Chapter 3:
Arithmetic for Computers

Ngo Lam Trung

[with materials from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK
and M.J. Irwin's presentation, PSU 2008]

Content

- Integer arithmetic
- Floating point number representation and arithmetic

Addition and subtraction

- Addition
  - Similar to what you do to add two numbers manually
  - Digits are added bit by bit from right to left
  - Carries passed to the next digit to the left
- Subtraction
  - Negate the second operand then add to the first operand

$$
\begin{array}{rll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} & = 7_{ten} \\
+ & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} & = 6_{ten} \\
\hline
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two} & = 13_{ten}
\end{array}
$$

Examples

- All numbers are 8-bit signed integer

```
12 + 8 =
```

```
122 + 8 =
```

```
122 + 80 =
```

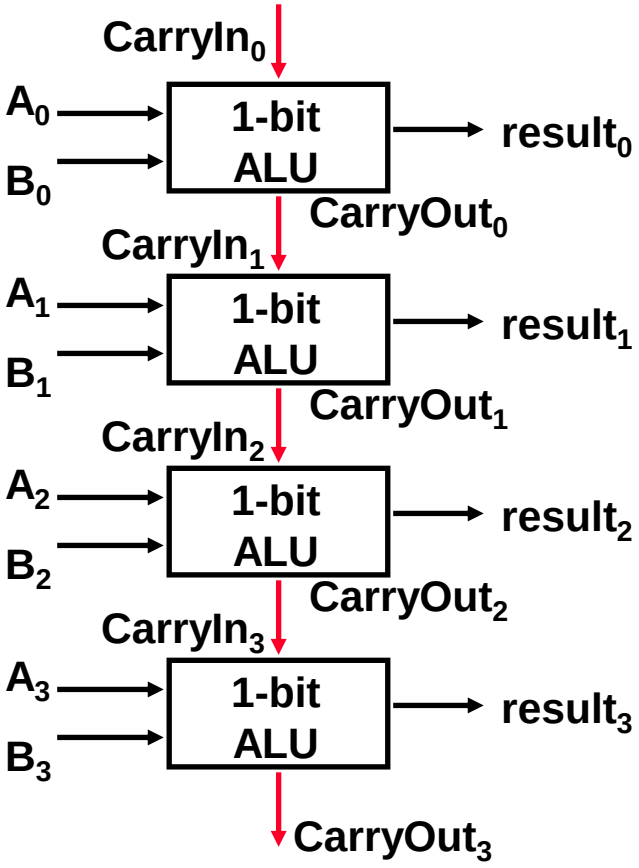
Dealing with Overflow

- **Overflow occurs when the result of an operation cannot be represented in 32-bits, i.e., when the sign bit contains a value bit of the result and not the proper sign bit**
  - **When adding operands with different signs or when subtracting operands with the same sign, overflow can *never* occur**

Operation	Operand A	Operand B	Result indicating overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

Adder implementation

- N-bit ripple-carry adder



Performance depends on data length

➔ Performance is low

Making addition faster: infinite hardware

- Parallelize the adder with the cost of hardware
- Given the addition:

$$a_{n\_1}a_{n\_2}\ldots a_1a_0 + bn_{\_1}b_{n\_2}\ldots b_1b_0$$

- Let c_i is the carry at bit i

$$c2 = (b1 . c1) + (a1 . c1) + (a1 . b1)$$
$$c1 = (b0 . C0) + (a0 . c0) + (a0 . b0)$$

Find c2 from a0, b0, a1, b1?

Making addition faster: Carry Look-ahead

- Approach
  - Make hardwired 4 bit adder → fast and simple enough
  - Develop a carry look-ahead unit to calculate the carry bit before finishing the addition
- At bit i

$$ci + 1 = (bi \cdot ci) + (ai \cdot ci) + (ai \cdot bi)$$
$$= (ai \cdot bi) + (ai + bi) \cdot ci$$

- Denote

$$gi = ai \cdot bi$$
$$pi = ai + bi$$

- Then

$$ci + 1 = gi + pi \cdot ci$$

Carry look-ahead

- With 4-bit adder

$$c1 = g0 + (p0 \cdot c0)$$

$$c2 = g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0)$$

$$c3 = g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0)$$

$$c4 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)$$

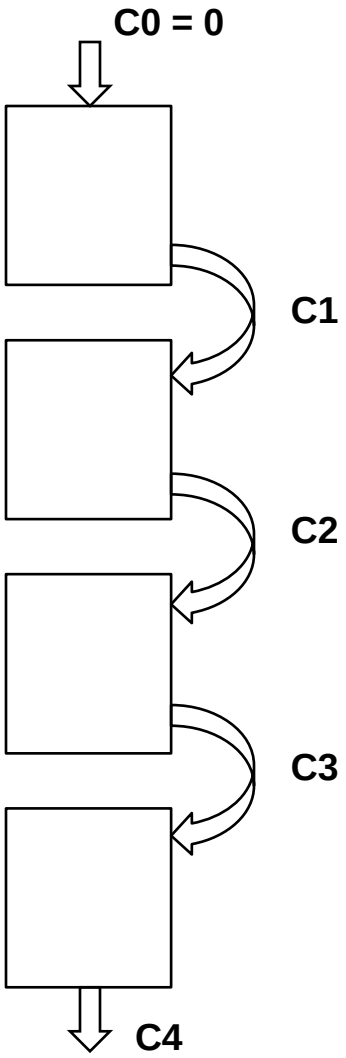
➔ All carry bits can be calculated after 3 gate delay

➔ All result bits can be calculated after maximum of 4 gate delay

➔ How to implement bigger adder?

