Chapter 2:
Arithmetic for Computers

Ngo Lam Trung

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

Content

- Integer representation and arithmetic
- Floating point number representation and arithmetic

What are stored inside computer?

- Data, of course!
- And data is represented as binary numbers
- Then how binary numbers are treated by MIPS?
  - Integers
    - Unsigned
    - Signed
  - Floating point numbers
    - Single precision
    - Double precision
    - Other formats

Unsigned Binary Integers

- Using n-bit binary number to represent non-negative integer

$$
\begin{aligned}
x &= x_{n-1}x_{n-2}\ldots x_1x_0 \\
&= x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0
\end{aligned}
$$

- **Range: 0 to $+2^n - 1$**
- **Example**

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
  $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
  $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

- **Data range using 32 bits**

  0 to $2^{32}-1 = 4{,}294{,}967{,}295$

Eg: 32 bit Unsigned Binary Integers

Hex	Binary	Decimal
0x00000000	0…0000	0
0x00000001	0…0001	1
0x00000002	0…0010	2
0x00000003	0…0011	3
0x00000004	0…0100	4
0x00000005	0…0101	5
0x00000006	0…0110	6
0x00000007	0…0111	7
0x00000008	0…1000	8
0x00000009	0…1001	9
	…	
0xFFFFFFFC	1…1100	2^{32}-4
0xFFFFFFFD	1…1101	2^{32}-3
0xFFFFFFFE	1…1110	2^{32}-2
0xFFFFFFFF	1…1111	2^{32}-1

Exercise

- Convert to 32-bit integers

  25 = 0000 0000 0000 0000 0000 0000 0001 1001

  125 = 0000 0000 0000 0000 0000 0000 0111 1101

  255 = 0000 0000 0000 0000 0000 0000 1111 1111

- Convert 32-bit integers to decimal

  0000 0000 0000 0000 0000 0000 1100 1111 = 207

  0000 0000 0000 0000 0000 0001 0011 0011 = 307

Signed binary integers

- Using n-bit binary number to represent integer, including negative values

$$x = x_{n-1}x_{n-2}\ldots x_1x_0$$
$$= -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- **Range: -2^{n-1} to $+2^{n-1} - 1$**

- **Example**

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
  $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
  $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

- **Using 32 bits**

  –2,147,483,648 to +2,147,483,647

Signed integer negation

- Given $x = x_{n-1}x_{n-2} \dots x_1x_0$, how to calculate $-x$?
- Let $\bar{x} = 1's\ complement\ of\ x$

$$\bar{x} = 1111 \dots 11_2 - x$$

(1 → 0, 0 → 1)

Then

$$\bar{x} + x = 1111 \dots 11_2 = -1$$

➔ $\bar{x} + 1 = -x$

- **Example: find binary representation of -2**

+2 = 0000 0000 … 0010_2

–2 = 1111 1111 … 1101_2 + 1
= 1111 1111 … 1110_2

Signed binary negation

$-2^3 =$

$-(2^3 - 1) =$

2’sc binary	decimal
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

$2^3 - 1 =$

complement all the bits

0101

and add a 1

0110

1011

and add a 1

1010

complement all the bits

Exercise

- Find 16 bit signed integer representation of

  16 = 0000 0000 0001 0000

  -16 = 1111 1111 1111 0000

  100 = 0000 0000 0110 0100

  -100 = 1111 1111 1001 1100

Sign extension

- Given n-bit integer $x = x_{n-1}x_{n-2}\ldots x_1x_0$
- Find corresponding m-bit representation (m > n) with the same numeric value

$$x = x_{m-1}x_{m-2}\ldots x_1x_0$$

- → Replicate the sign bit to the left

- Examples: 8-bit to 16-bit

  +2: 0000 0010 => 0000 0000 0000 0010

  –2: 1111 1110 => 1111 1111 1111 1110

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

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

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

Basics of logic design (Appendix B)

- Boolean logic: logic variable and operators
- Logic variable: values of 1 (TRUE) or 0 (FALSE)
- Basic operators: AND, OR, NOT
  - A AND B : $A \cdot B$ hay AB
  - A OR B : $A + B$
  - NOT A : $\overline{A}$
  - Order: NOT > AND > OR
- Additional operators: NAND, NOR, XOR
  - A NAND B: $\overline{A \cdot B}$
  - A NOR B : $\overline{A + B}$
  - A XOR B: $A \oplus B = A \bullet \overline{B} + \overline{A} \bullet B$

Truth tables

A	B	A AND B $A \bullet B$
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B $A + B$
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A $\overline{A}$
0	1
1	0

Unary operator NOT

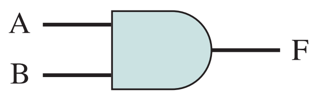
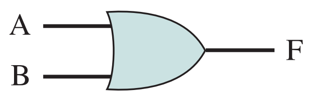
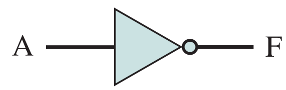
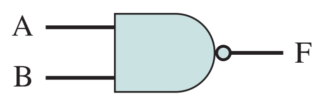
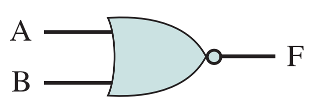
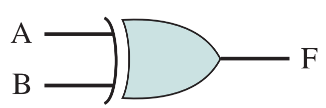
A	B	A NAND B $\overline{A \bullet B}$
0	0	1
0	1	1
1	0	1
1	1	0

