

---

# Chapter 3: Arithmetic for Computers

Ngo Lam Trung

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

# Content

---

- ❑ Integer representation and 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

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

# Examples

---

- ❑ All numbers are 8-bit signed integer

$$12 + 8 =$$

$$122 + 8 =$$

$$122 + 80 =$$

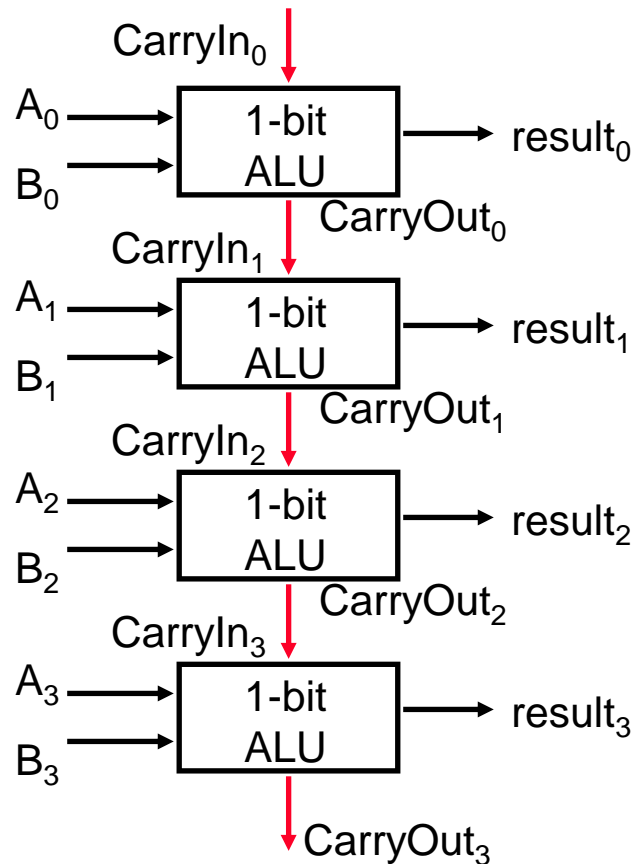
# Dealing with Overflow

- ❑ 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	$\geq 0$	$\geq 0$	$< 0$
A + B	$< 0$	$< 0$	$\geq 0$
A - B	$\geq 0$	$< 0$	$< 0$
A - B	$< 0$	$\geq 0$	$\geq 0$

# 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:

$$a_{n-1}a_{n-2} \dots a_1a_0 + b_{n-1}b_{n-2} \dots b_1b_0$$

❑ Let  $c_i$  is the carry at bit  $i$

$$c_2 = (b_1 \cdot c_1) + (a_1 \cdot c_1) + (a_1 \cdot b_1)$$

$$c_1 = (b_0 \cdot c_0) + (a_0 \cdot c_0) + (a_0 \cdot b_0)$$

Find  $c_2$  from  $a_0, b_0, a_1, b_1$ ?

# Making addition faster: Carry Lookahead

## □ Approach

- Make hardwired 4 bit adder → fast and simple enough
- Develop a carry lookahead unit to calculate the carry bit before finishing the addition

## □ At bit $i$

$$\begin{aligned}c_{i+1} &= (b_i \cdot c_i) + (a_i \cdot c_i) + (a_i \cdot b_i) \\ &= (a_i \cdot b_i) + (a_i + b_i) \cdot c_i\end{aligned}$$