Carry look-ahead

- For 16-bit adder → fast C1, C2, C3, C4 is needed



Carry look-ahead

- Denote

$$P0 = p3 \cdot p2 \cdot p1 \cdot p0 \qquad G0 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0)$$

$$P1 = p7 \cdot p6 \cdot p5 \cdot p4 \qquad G1 = g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4)$$

$$P2 = p11 \cdot p10 \cdot p9 \cdot p8 \qquad G2 = g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8)$$

$$P3 = p15 \cdot p14 \cdot p13 \cdot p12 \qquad G3 = g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)$$

- Then big-carry bits can be calculated fast

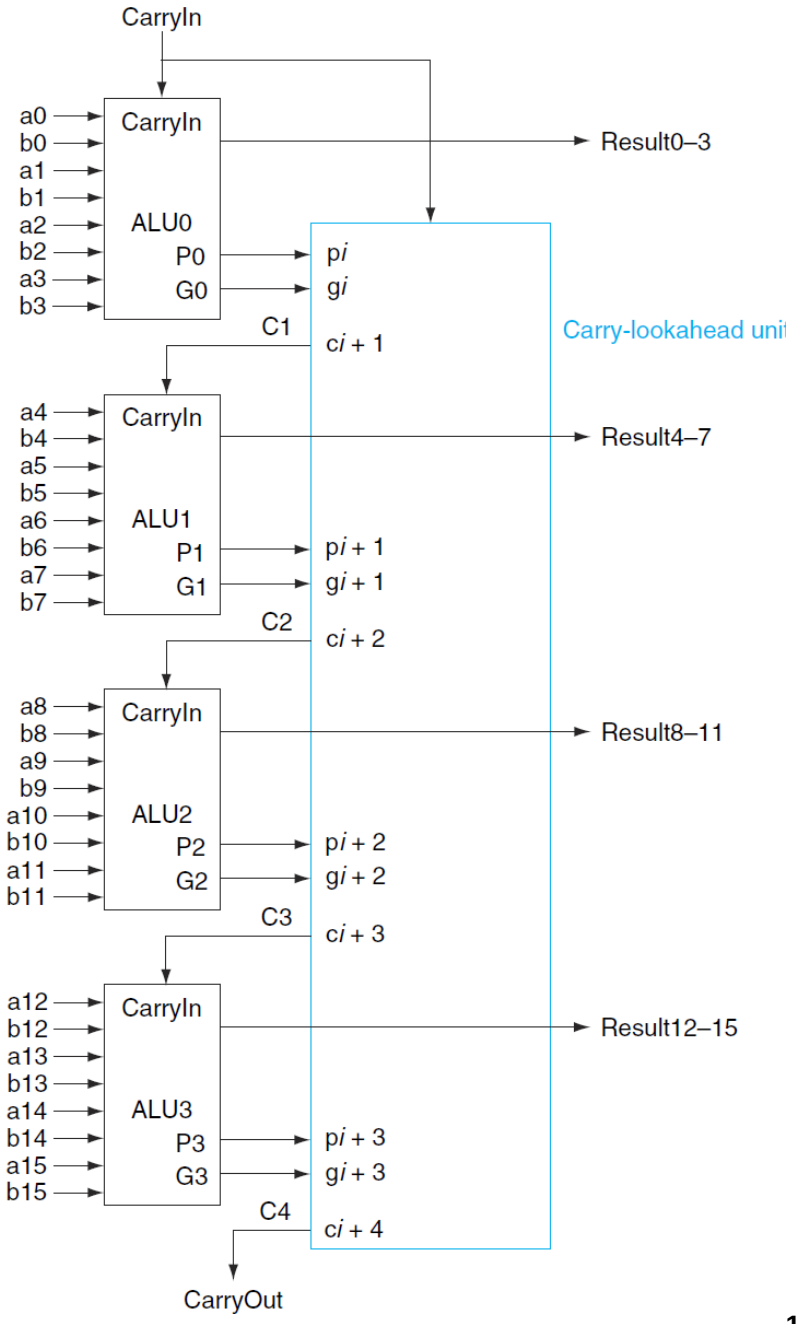
$$C1 = G0 + (P0 \cdot c0)$$

$$C2 = G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0)$$

$$C3 = G2 + (P2 \cdot G1) + (P2 \cdot P1 \cdot G0) + (P2 \cdot P1 \cdot P0 \cdot c0)$$

$$C4 = G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)$$

16-bit Adder


CarryIn
a0
b0
a1
b1
a2
b2
a3
b3
CarryIn
ALU0
P0
G0
C1
pi
gi
ci + 1
Carry-lookahead unit
Result0–3
a4
b4
a5
b5
a6
b6
a7
b7
CarryIn
ALU1
P1
G1
C2
pi + 1
gi + 1
ci + 2
Result4–7
a8
b8
a9
b9
a10
b10
a11
b11
CarryIn
ALU2
P2
G2
C3
pi + 2
gi + 2
ci + 3
Result8–11
a12
b12
a13
b13
a14
b14
a15
b15
CarryIn
ALU3
P3
G3
C4
pi + 3
gi + 3
ci + 4
Result12–15
CarryOut

Exercise

- Dertermine g_i, pi, G_i, Pi when adding the two 16-bit numbers

$$a = 0001\ 1010\ 0011\ 0011$$
$$b = 1110\ 0101\ 1110\ 1011$$

- Calculate c_{15}

Exercise

- p_i, g_i

$$gi = ai \cdot bi$$
$$pi = ai + bi$$

a:	0001	1010	0011	0011
b:	1110	0101	1110	1011
g*i*:	0000	0000	0010	0011
p*i*:	1111	1111	1111	1011

$$P3 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P2 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P1 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$
$$P0 = 1 \cdot 0 \cdot 1 \cdot 1 = 0$$

$$\begin{aligned} G0 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\ &= 0 + (1 \cdot 0) + (1 \cdot 0 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 1) = 0 + 0 + 0 + 0 = 0 \\ G1 &= g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 1 + 0 = 1 \\ G2 &= g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \\ G3 &= g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \end{aligned}$$