A	B	A XOR B / A NOR B $A \oplus B$
0	0	0 / 1
0	1	0 / 1
1	0	0 / 1
1	1	0

Laws of Boolean algebra

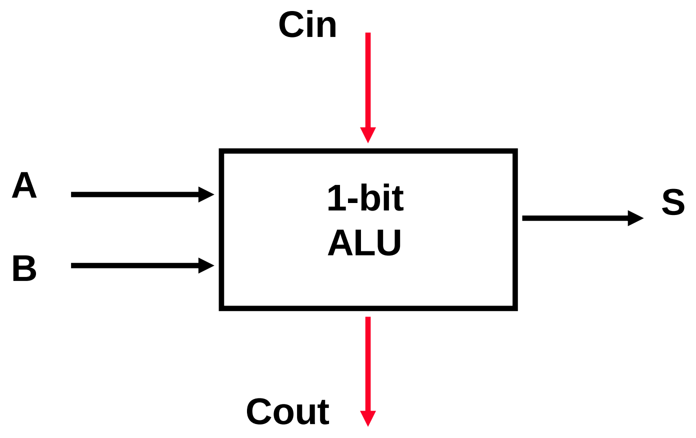
$A \bullet B = B \bullet A$	$A + B = B + A$
$A \bullet (B + C) = (A \bullet B) + (A \bullet C)$	$A + (B \bullet C) = (A + B) \bullet (A + C)$
$1 \bullet A = A$	$0 + A = A$
$A \bullet \overline{A} = 0$	$A + \overline{A} = 1$
$0 \bullet A = 0$	$1 + A = 1$
$A \bullet A = A$	$A + A = A$
$A \bullet (B \bullet C) = (A \bullet B) \bullet C$	$A + (B + C) = (A + B) + C$
$\overline{A \bullet B} = \overline{A} + \overline{B}$ (DeMorgan's law)	$\overline{A + B} = \overline{A} \bullet \overline{B}$ (DeMorgan's law)

Logic gates

<table>
<tr><th>Name</th><th>Graphical Symbol</th><th>Algebraic Function</th><th>Truth Table</th></tr>
<tr><td>AND</td><td></td><td>$F = A \bullet B$
or
$F = AB$</td><td>$\begin{array}{cc|c} A & B & F \\ \hline 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{array}$</td></tr>
<tr><td>OR</td><td></td><td>$F = A + B$</td><td>$\begin{array}{cc|c} A & B & F \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{array}$</td></tr>
<tr><td>NOT</td><td></td><td>$F = \overline{A}$
or
$F = A'$</td><td>$\begin{array}{c|c} A & F \\ \hline 0 & 1 \\ 1 & 0 \end{array}$</td></tr>
<tr><td>NAND</td><td></td><td>$F = \overline{AB}$</td><td>$\begin{array}{cc|c} A & B & F \\ \hline 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{array}$</td></tr>
<tr><td>NOR</td><td></td><td>$F = \overline{A + B}$</td><td>$\begin{array}{cc|c} A & B & F \\ \hline 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{array}$</td></tr>
<tr><td>XOR</td><td></td><td>$F = A \oplus B$</td><td>$\begin{array}{cc|c} A & B & F \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{array}$</td></tr>
</table>

Adder implementation

- 1-bit full adder

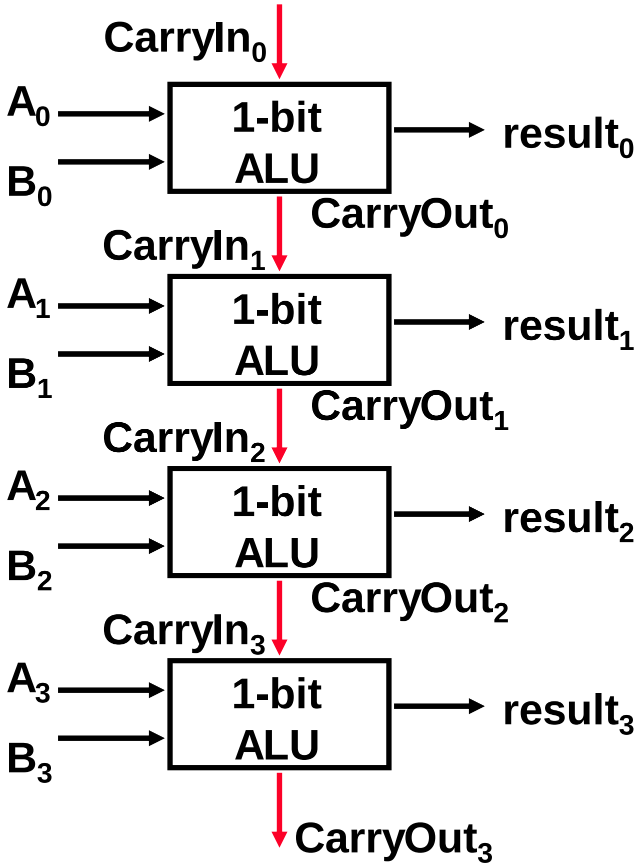


Inputs			Outputs	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- $S = Cin \oplus (A \oplus B)$
- $Cout = AB + BCin + ACin$