## □ Denote

$$\begin{aligned}g_i &= a_i \cdot b_i \\ p_i &= a_i + b_i\end{aligned}$$

## □ Then

$$c_{i+1} = g_i + p_i \cdot c_i$$



# Carry lookahead

---

## ❑ With 4-bit adder

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

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

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

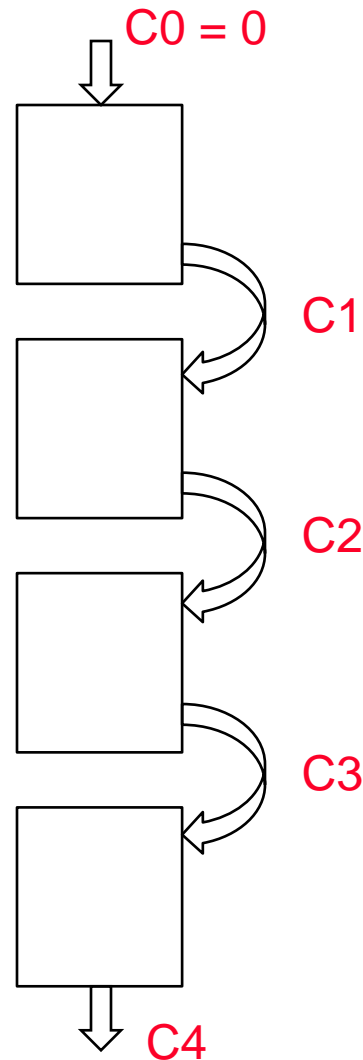
$$c4 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\ + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)$$