Exercise

- c_{15} is actually C_4

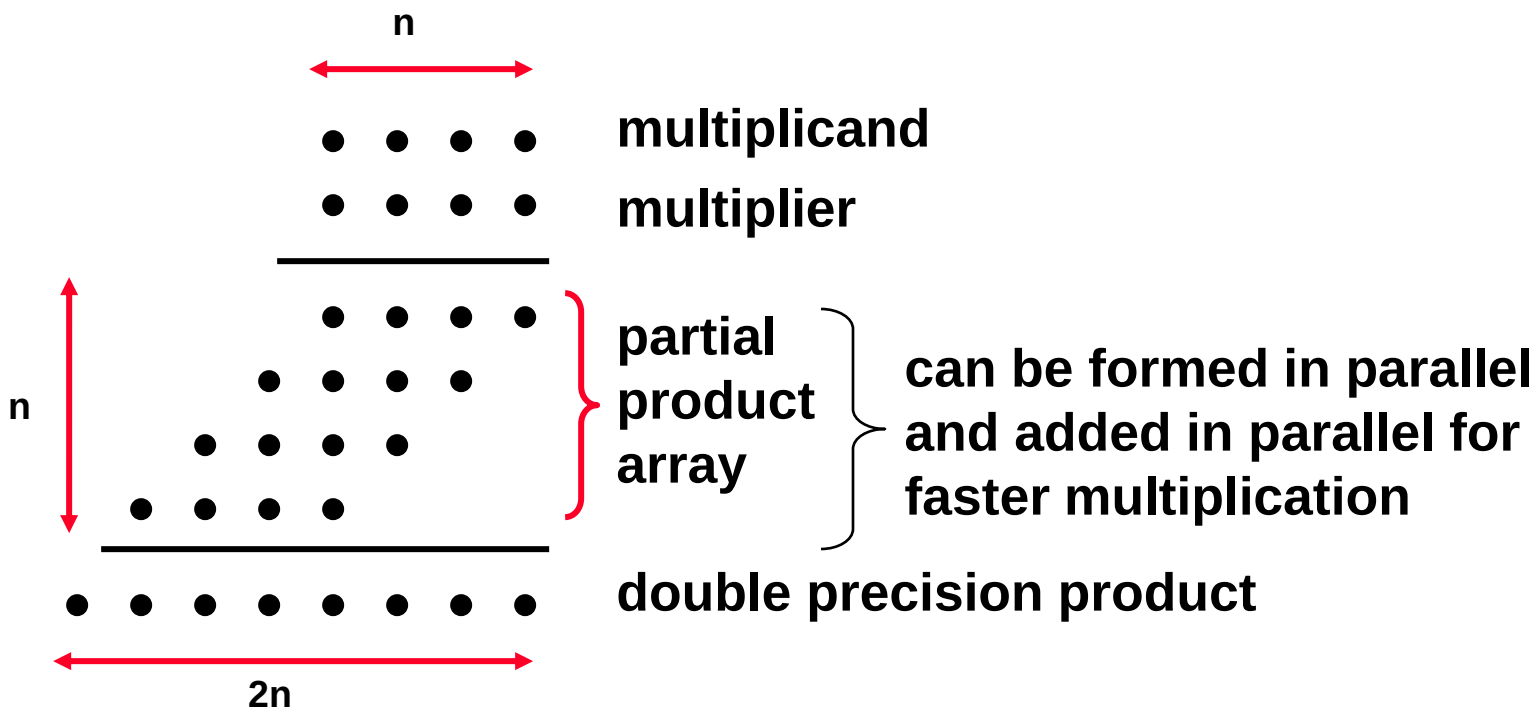
$$
\begin{aligned}
C4 &= G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) \\
&\quad + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0) \\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0) \\
&= 0 + 0 + 1 + 0 + 0 = 1
\end{aligned}
$$

Exercise

- Compare performance of 16-bit ripple carry and 16-bit carry look-ahead adders, assuming delay of all logic gates are equal?

Multiply

- Binary multiplication is just a *bunch* of right shifts and adds

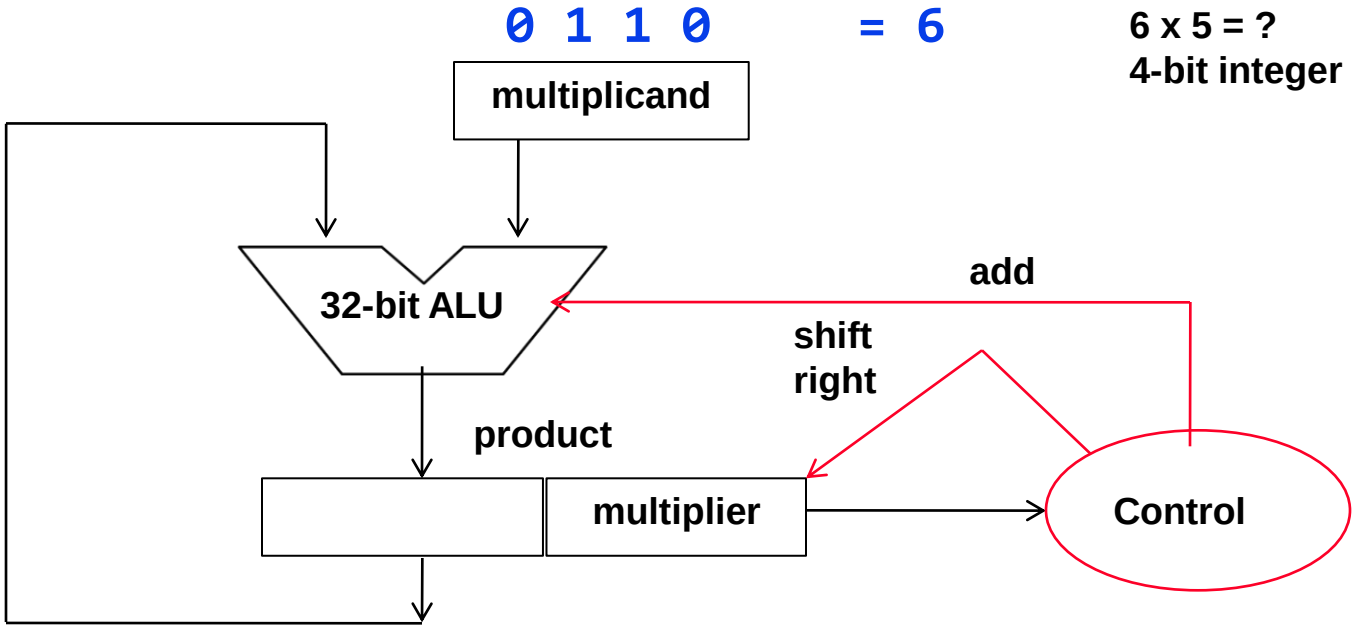


n-bit multiplicand and multiplier → 2n-bit product

Example

Multiplicand		1000_{ten}
Multiplier	x	1001_{ten}
		1000
		0000
		0000
		1000
Product		1001000_{ten}