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:

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

- 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 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

$$
\begin{aligned}
ci+1 &= (bi \cdot ci) + (ai \cdot ci) + (ai \cdot bi) \\
&= (ai \cdot bi) + (ai + bi) \cdot ci
\end{aligned}
$$

- Denote

$$
\begin{aligned}
gi &= ai \cdot bi \\
pi &= ai + bi
\end{aligned}
$$

- Then

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

Carry lookahead

- With 4-bit adder

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

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

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

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

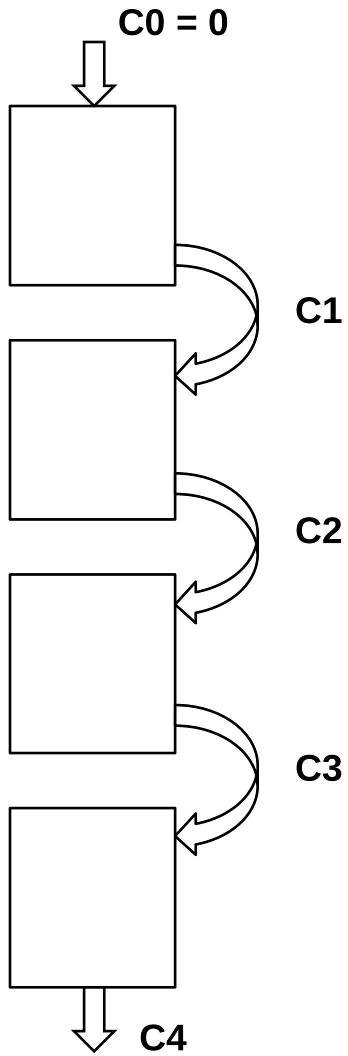
➔ All carry bits can be calculated after 3 gate delay