- ➔ 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 lookahead

---

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



# Carry lookahead

---

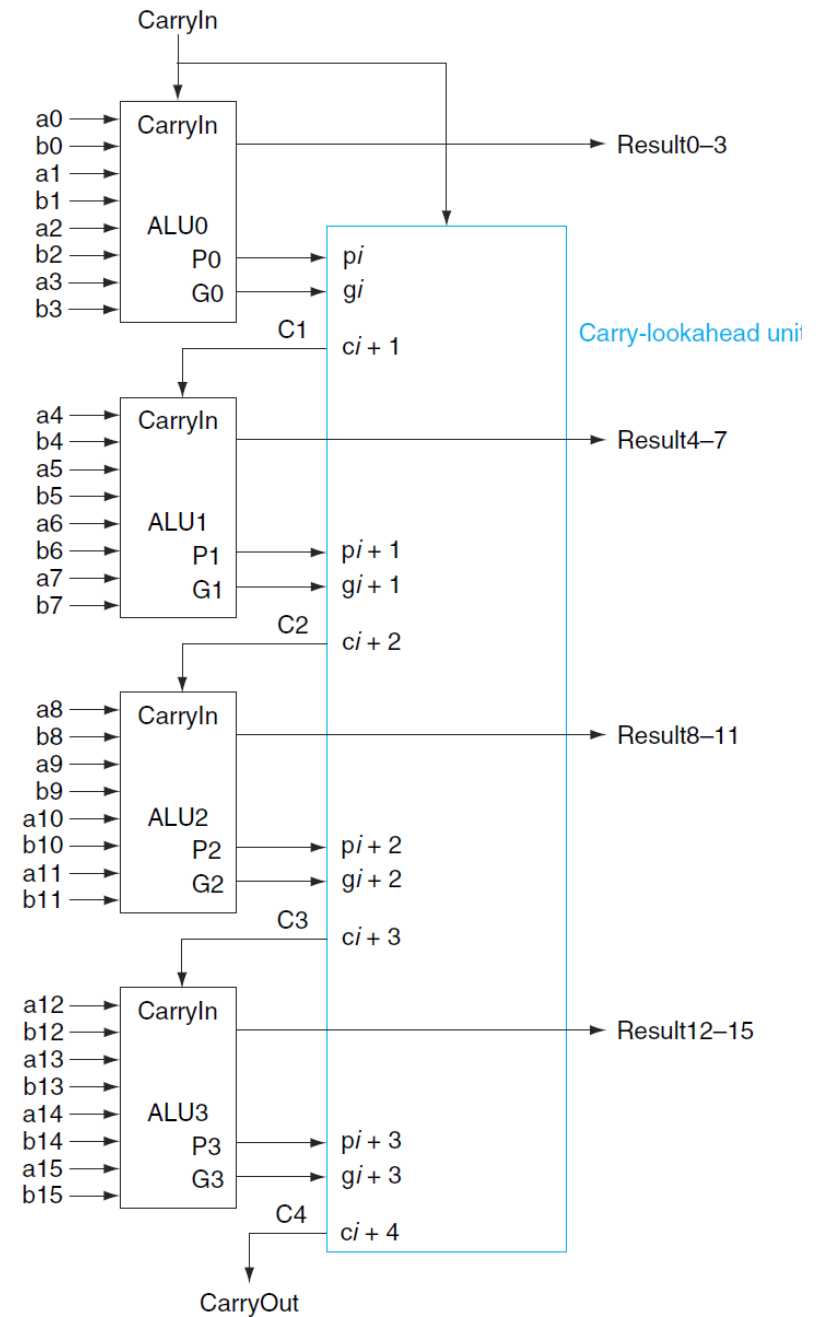
## □ Denote

$$\begin{array}{ll} P_0 = p_3 \cdot p_2 \cdot p_1 \cdot p_0 & G_0 = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) \\ P_1 = p_7 \cdot p_6 \cdot p_5 \cdot p_4 & G_1 = g_7 + (p_7 \cdot g_6) + (p_7 \cdot p_6 \cdot g_5) + (p_7 \cdot p_6 \cdot p_5 \cdot g_4) \\ P_2 = p_{11} \cdot p_{10} \cdot p_9 \cdot p_8 & G_2 = g_{11} + (p_{11} \cdot g_{10}) + (p_{11} \cdot p_{10} \cdot g_9) + (p_{11} \cdot p_{10} \cdot p_9 \cdot g_8) \\ P_3 = p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12} & G_3 = g_{15} + (p_{15} \cdot g_{14}) + (p_{15} \cdot p_{14} \cdot g_{13}) + (p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12}) \end{array}$$

## □ Then big-carry bits can be calculated fast

$$\begin{aligned} C_1 &= G_0 + (P_0 \cdot c_0) \\ C_2 &= G_1 + (P_1 \cdot G_0) + (P_1 \cdot P_0 \cdot c_0) \\ C_3 &= G_2 + (P_2 \cdot G_1) + (P_2 \cdot P_1 \cdot G_0) + (P_2 \cdot P_1 \cdot P_0 \cdot c_0) \\ C_4 &= G_3 + (P_3 \cdot G_2) + (P_3 \cdot P_2 \cdot G_1) + (P_3 \cdot P_2 \cdot P_1 \cdot G_0) \\ &\quad + (P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0) \end{aligned}$$

# 16-bit Adder



## Exercise

---

- ❑ Determine  $g_i, p_i, G_i, P_i$  when adding the two 16-bit numbers

$$a = 0001\ 1010\ 0011\ 0011$$

$$b = 1110\ 0101\ 1110\ 1011$$

- ❑ Calculate  $c_{15}$