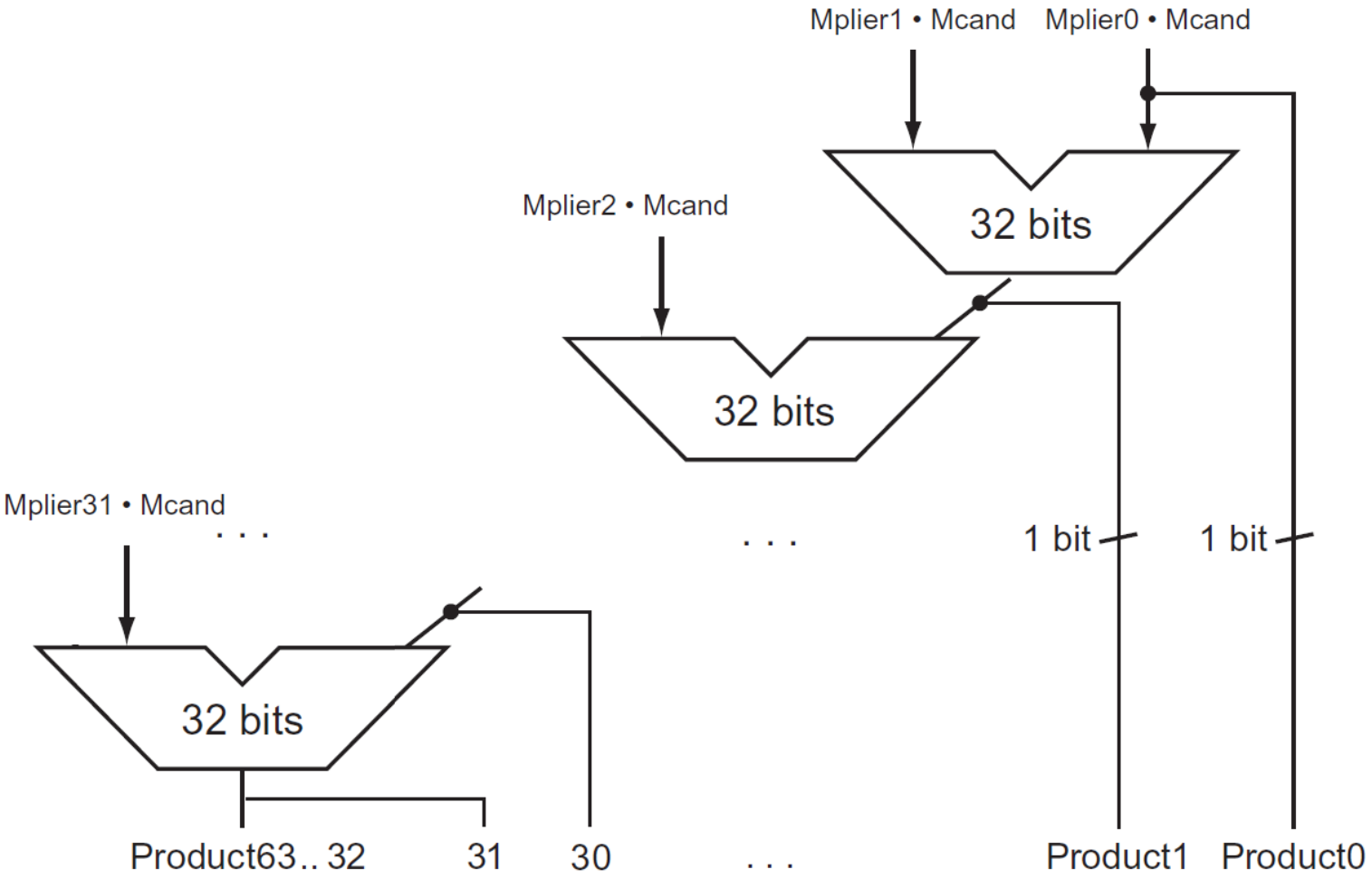
Add and Right Shift Multiplier Hardware



	0 0 0 0 0 1 0 1	= 5
add	0 1 1 0 0 1 0 1	LSB=1 → add multiplicand
	0 0 1 1 → 0 0 1 0	shift right
add	0 0 1 1 0 0 1 0	LSB=0 → no change
	0 0 0 1 → 1 0 0 1	shift right
add	0 1 1 1 1 0 0 1	LSB=1 → add multiplicand
	0 0 1 1 → 1 1 0 0	shift right
add	0 0 1 1 1 1 0 0	LSB=0 → no change
	0 0 0 1 → 1 1 1 0	shift right = 30

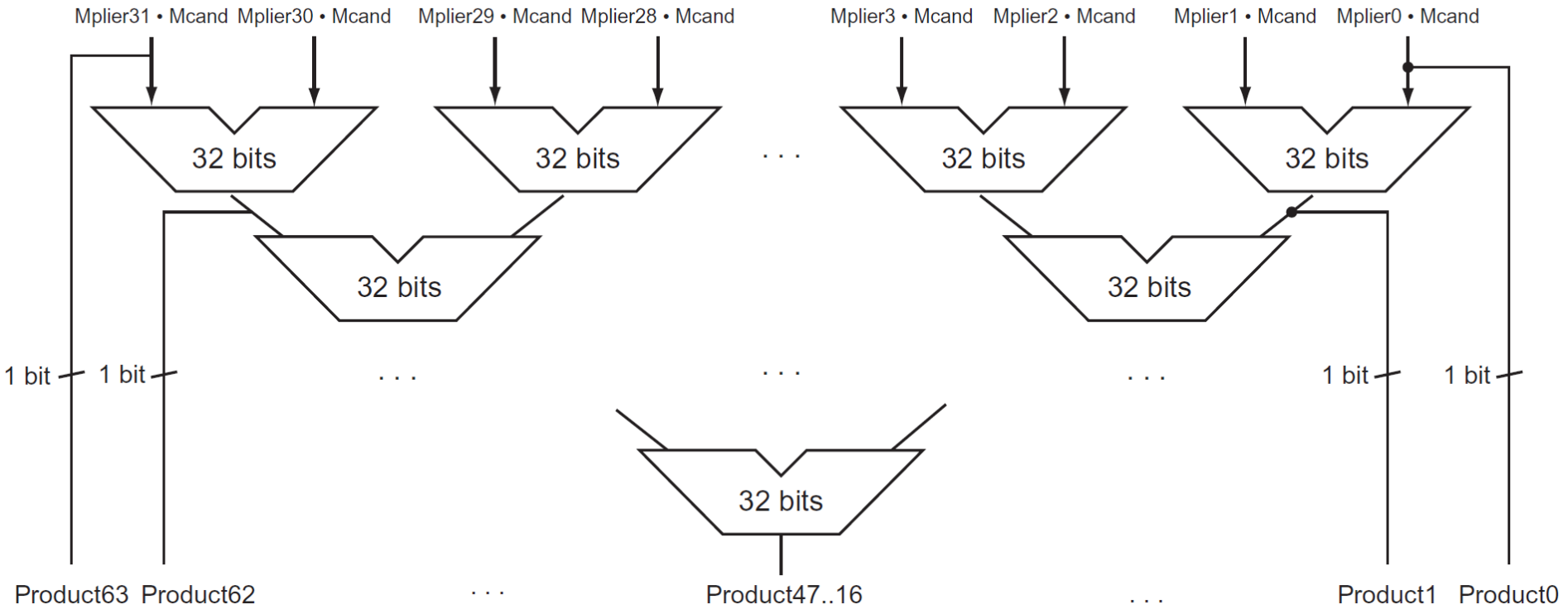
Fast multiplier – Design for Moore

- Why is this fast?



