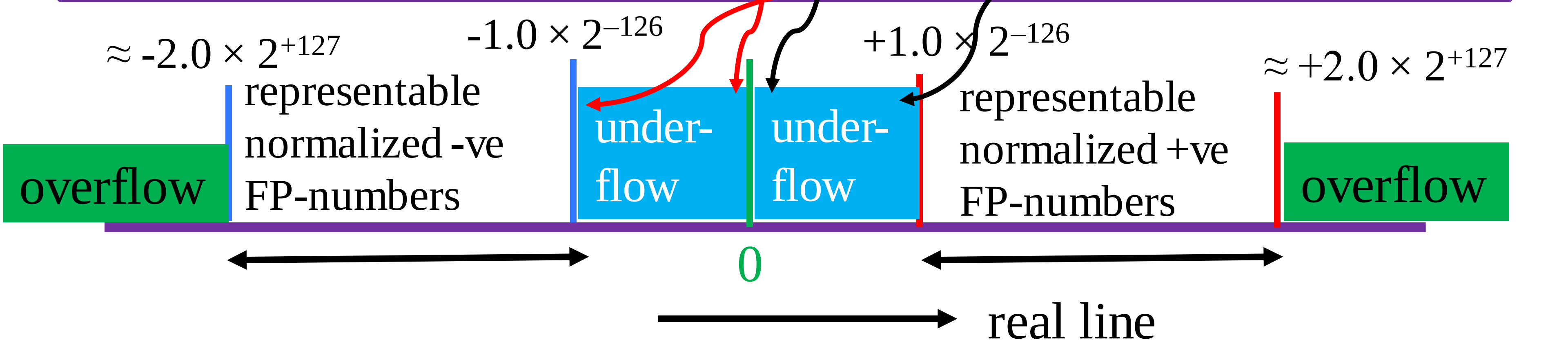
- Exponents 00000000 and 11111111 reserved
- Smallest value (normalized)

0/1	0000 0001	0000 0000 0000 0000 0000 000

  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
  - Denormal numbers (gradual underflow)

0/1	0000 0000	$M \neq 0$

$= \pm\, 0.M \times 2^{-126}$ $\{\text{min: } \pm\, 2^{-23} \times 2^{-126}\text{; max: } \pm\, (1-2^{-23}) \times 2^{-126}\}$



Denormal FP-Numbers

- Exponent = 000...0 $\Rightarrow$ hidden bit is 0

$$N = (-1)^S \times (0.\text{Mantissa}) \times 2^{-126}$$

- Smaller than normalized numbers
  - allow for gradual underflow, with diminishing precision
- Denormal with mantissa = 000...0
- → +0, -0

Infinities and NaNs

- Exponent = 111...1, Mantissa= 000...0
  - ± Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Mantissa ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0, √-3
  - Can be used in subsequent calculations

Floating Point Complexities

- Operations are somewhat more complicated
- In addition to overflow, we may have "underflow"
- Accuracy can be a big problem
  - IEEE 754 keeps three extra bits, guard, round, and sticky
  - several rounding modes
  - Non-zero number divide-by-zero yields "infinity" → overflow
  - Non-zero number divide-by-infinity yields → underflow
  - zero divide-by-zero yields "not a number (NaN)"
- Implementing the standard can be tricky
- Not using the standard can be even worse
- Remember the 1994 Pentium FDIV bug; write-off cost US$ 300 M

CS 31007 Autumn 2021

COMPUTER ORGANIZATION AND ARCHITECTURE

Instructors

Rajat Subhra Chakraborty (*RSC*)

Bhargab B. Bhattacharya (*BBB*)