➔ 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

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

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

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

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

- Then big-carry bits can be calculated fast

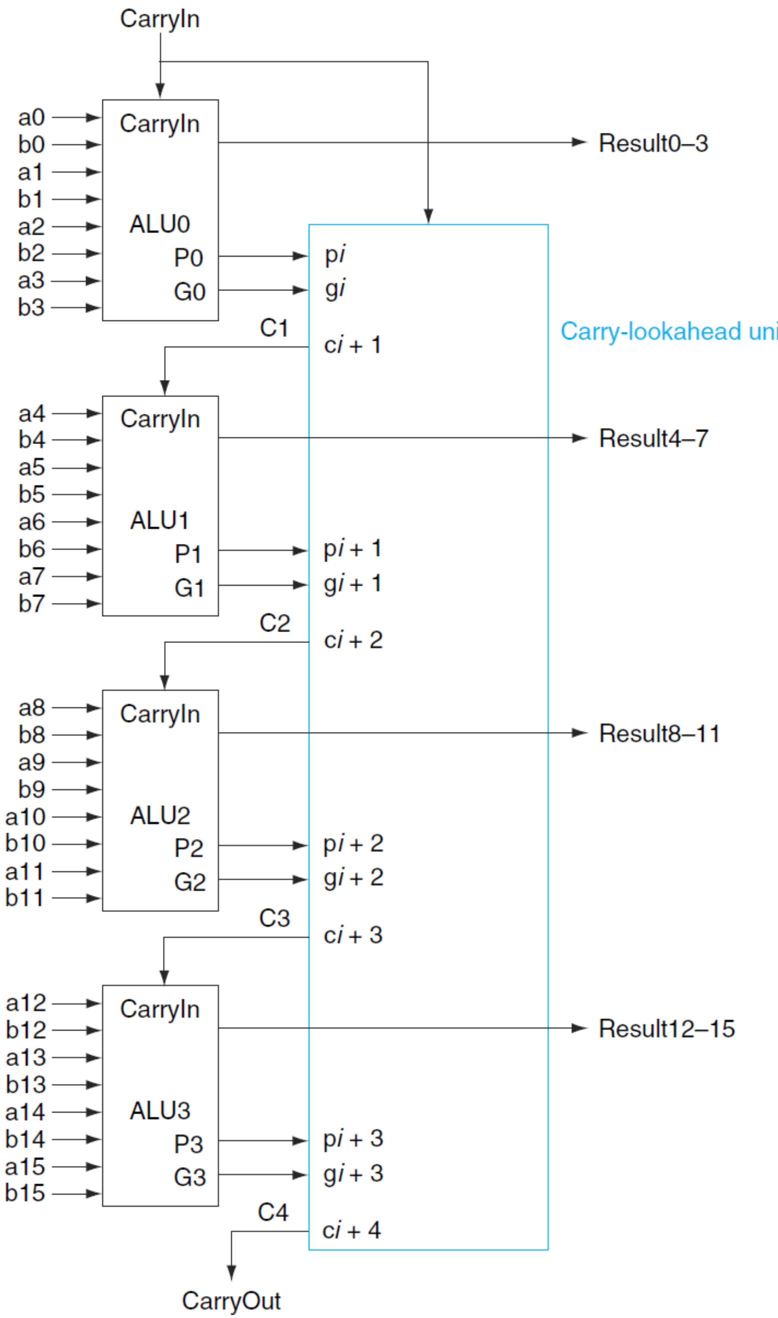
$$C1 = G0 + (P0 \cdot c0)$$

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

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

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

16-bit Adder



CarryIn
a0
b0
a1
b1
a2
b2
a3
b3
CarryIn
ALU0
P0
G0
Result0–3
pi
gi
C1
ci + 1
Carry-lookahead unit
a4
b4
a5
b5
a6
b6
a7
b7
CarryIn
ALU1
P1
G1
Result4–7
pi + 1
gi + 1
C2
ci + 2
a8
b8
a9
b9
a10
b10
a11
b11
CarryIn
ALU2
P2
G2
Result8–11
pi + 2
gi + 2
C3
ci + 3
a12
b12
a13
b13
a14
b14
a15
b15
CarryIn
ALU3
P3
G3
Result12–15
pi + 3
gi + 3
C4
ci + 4
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
gi:	0000	0000	0010	0011
pi:	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

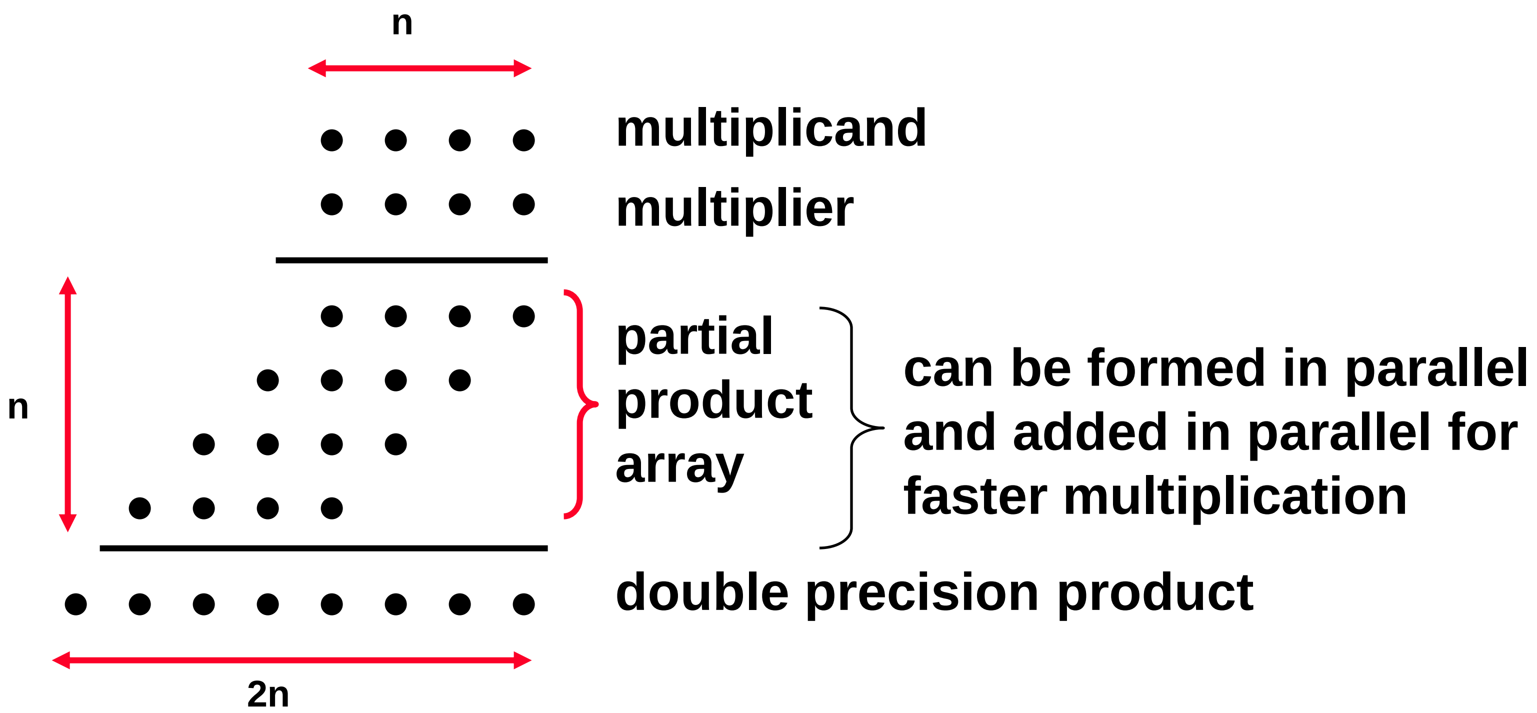
$$\begin{aligned} C4 &= G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) \\ &\quad + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0) \\ &= 0 + 0 + 1 + 0 + 0 = 1 \end{aligned}$$