Fast multiplier – Design for Moore

- How fast is this?
- Anything wrong?



MIPS Multiply Instruction

- Multiply (`mult` and `multu`) produces a double precision product (2 x 32 bit)

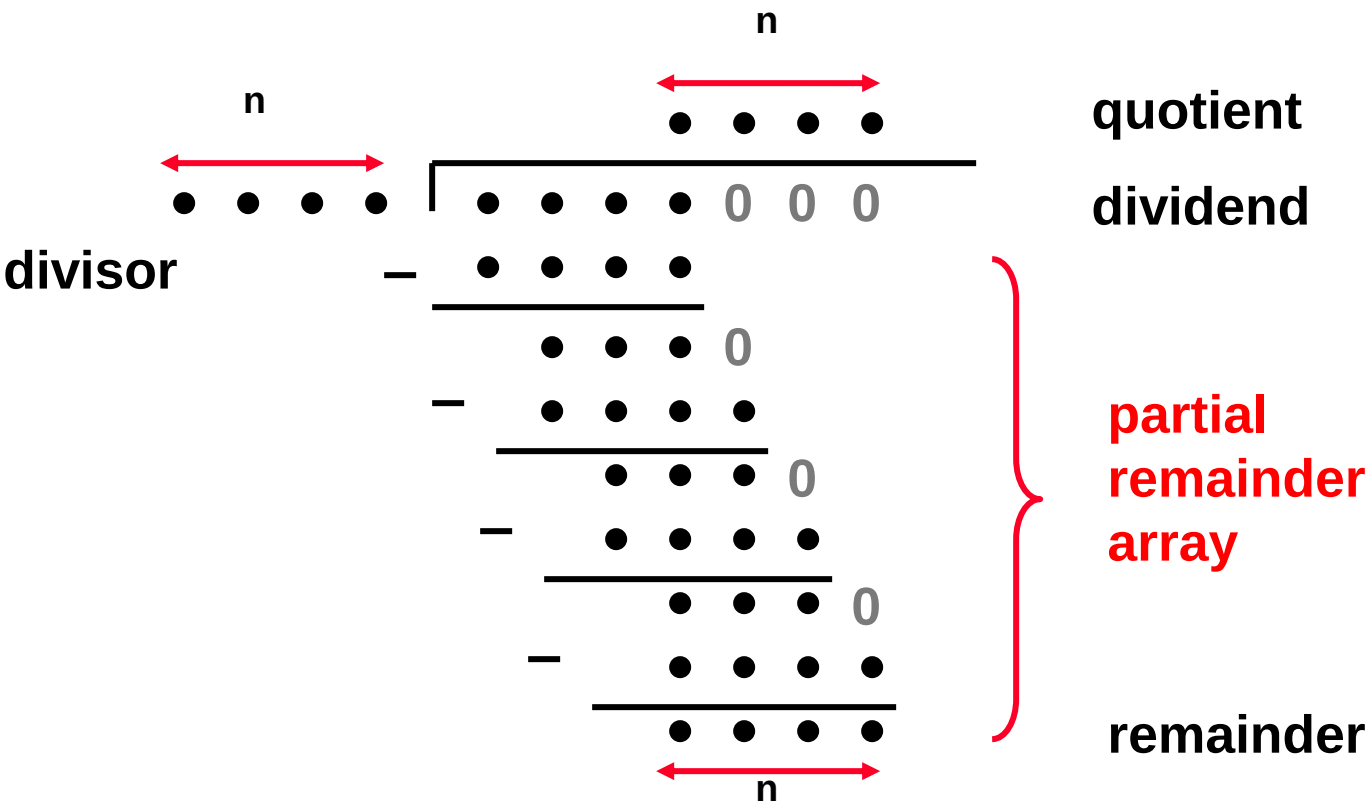
```
mult   $s0, $s1        # hi||lo = $s0 * $s1
```

  - Two additional registers: **hi** and **lo**
  - Low-order word of the product is stored in processor register `lo` and the high-order word is stored in register `hi`
  - Instructions `mfhi rd` and `mflo rd` are provided to move the product to (user accessible) registers in the register file

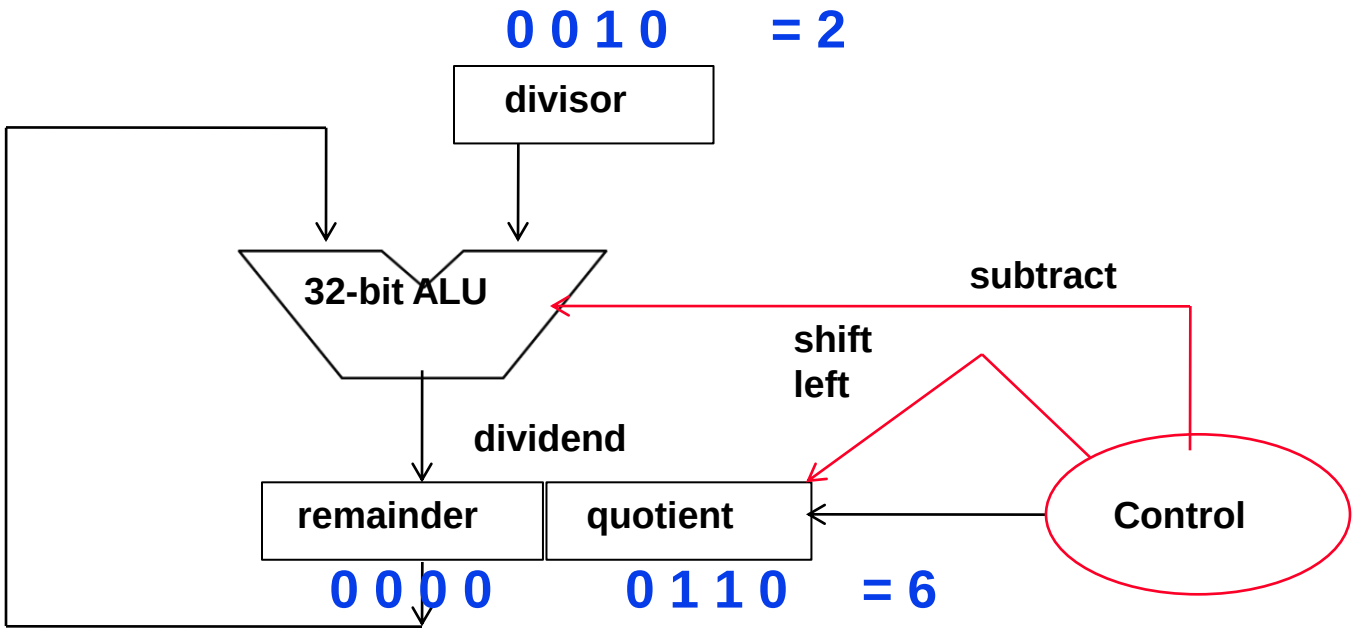
Division

- Division is just a *bunch* of quotient digit guesses and left shifts and subtracts