Lecture #25: Tutorial on Floating-Point Arithmetic

23 September 2021

Indian Institute of Technology Kharagpur

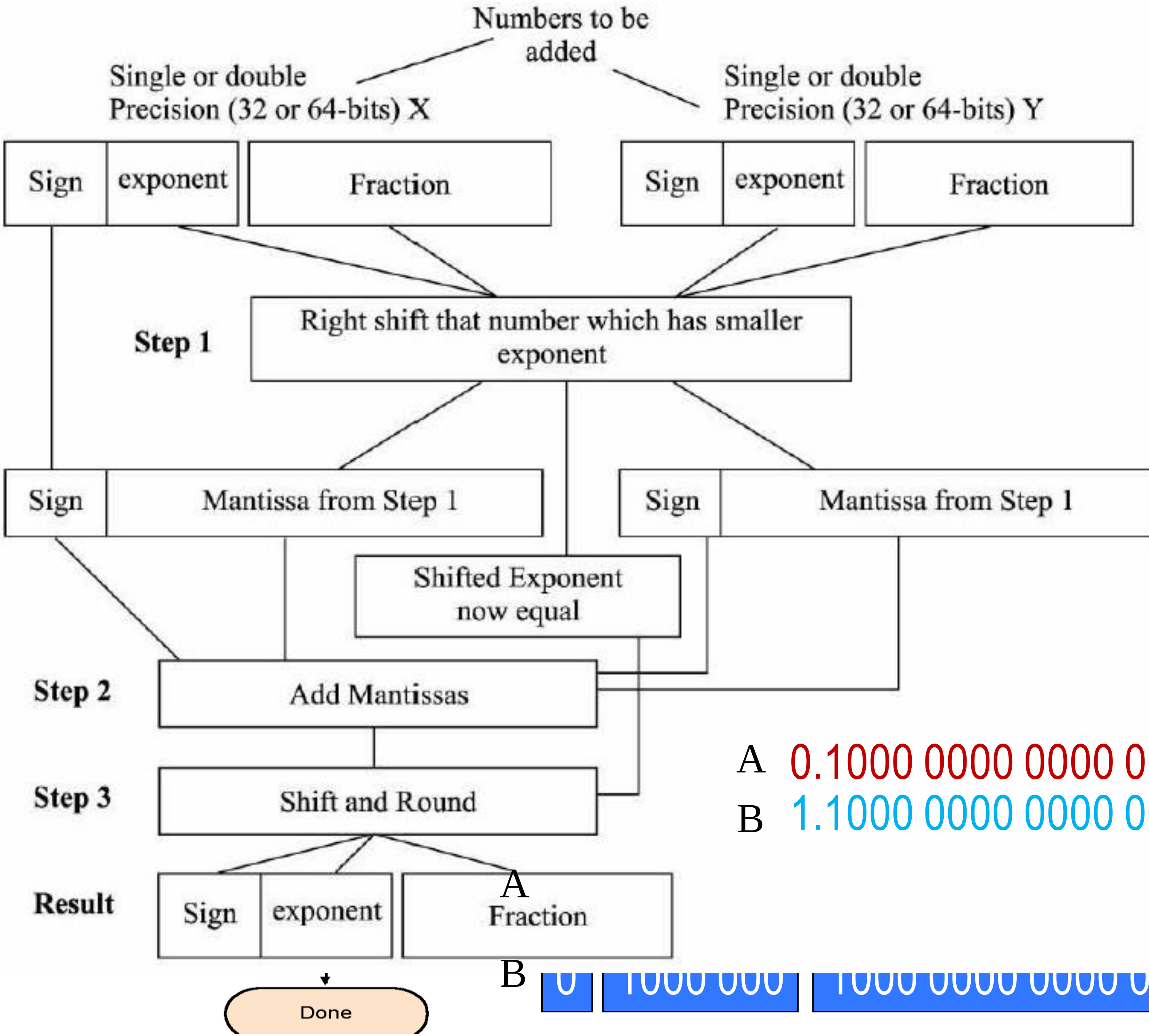
Computer Science and Engineering

Floating-Point Addition

- Consider a 3-digit mantissa binary example
  - $\boxed{1}.000_2 \times 2^{-1} + -\boxed{1}.111_2 \times 2^{-2}$
- 1. Align binary points
  - Shift right the number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$; rightmost 1 is lost
- 2. Add significands (integer addition)
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 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