Exercise

- Compare performance of 16-bit ripple carry and 16-bit carry 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 → 2n-bit product

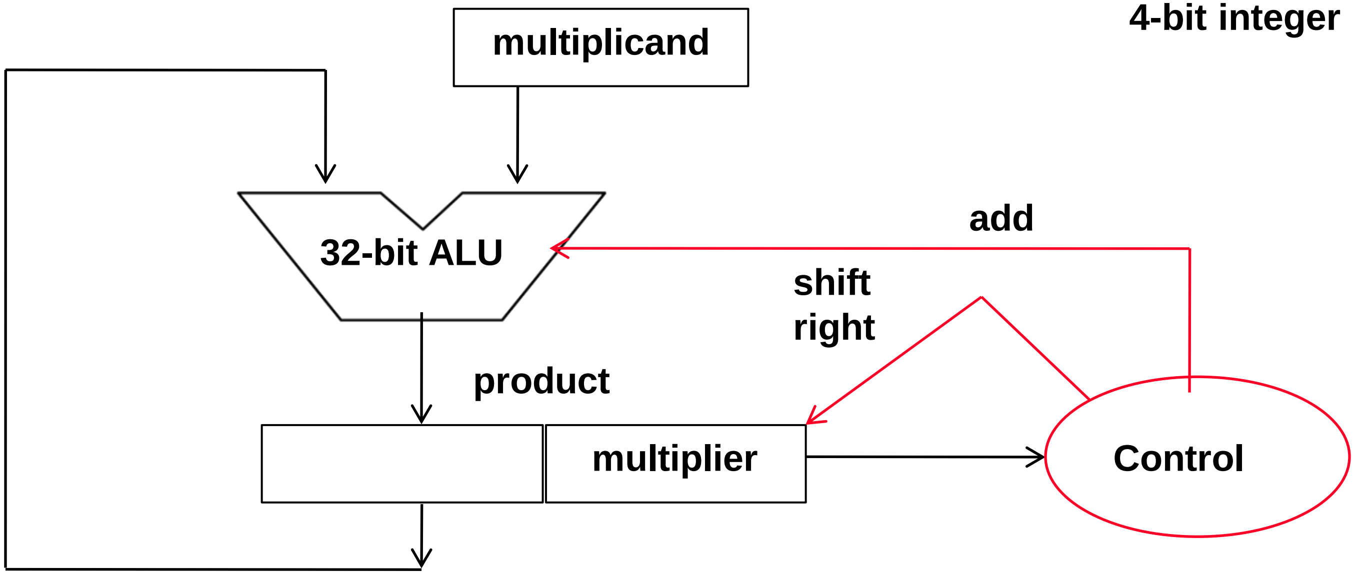
Example

$$
\begin{array}{llr}
\text{Multiplicand} & & 1000_{ten} \\
\text{Multiplier} & \times & 1001_{ten} \\
\hline
 & & 1000 \\
 & & 0000 \\
 & & 0000 \\
 & & 1000 \\
\hline
\text{Product} & & 1001000_{ten}
\end{array}
$$