dividend = quotient x divisor + remainder



Left Shift and Subtract Division Hardware



0 0 0 0	← 1 1 0 0	
sub 1 1 1 0	1 1 0 0	rem neg, so ‘ient bit = 0
0 0 0 0	1 1 0 0	restore remainder
0 0 0 1	← 1 0 0 0	
sub 1 1 1 1	1 1 0 0	rem neg, so ‘ient bit = 0
0 0 0 1	1 0 0 0	restore remainder
0 0 1 1	← 0 0 0 0	
sub 0 0 0 1	0 0 0 1	rem pos, so ‘ient bit = 1
0 0 1 0	← 0 0 1 0	
sub 0 0 0 0	0 0 1 1	rem pos, so ‘ient bit = 1

= 3 with 0 remainder

MIPS Divide Instruction

- Divide (`div` and `divu`) generates the reminder in `hi` and the quotient in `lo`

```
div    $s0, $s1          # lo = $s0 / $s1

                         # hi = $s0 mod $s1
```

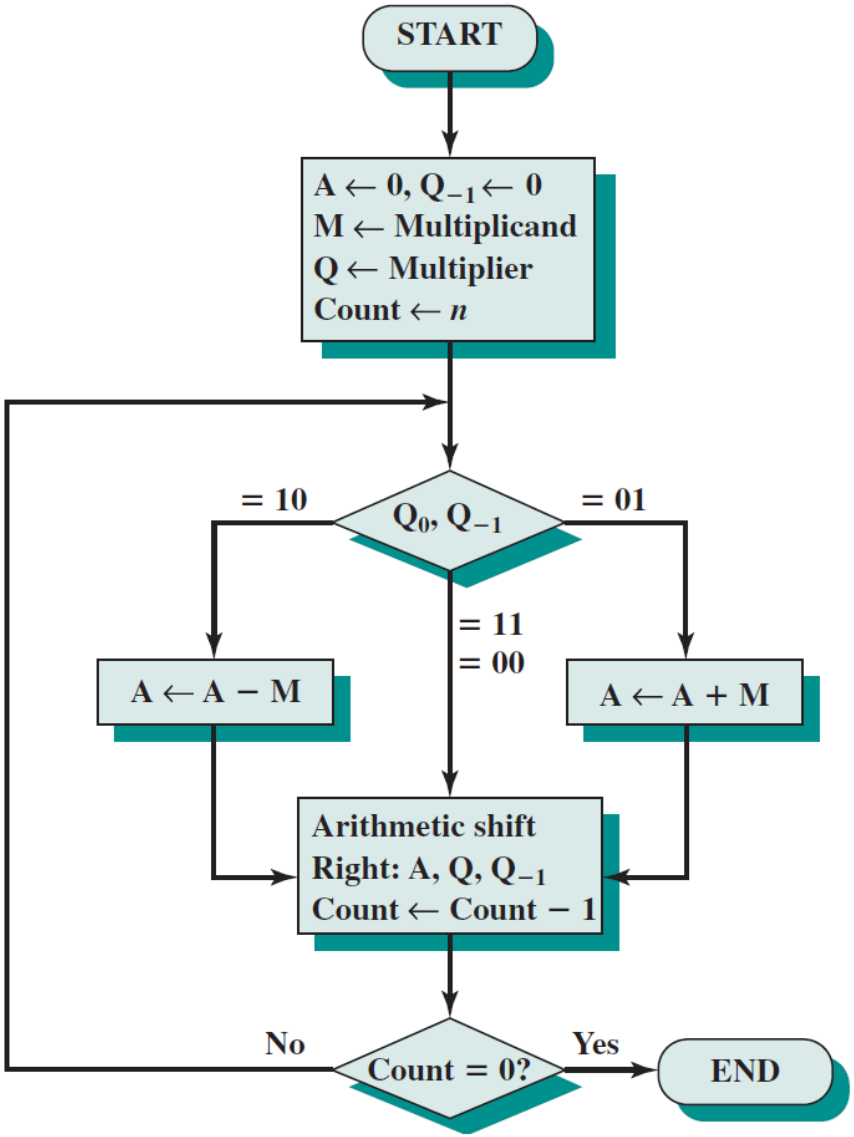
  - Instructions `mfhi rd` and `mflo rd` are provided to move the quotient and reminder to (user accessible) registers in the register file

- **As with multiply, divide ignores overflow so software must determine if the quotient is too large. Software must also check the divisor to avoid division by 0.**

Signed integer multiplication and division

- Reuse unsigned multiplication then fix product sign later
- Multiplication
  - Multiplicand and multiplier are of the same sign: keep product
  - Multiplicand and multiplier are of different sign: negate product
- Division:
  - Dividend and divisor of the same sign:
    - Keep quotient
    - Keep/negate remainder so it is of the same sign with dividend
  - Dividend and divisor of different sign:
    - Negate quotient
    - Keep/negate remainder so it is of the same sign with dividend

Signed integer with Booth algorithm



Representing Big (and Small) Numbers

- Encoding non-integer value?
  - Earth mass: (5.9722±0.0006)×1024 (kg)
  - Weight of an amu (atomic mass unit, 1/12 mass of C_{12})

    0.00000000000000000000000166 or 1.6×10^{-27} (kg)
  - PI number

    PI = 3.14159….
- Problem: how to represent the above numbers?

➔ We need reals or floating-point numbers!