overflow?
1.111
1.111
+ ------
11.110

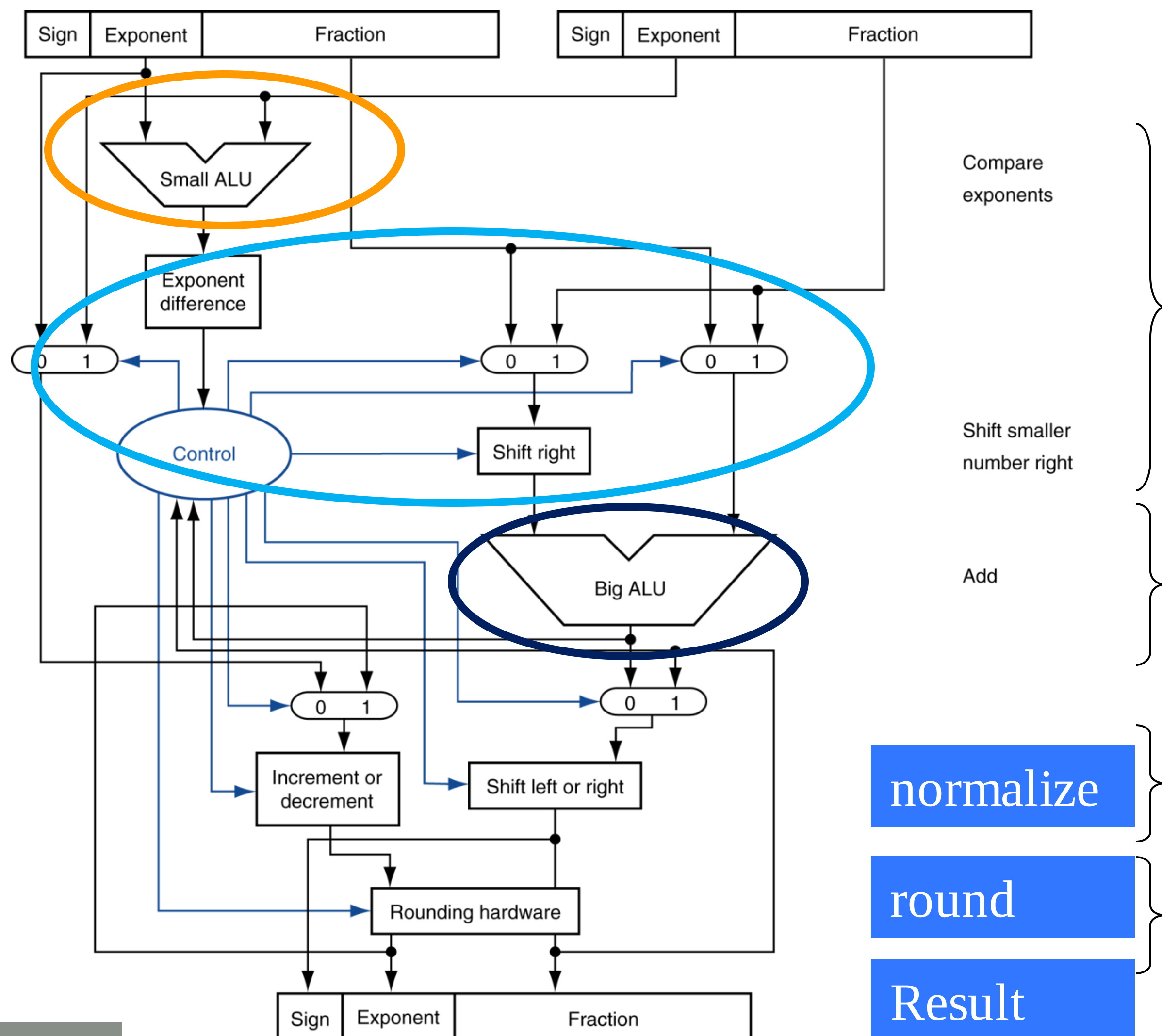
How do you determine the sign of the result?