Add and Right Shift Multiplier Hardware

0 1 1 0 = 6

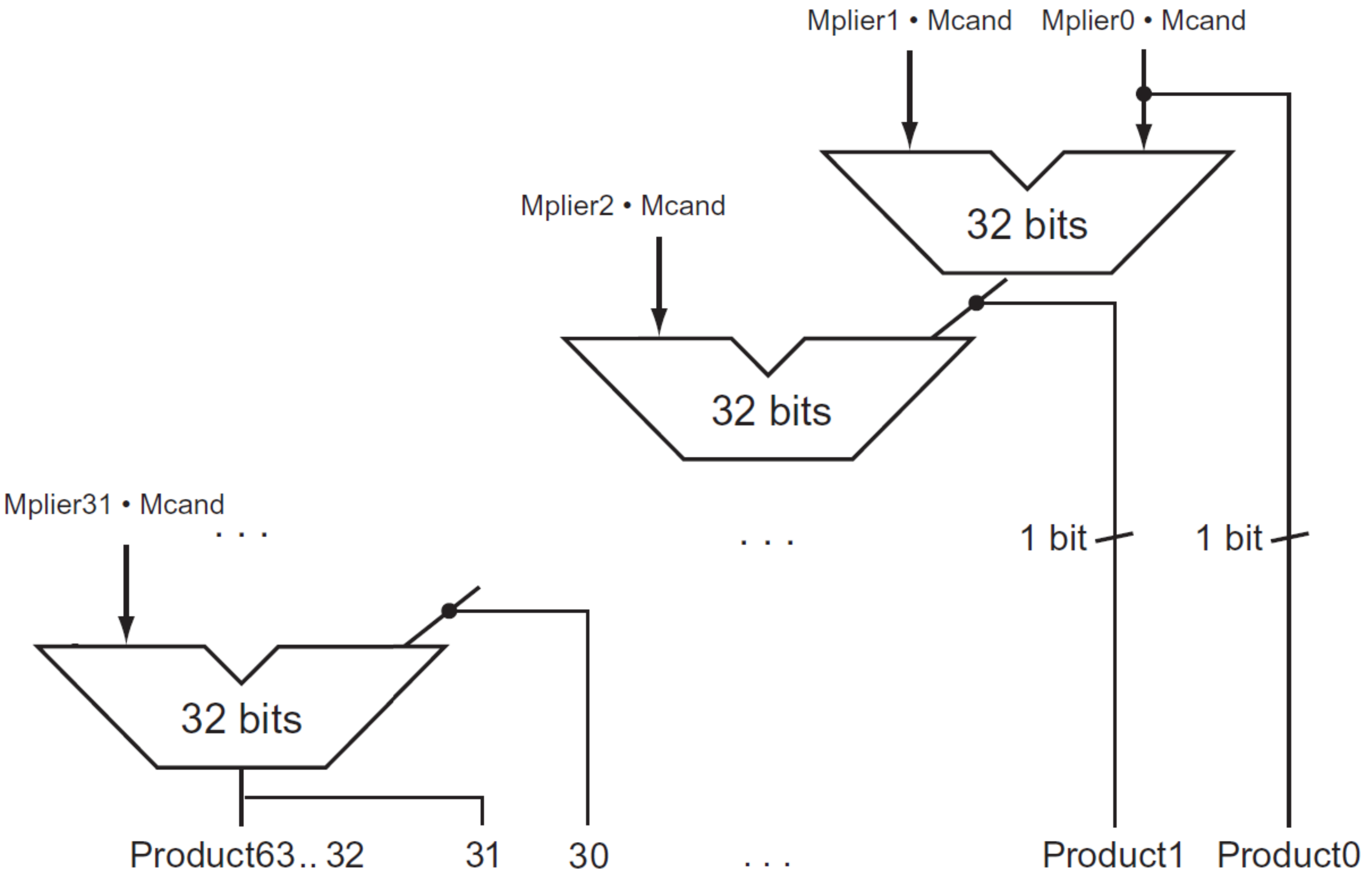
6 x 5 = ?
4-bit integer



	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

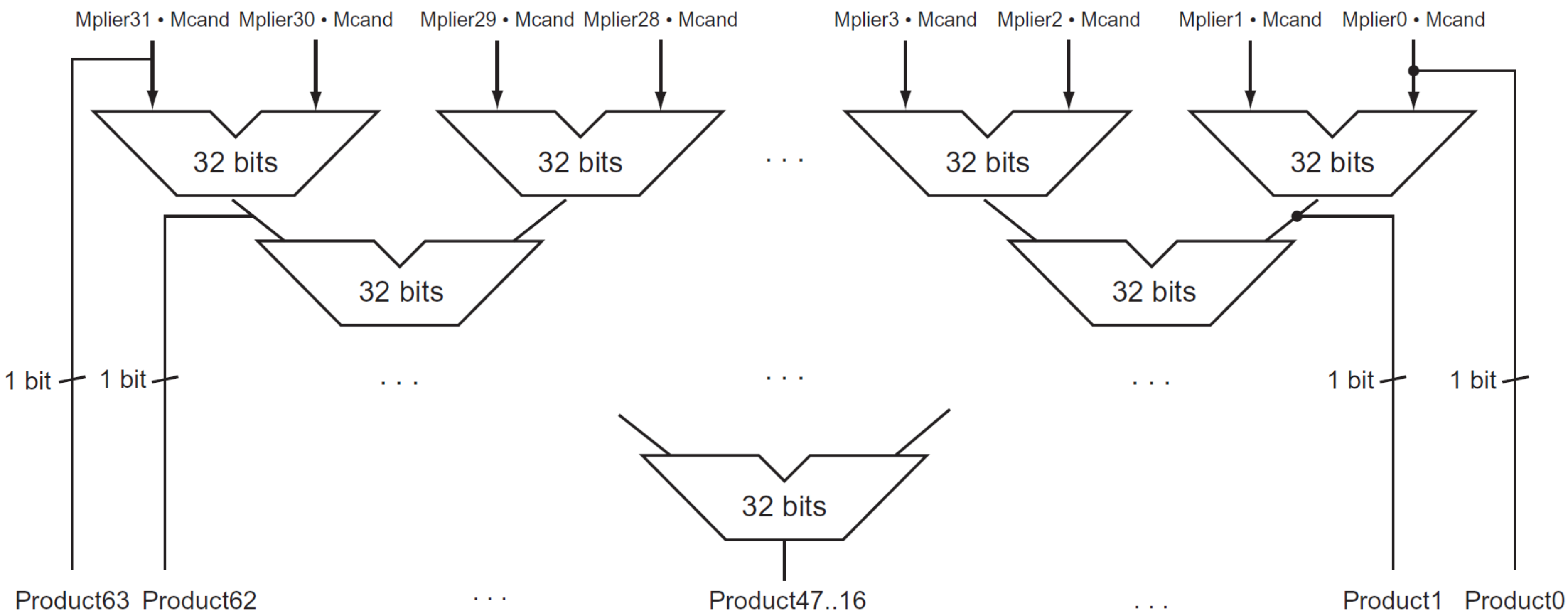
Fast multiplier – Design for Moore

- Why is this fast?