➔ Floating point numbers in decimal:

- ➔ 1000
- ➔ 1×10^3
- ➔ 0.1×10^4

Floating point number

- In decimal system

$$2013.1228 = 201.31228 * 10$$

$$= 20.131228 * 10^{2}$$

$$= 2.0131228 * 10^{3}$$

$$= 20131228 * 10^{-4}$$

- What is the “standard” form?

$$2.0131228 * 10^{3} = 2.0131228E+03$$

mantissa **exponent**

- In binary $X = \pm 1.xxxxx * 2^{yyyy}$
- ***Sign, mantissa, and exponent need to be represented***

Floating point number

- Floating point representation in binary

$$(-1)^{sign} \times 1.F \times 2^{E-bias}$$

  - Still have to fit everything in 32 bits (single precision)
  - Bias = 127 with single precision floating point number

1 sign bit 8 bits 23 bits

- Defined by the IEEE 754-1985 standard
  - Single precision: 32 bit
  - Double precision: 64 bit
  - Correspond to float and double in C

Examples

- Ex1: convert X into decimal value

X = 1100 0001 0101 0110 0000 0000 0000 0000

sign = 1 → X is negative

E = 1000 0010 = 130

F = 10101100...00

→ X = $(-1)^1$ x 1.101011000..00 x $2^{130-127}$

= -1.101011 x 2^3 = -1101.011

= -13.375

Example

- Ex2: find decimal value of X

X = 0011 1111 1000 0000 0000 0000 0000 0000

sign = 0

e = 0111 1111 = 127

m = 000…0000 (23 bit 0)

$X = (-1)^0 \times 1.00\ldots000 \times 2^{127-127} = 1.0$

Example

- Ex3: find binary representation of X = 9.6875 in IEEE 754 single precision

Converting X to plain binary

$9_{10} = 1001_2$

0.6875 x 2 = 1.375 → get bit 1
0.375 x 2 = 0.75 → get bit 0
0.75 x 2 = 1.5 → get bit 1
0.5 x = 1.0 → get bit 1

➔ $9.6875_{10} = 1001.1011_2$

Example

- Ex3: find binary representation of X = 9.6875 in IEEE 754 single precision

$X = 9.6875_{(10)} = 1001.1011_{(2)} = 1.0011011 \times 2^3$

Then

S = 0

$e = 127 + 3 = 130_{(10)} = 1000\ 0010_{(2)}$

m = 001101100...00 (23 bit)

Finally

X = 0100 0001 0001 1011 0000 0000 0000 0000

Examples

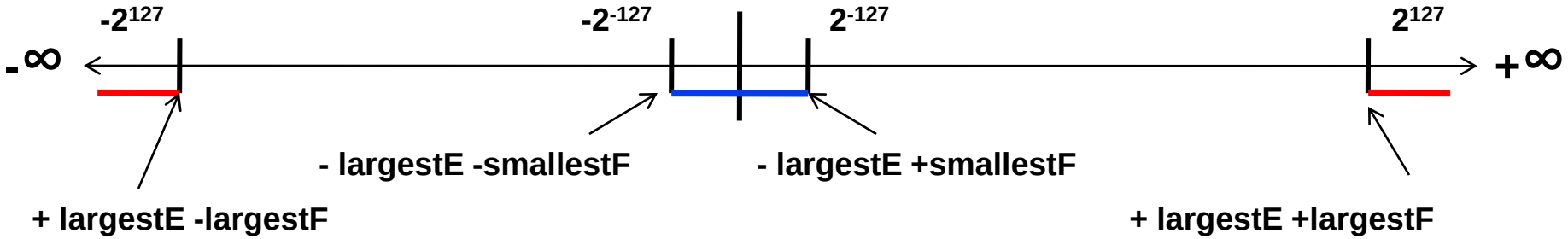
- $1.0_2 \times 2^{-1} =$
- $100.75_{10} =$

Some special values

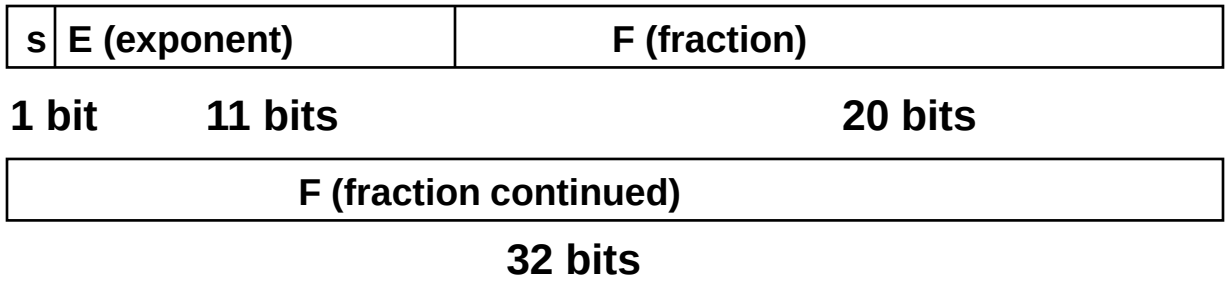
- Smallest+: 0 00000001 1.00000000000000000000000
  = $1 \times 2^{1-127}$
- Zero: 0 00000000 00000000000000000000000
  = true 0
- Largest+: 0 11111110 1.11111111111111111111111
  = $(2-2^{-23}) \times 2^{254-127}$

Too large or too small values

- Overflow (floating point) happens when a positive exponent becomes too large to fit in the exponent field
- Underflow (floating point) happens when a negative exponent becomes too large to fit in the exponent field



- **Reduce the chance of underflow or overflow is to offer another format that has a larger exponent field**
  - **Double precision – takes two MIPS words**



Reduce underflow with the same bit length?