A 0.1000 0000 0000 0000 0000 000

B 1.1000 0000 0000 0000 0000 000

FP Adder Hardware



Step 1
compare exponents

Step 2
equalizing exponents

Integer addition
Step 3

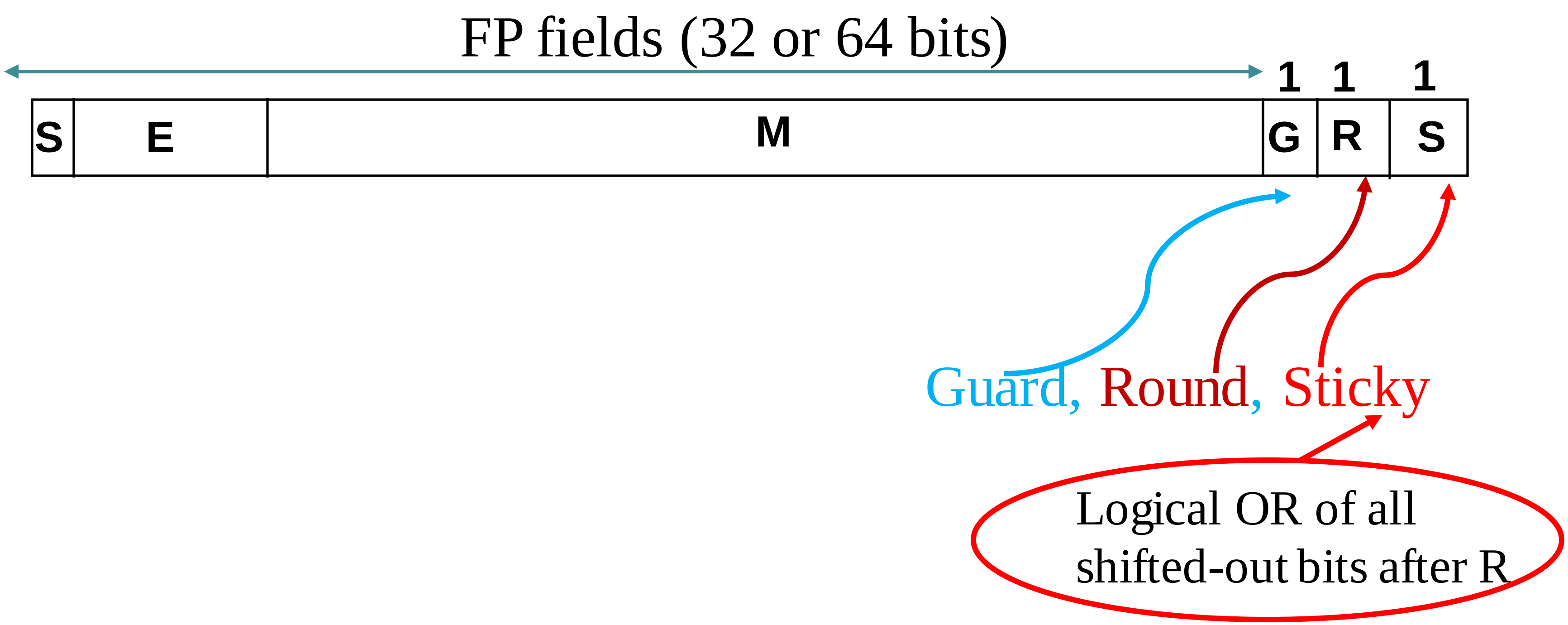
normalize
Step 4

round
Step 5

Result

Three Extra Bits for Internal Use

- Three extra bits are added called Guard, Round, Sticky
  - To achieve accurate arithmetic such as rounding of significand, otherwise some bits would have been lost during right shifts
  - Reduces hardware without compromising precision



Guard Bit

- When we shift bits to the right for alignment, some bits are lost
- We may need to shift the result to the left for normalization after operation
- Storing the lost bits shifted to the right will make results more accurate during normalization
- Round and Sticky bits provide further handles for accurate rounding

Guard, Round, and Sticky Bits

- Two extra bits are needed for rounding
  - Rounding performed after normalizing a result significand
  - Round bit: appears after the guard bit
  - Sticky bit: appears after the round bit (OR of all additional bits)