## Exercise

□  $p_i, g_i$

$$g_i = a_i \cdot b_i$$

$$p_i = a_i + b_i$$

a :	0001	1010	0011	0011
b :	1110	0101	1110	1011
$g_i$ :	0000	0000	0010	0011
$p_i$ :	1111	1111	1111	1011

$$P_3 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

$$P_2 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

$$P_1 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

$$P_0 = 1 \cdot 0 \cdot 1 \cdot 1 = 0$$

$$\begin{aligned} G_0 &= g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) \\ &= 0 + (1 \cdot 0) + (1 \cdot 0 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 1) = 0 + 0 + 0 + 0 = 0 \end{aligned}$$

$$\begin{aligned} G_1 &= g_7 + (p_7 \cdot g_6) + (p_7 \cdot p_6 \cdot g_5) + (p_7 \cdot p_6 \cdot p_5 \cdot g_4) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 1 + 0 = 1 \end{aligned}$$

$$\begin{aligned} G_2 &= g_{11} + (p_{11} \cdot g_{10}) + (p_{11} \cdot p_{10} \cdot g_9) + (p_{11} \cdot p_{10} \cdot p_9 \cdot g_8) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \end{aligned}$$

$$\begin{aligned} G_3 &= g_{15} + (p_{15} \cdot g_{14}) + (p_{15} \cdot p_{14} \cdot g_{13}) + (p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12}) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \end{aligned}$$

## Exercise

---

□  $c_{15}$  is actually  $C_4$

$$\begin{aligned} C_4 &= G_3 + (P_3 \cdot G_2) + (P_3 \cdot P_2 \cdot G_1) + (P_3 \cdot P_2 \cdot P_1 \cdot G_0) \\ &\quad + (P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0) \\ &= 0 + 0 + 1 + 0 + 0 = 1 \end{aligned}$$