Fast multiplier – Design for Moore

- How fast is this?
- Anything wrong?



Multiply Instruction

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

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

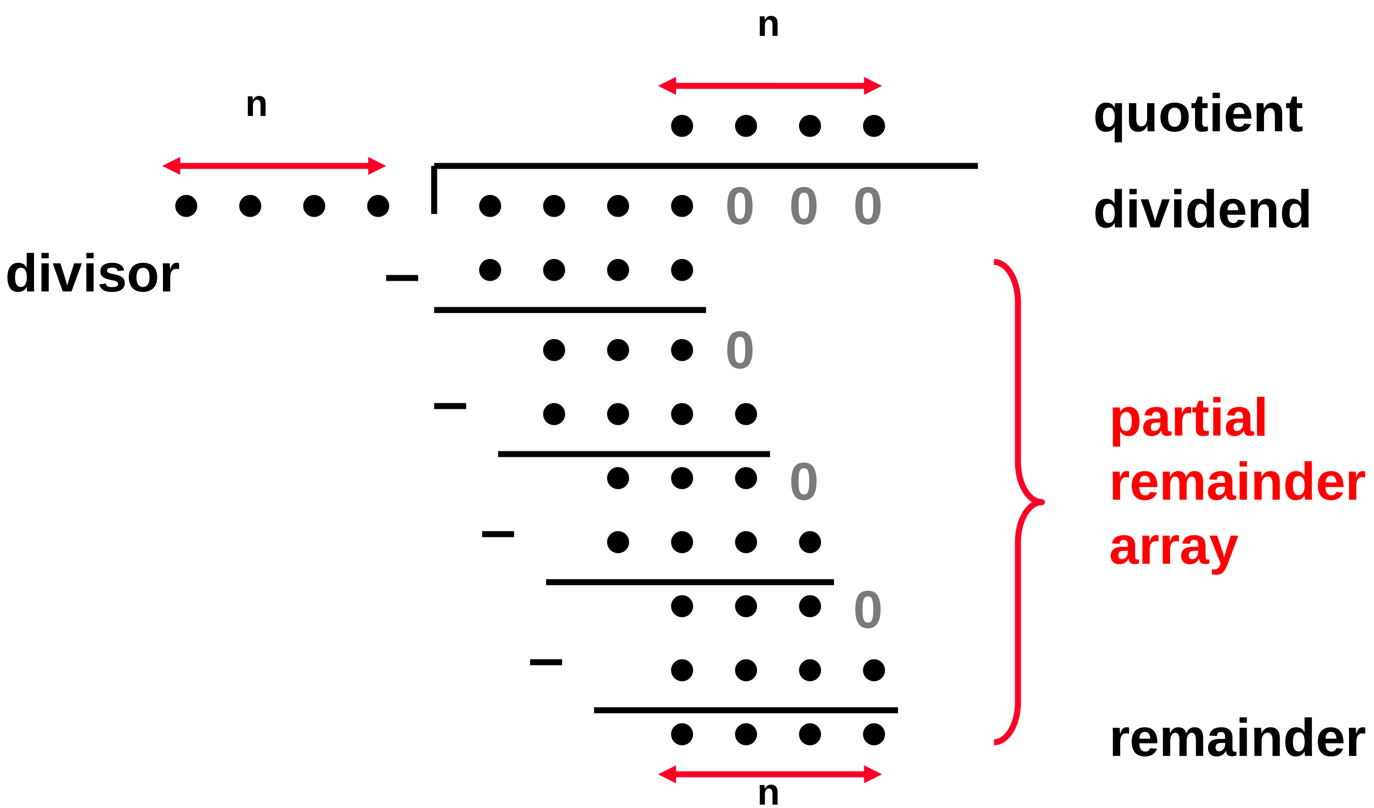
0	16	17	0	0	0x18

  - 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

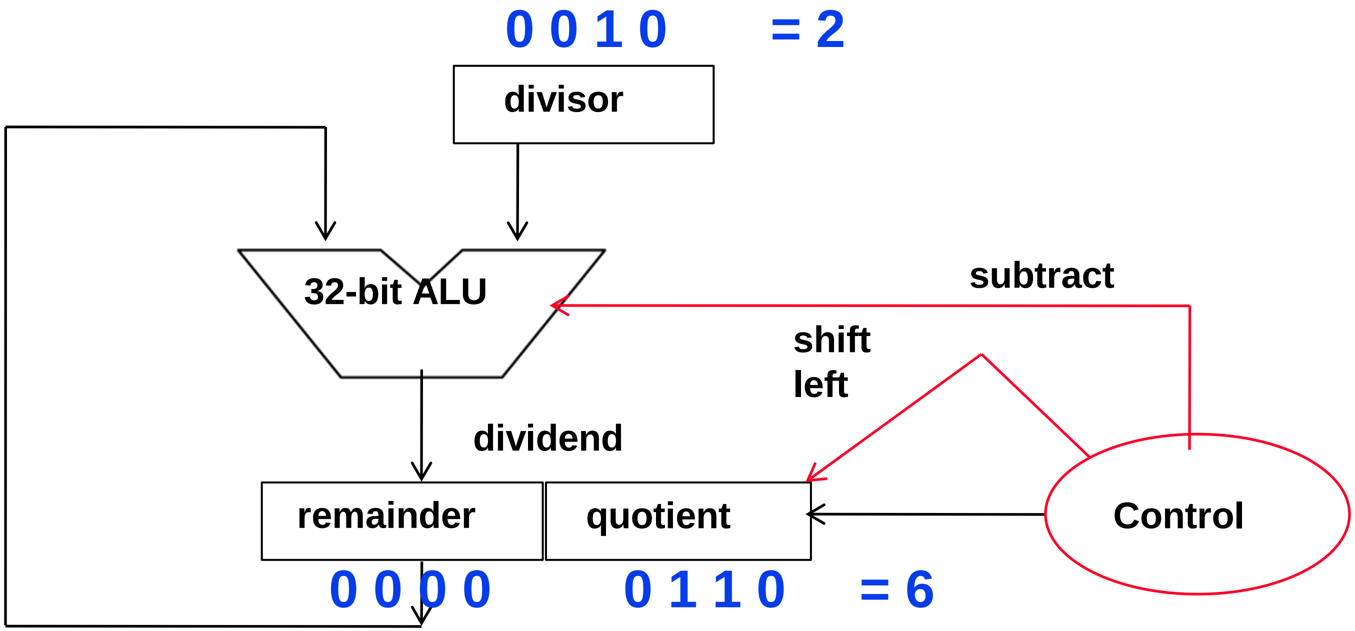
Division

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