Guard bit Round bit

OR-reduce

$$
\begin{array}{rlllll}
 & 1.00000000000000000000000 & & & & \times 2^5 \\
- & 0.01111111111111111111111 & 1 & 1 & 00110 & \times 2^5 \text{ (shifted right 7 bits)} \\
\hline
 & 0.10000000000000000000001 & 1 & 1 & 1 & \times 2^5 \text{ (sum)} \\
 & 1.00000000000000000000000 & 1 & 1 & 1 & \times 2^4 \text{ (normalized)}
\end{array}
$$

Round bit Sticky bit

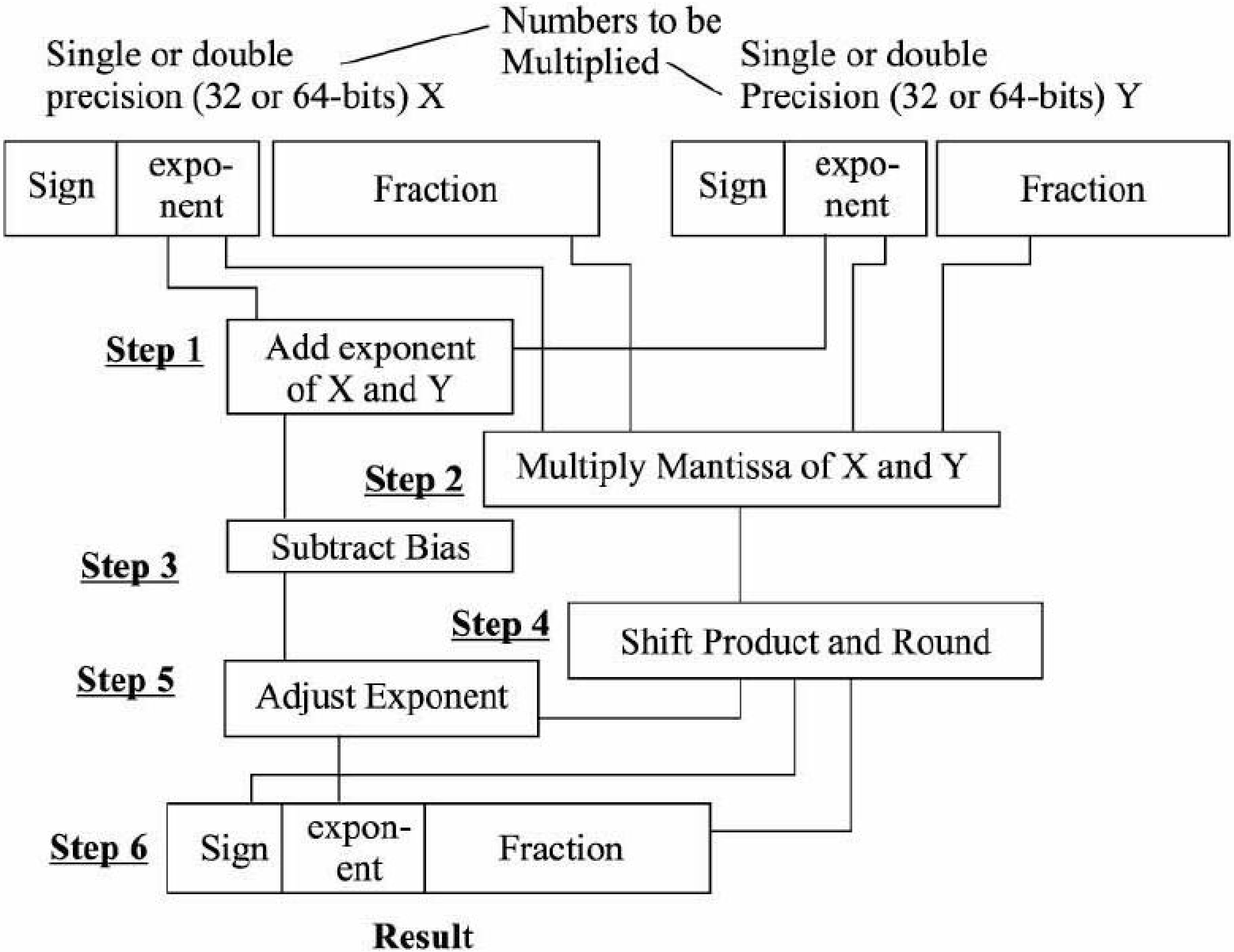
$0.11111111111111111111110 \times 2^4$

$1.11111111111111111111110 \times 2^3$

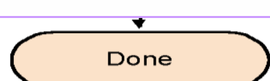
Floating-Point Multiplication

- Now consider a 3-digit mantissa 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: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
- 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
  - $-1.110_2 \times 2^{-3} = -0.21875$