## Exercise

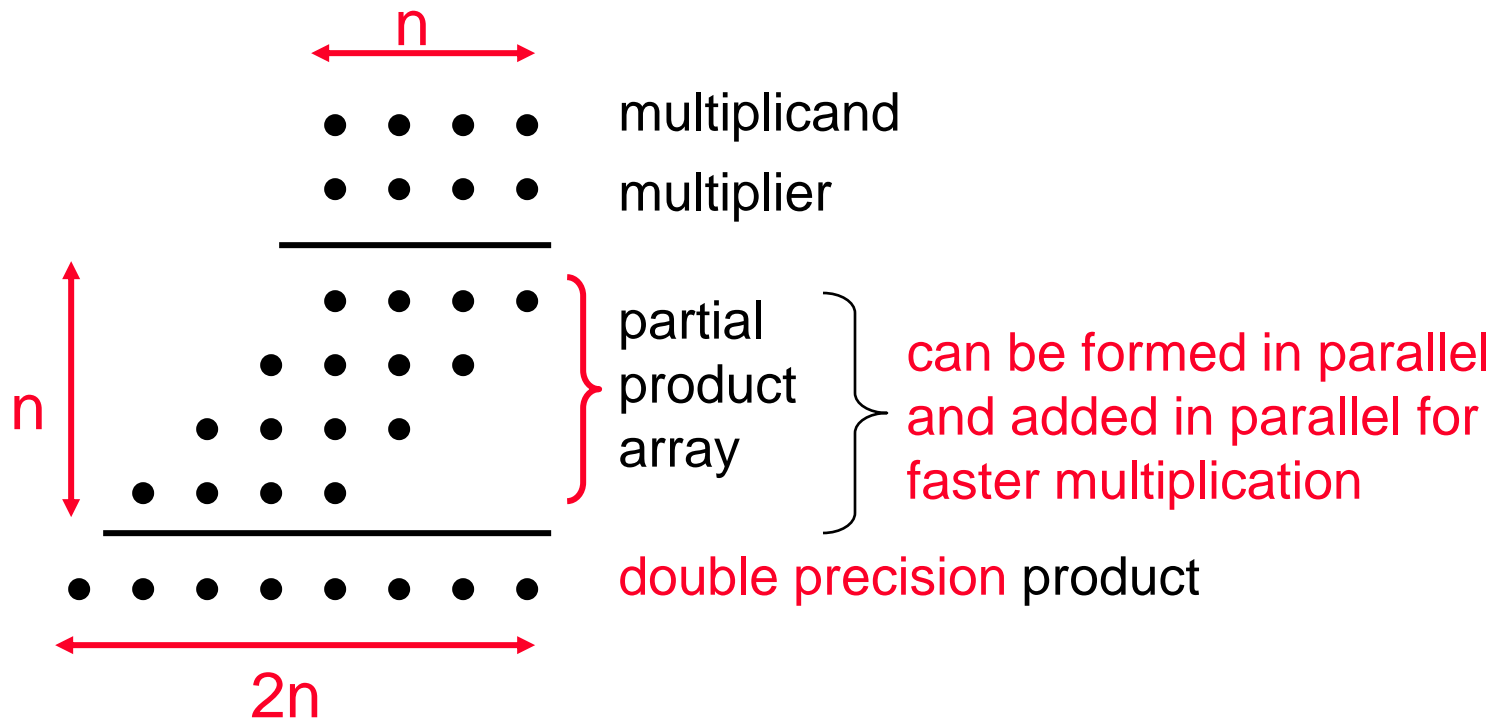
---

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



# Multiply

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



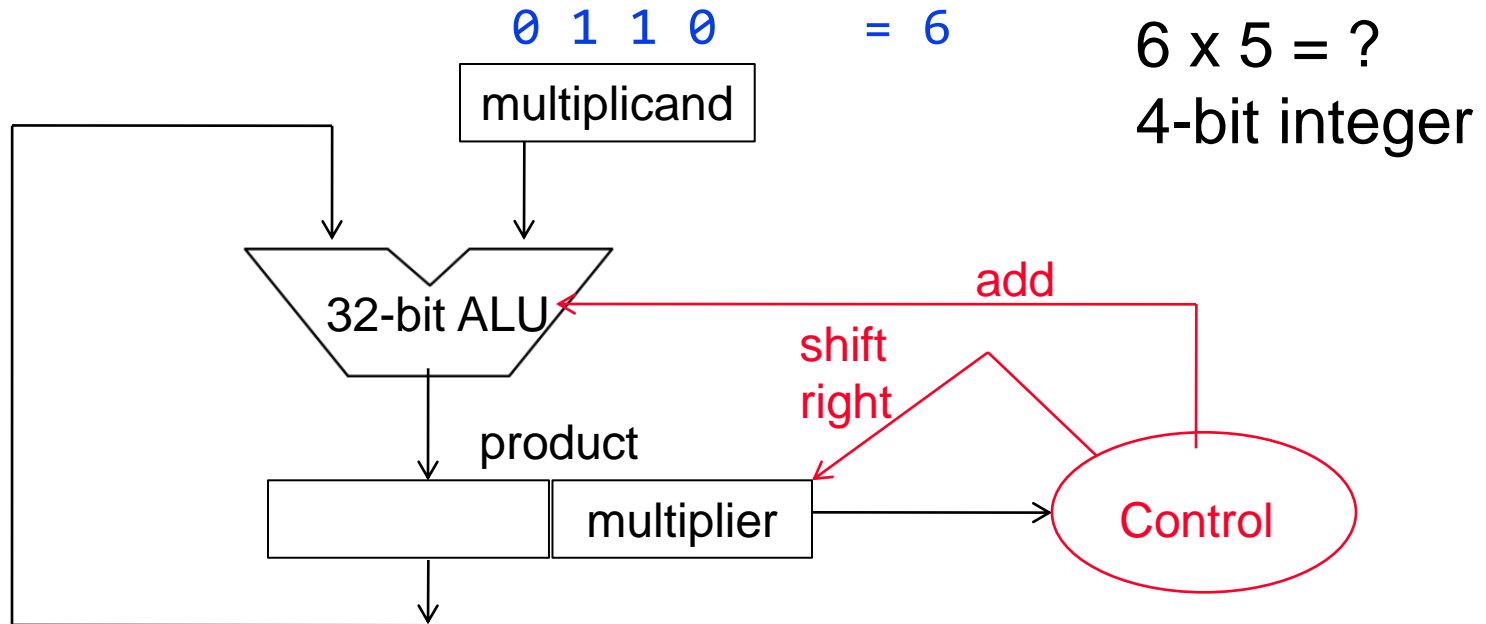
*$n$ -bit multiplicand and multiplier  $\rightarrow 2n$ -bit product*

# Example

---

Multiplicand		1000 <sub>ten</sub>
Multiplier	x	1001 <sub>ten</sub>
		<hr/>
		1000
		0000
		0000
		1000
		<hr/>
Product		1001000 <sub>ten</sub>