- De-normalized number

IEEE 754 FP Standard Encoding

- Special encodings are used to represent unusual events
  - ± infinity for division by zero
  - NAN (not a number) for invalid operations such as 0/0
  - True zero is the bit string all zero

Single Precision		Double Precision		Object Represented
E (8)	F (23)	E (11)	F (52)	
0000 0000	0	0000 … 0000	0	true zero (0)
0000 0000	nonzero	0000 … 0000	nonzero	± denormalized number
0111 1111 to +127,-126	anything	0111 …1111 to +1023,-1022	anything	± floating point number
1111 1111	+ 0	1111 … 1111	- 0	± infinity
1111 1111	nonzero	1111 … 1111	nonzero	not a number (NaN)

Floating Point Addition

- Addition (and subtraction)

$$(\pm F1 \times 2^{E1}) + (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$$

  - **Step 0: Restore the hidden bit in F1 and in F2**
  - **Step 1: Align fractions by right shifting F2 by E1 - E2 positions (assuming E1 ≥ E2) keeping track of (three of) the bits shifted out in G R and S**
  - **Step 2: Add the resulting F2 to F1 to form F3**
  - **Step 3: Normalize F3 (so it is in the form 1.XXXXX …)**
    - **If F1 and F2 have the same sign → F3 ∈[1,4) → 1 bit right shift F3 and increment E3 (check for overflow)**
    - **If F1 and F2 have different signs → F3 may require *many* left shifts each time decrementing E3 (check for underflow)**
  - **Step 4: Round F3 and possibly normalize F3 again**
  - **Step 5: Rehide the most significant bit of F3 before storing the result**

Floating Point Addition Example

- Add

$$(0.5 = 1.0000 \times 2^{-1}) + (-0.4375 = -1.1100 \times 2^{-2})$$

  - **Step 0:**
  - **Step 1:**
  - **Step 2:**
  - **Step 3:**
  - **Step 4:**
  - **Step 5:**

Floating Point Addition Example

- Add: 0.5 + (-0.4375) = ?

  $(0.5 = 1.0000 \times 2^{-1}) + (-0.4375 = -1.1100 \times 2^{-2})$

  - **Step 0:** **Hidden bits restored in the representation above**
  - **Step 1:** **Shift significand with the smaller exponent (1.1100) right until its exponent matches the larger exponent (so once)**
  - **Step 2:** **Add significands**

    $1.0000 + (-0.111) = 1.0000 - 0.111 = 0.001$

  - **Step 3:** **Normalize the sum, checking for exponent over/underflow**

    $0.001 \times 2^{-1} = 0.010 \times 2^{-2} = .. = 1.000 \times 2^{-4}$
  - **Step 4:** **The sum is already rounded, so we're done**
  - **Step 5:** **Re-hide the hidden bit before storing**

Floating Point Multiplication

- Multiplication

$$(\pm F1 \times 2^{E1}) \times (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$$

  - **Step 0: Restore the hidden bit in F1 and in F2**
  - **Step 1: Add the two (biased) exponents and subtract the bias from the sum, so E1 + E2 – 127 = E3**

    also determine the sign of the product (which depends on the sign of the operands (most significant bits))
  - **Step 2: Multiply F1 by F2 to form a double precision F3**
  - **Step 3: Normalize F3 (so it is in the form 1.XXXXX …)**
    - **Since F1 and F2 come in normalized → F3 ∈[1,4) → 1 bit right shift F3 and increment E3**
    - **Check for overflow/underflow**
  - **Step 4: Round F3 and possibly normalize F3 again**
  - **Step 5: Rehide the most significant bit of F3 before storing the result**

Floating Point Multiplication Example

- Multiply

$$(0.5 = 1.0000 \times 2^{-1}) \text{ x } (-0.4375 = -1.1100 \times 2^{-2})$$

  - **Step 0:**
  - **Step 1:**
  - **Step 2:**
  - **Step 3:**
  - **Step 4:**
  - **Step 5:**

Floating Point Multiplication Example

- Multiply

$$(0.5 = 1.0000 \times 2^{-1}) \times (-0.4375 = -1.1100 \times 2^{-2})$$

- **Step 0:** **Hidden bits restored in the representation above**
- **Step 1:** **Add the exponents (not in bias would be -1 + (-2) = -3 and in bias would be (-1+127) + (-2+127) – 127 = (-1 -2) + (127+127-127) = -3 + 127 = 124**
- **Step 2:** **Multiply the significands**
  1.0000 x 1.110 = 1.110000
- **Step 3:** **Normalized the product, checking for exp over/underflow**
  1.110000×2^{-3} is already normalized
- **Step 4:** **The product is already rounded, so we're done**
- **Step 5:** **Rehide the hidden bit before storing**

Support for Accurate Arithmetic

- **IEEE 754 FP rounding modes**
  - **Always round up (toward +∞)**
  - **Always round down (toward -∞)**
  - **Truncate**
  - **Round to nearest even (when the Guard || Round || Sticky are 100) – always creates a 0 in the least significant (kept) bit of F**
- Rounding (except for truncation) requires the hardware to include extra F bits during calculations
  - Guard bit – used to provide one F bit when shifting left to normalize a result (e.g., when normalizing F after division or subtraction)
  - Round bit – used to improve rounding accuracy
  - Sticky bit – used to support Round to nearest even; is set to a 1 whenever a 1 bit shifts (right) through it (e.g., when aligning F during addition/subtraction)

F = 1 . xxxxxxxxxxxxxxxxxxxxxxx G R S

http://pages.cs.wisc.edu/~markhill/cs354/Fall2008/notes/flpt.apprec.html

Example

- Calculate:

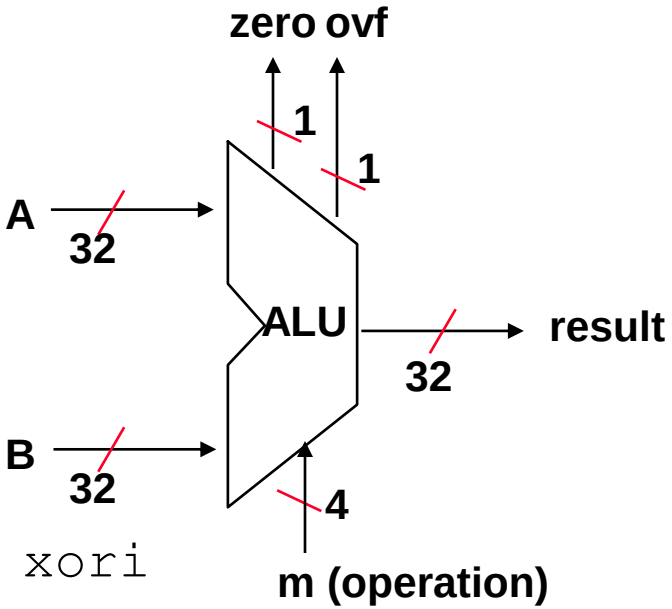
  0.2 x 5 = ?

  0.333 x 3 = ?

  (1.0/3) x 3 = ?

MIPS Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA




```
add, addi, addiu, addu
sub, subu
mult, multu, div, divu
and, andi, nor, or, ori, xor, xori
beq, bne, slt, slti, sltiu, sltu
```

- **With special handling for**
  - **sign extend – `addi, addiu, slti, sltiu`**
  - **zero extend – `andi, ori, xori`**
  - **overflow detection – `add, addi, sub`**