on



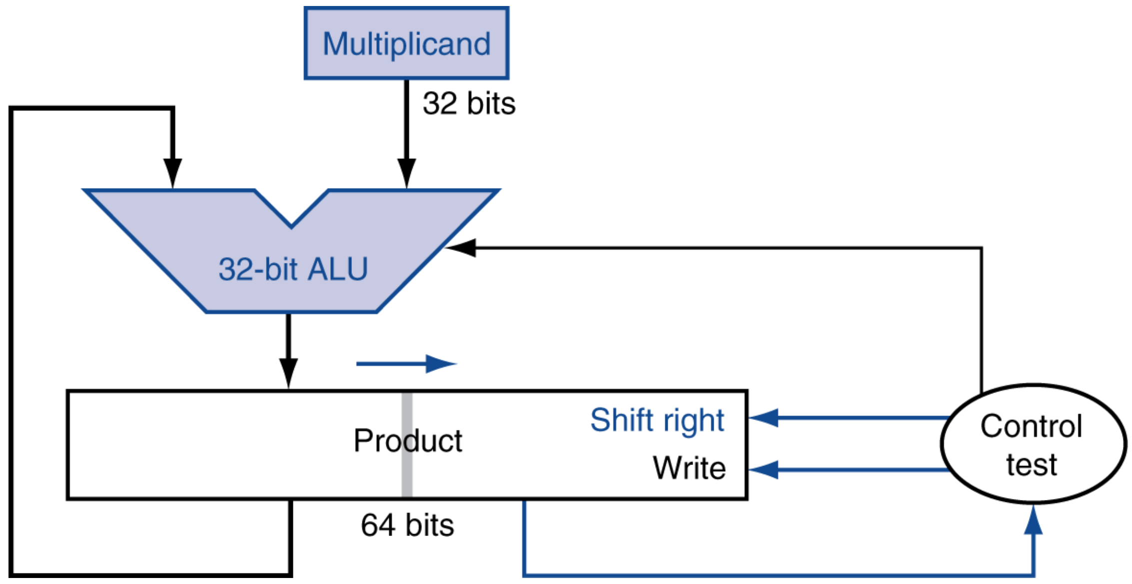
Numbers to be Multiplied
Single or double precision (32 or 64-bits) X
Single or double Precision (32 or 64-bits) Y
Sign
expo-nent
Fraction
Sign
expo-nent
Fraction
Step 1
Add exponent of X and Y
Step 2
Multiply Mantissa of X and Y
Step 3
Subtract Bias
Step 4
Shift Product and Round
Step 5
Adjust Exponent
Step 6
Sign
expon-ent
Fraction
Result