# 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

# MIPS Multiply Instruction

---

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

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

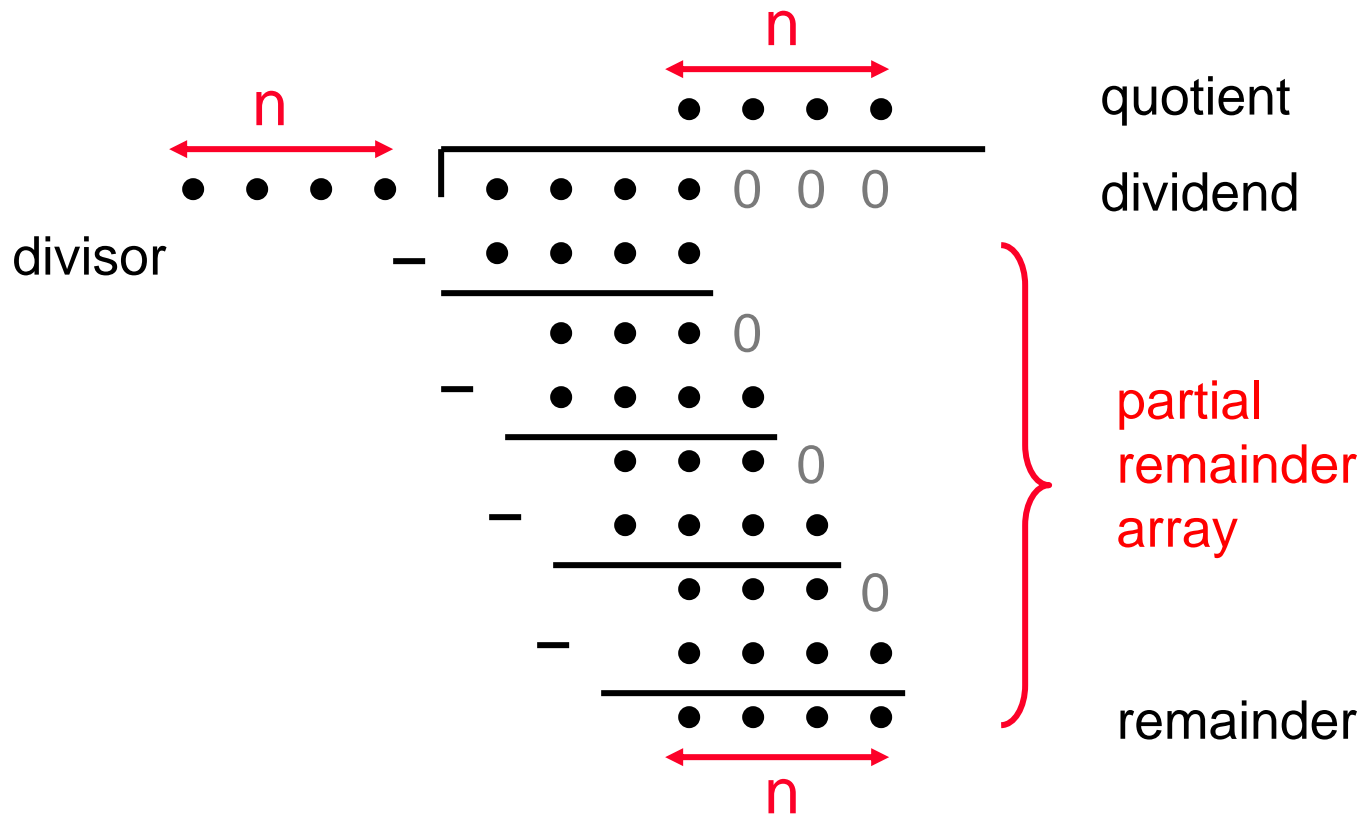
0	16	17	0	0	0x18
---	----	----	---	---	------

- ❑ 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

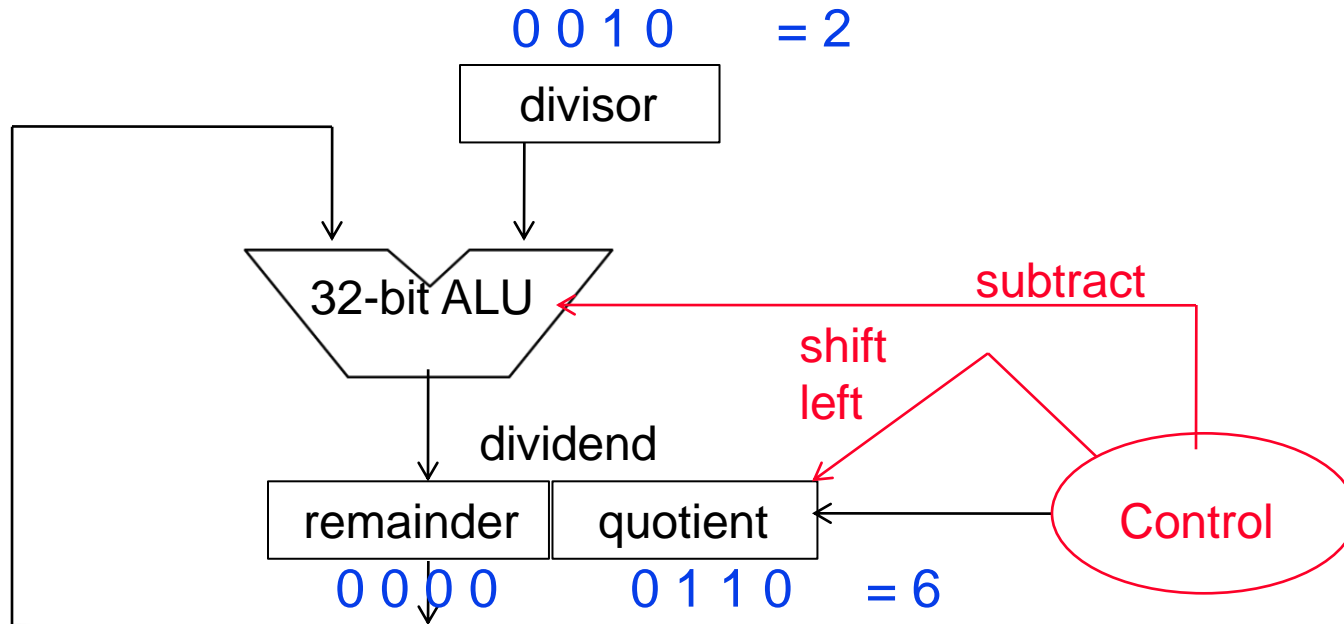
# Division

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

$$\text{dividend} = \text{quotient} \times \text{divisor} + \text{remainder}$$



# 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 remainder in `hi` and the quotient in `lo`

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

   # `hi` = `$s0` mod `$s1`

0	16	17	0	0	0x1A
---	----	----	---	---	------

- Instructions `mfhi rd` and `mflo rd` are provided to move the quotient and remainder 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.

# Representing Big (and Small) Numbers

## ❑ Encoding non-integer value?

- [illegible]

□ PI number

$$\text{PI} = 3.14159\dots$$

❑ Problem: how to represent the above numbers?

➔ We need reals or floating-point numbers!

➔ Floating point numbers in decimal:

→ 1000

→  $1 \times 10^3$

→  $0.1 \times 10^4$



# 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?

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

*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

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

- ❑ 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

$X = 1\textcolor{red}{100}\textcolor{red}{0001}\textcolor{red}{0}101\ 0110\ 0000\ 0000\ 0000\ 0000$

sign = 1  $\rightarrow$  X is negative

$E = 1000\ 0010 = 130$

$F = 10101100\dots00$

$$\begin{aligned}\rightarrow X &= (-1)^1 \times 1.101011\textcolor{red}{000}\textcolor{red}{.00} \times 2^{130-127} \\ &= -1.101011 \times 2^3 = -1101.011 \\ &= -13.375\end{aligned}$$

## 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)

$X = (-1)^0 \times 1.00...000 \times 2^{127-127} = 1.0$

## Example

---

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

Converting  $X$  to plain binary

$$9_{10} = 1001_2$$

$$0.6875 \times 2 = 1.375 \quad \rightarrow \text{get bit } 1$$

$$0.375 \times 2 = 0.75 \quad \rightarrow \text{get bit } 0$$

$$0.75 \times 2 = 1.5 \quad \rightarrow \text{get bit } 1$$

$$0.5 \times 2 = 1.0 \quad \rightarrow \text{get bit } 1$$



$$\rightarrow 9.6875_{10} = 1001.1011_2$$

## 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*

$$S = 0$$

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

$$m = 001101100\dots00 \text{ (23 bit)}$$

*Finally*

$$X = 0100\ 0001\ 0001\ 1011\ 0000\ 0000\ 0000\ 0000$$

# Examples

---

□  $1.0_2 \times 2^{-1} =$

□  $100.75_{10} =$

## Some special values

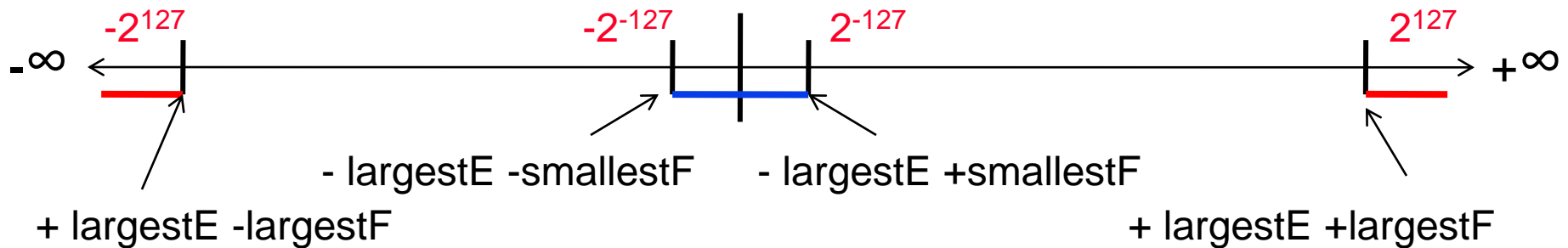
---

- ❑ Smallest+: 0 00000001 1.00000000000000000000000000000000  
=  $1 \times 2^{1-127}$
- ❑ Zero: 0 00000000 00000000000000000000000000000000  
= true 0
- ❑ Largest+: 0 11111110 1.11111111111111111111111111111111  
=  $(2-2^{-23}) \times 2^{254-127}$