Done

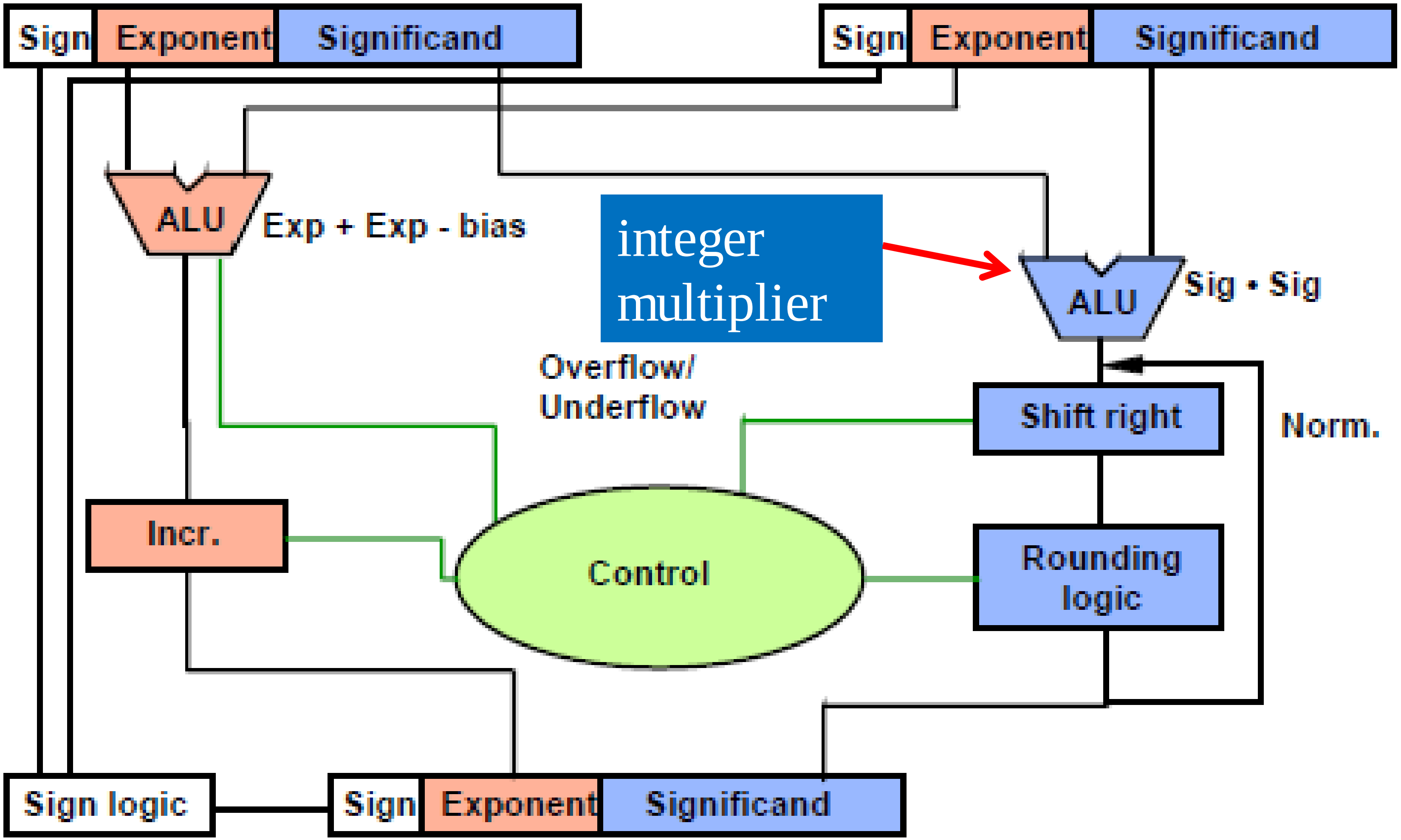
Optimized Integer Multiplier

- Perform steps in parallel: add/shift



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

FP Multiplier – Hardware Realization



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 MIPS

- FP hardware is Co-processor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg's
- 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
  - `lwc1`, `swc1`,
    - e.g., `lwc1 $f8, 32($sp)`

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
- Branch on FP condition code true or false

FP Example: °F to °C

- C code:

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

  - fahr in $f12, result in $f0, literals in global memory space

- Compiled MIPS code:

```
f2c: lwc1  $f16, const5($gp)
     lwc2  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

Summary

- Computer arithmetic is constrained by limited precision
- Bit patterns have no inherent meaning but standards do exist
  - two's complement
  - IEEE 754 floating point
- Computer instructions determine "meaning" of the bit patterns
- Performance and accuracy are important so there are many complexities and implementation challenges in real machines

Problem: Compute $z = \lfloor s/9 \rfloor$ using only integer addition/subtraction, and shift; s: integer

Division operation not allowed

$$z = \frac{s}{9}$$

Hardware requirement:
Only shifter and integer adder;
Iterate a few times to obtain a close approximation!

z

$$\bar{z} = \frac{s - \frac{s - \frac{s - \frac{s}{8} \cdots \text{to } \infty}{8}}{8}}{8}$$

$z = (s - z)/8$

$\Rightarrow 8z = s - z$

$\Rightarrow z = s/9$