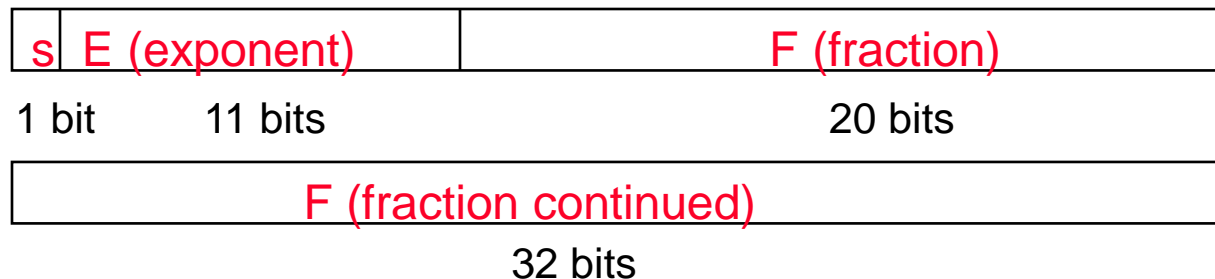


# Too large or too small values

- ❑ **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
  - ❑  $\pm$  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	$\pm$ denormalized number
0111 1111 to +127,-126	anything	0111 ...1111 to +1023,-1022	anything	$\pm$ floating point number
1111 1111	+ 0	1111 ... 1111	- 0	$\pm$ 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  $E1 \geq E2$ ) 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.XXXXXX ...)
  - If F1 and F2 have the same sign  $\rightarrow F3 \in [1,4) \rightarrow$  1 bit right shift F3 and increment  $E3$  (check for overflow)
  - If F1 and F2 have different signs  $\rightarrow$  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:  $0.5 + (-0.4375) = ?$

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

- 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  
 $0.001 \times 2^{-1} = 0.010 \times 2^{-2} = \dots = 1.000 \times 2^{-4}$
- Step 4: The sum is already rounded, so we're done
- Step 5: Rehide 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.XXXXXX ...)
  - Since F1 and F2 come in normalized  $\rightarrow F3 \in [1,4) \rightarrow$  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

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

- ❑ 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  
 $1.0000 \times 1.110 = 1.110000$
- ❑ Step 3: Normalized the product, checking for exp over/underflow  
 $1.110000 \times 2^{-3}$  is already normalized
- ❑ Step 4: The product is already rounded, so we're done
- ❑ Step 5: Rehide the hidden bit before storing

# MIPS Arithmetic Logic Unit (ALU)

- ❑ Must support the Arithmetic/Logic operations of the ISA

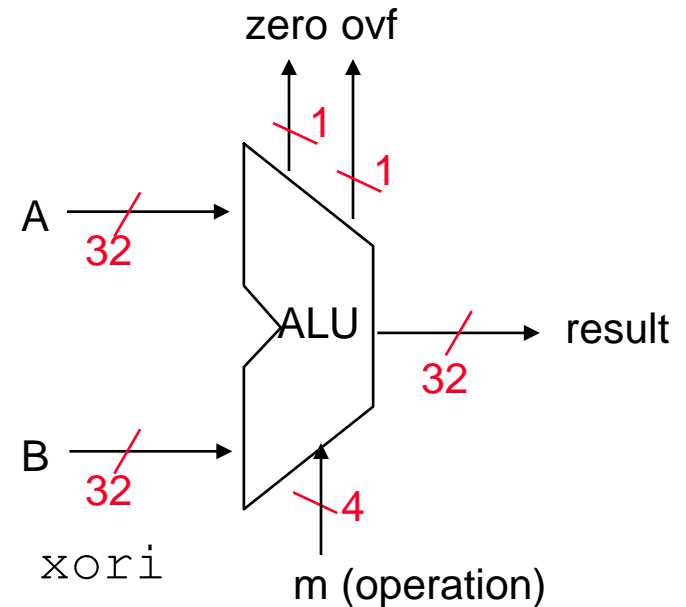
add, addi, addiu, addu

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