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
```

0	16	17	0	0	0x1A	

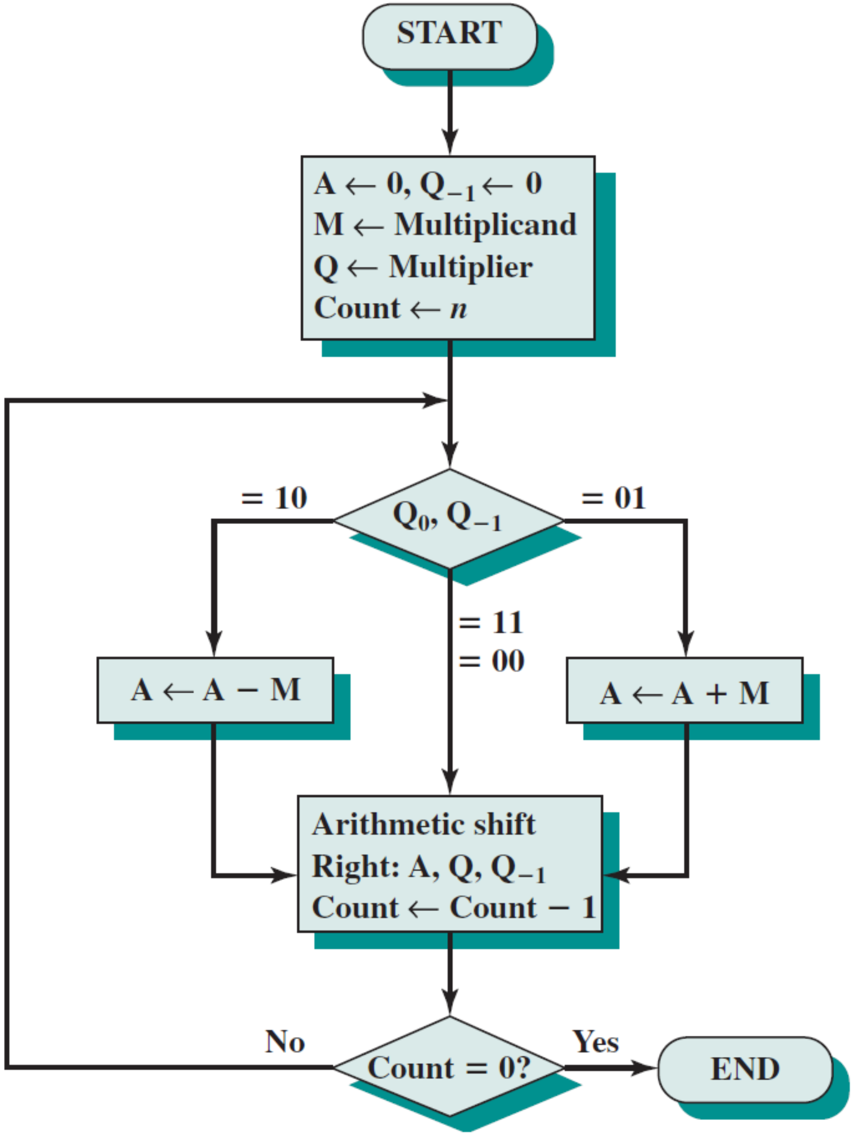
  - Instructions `mfhi rd` and `mflo rd` are provided to move the quotient and reminder to (user accessible) registers in the register file

- **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})

    0.00000000000000000000000000166 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?

$$2.0131228 * 10^3 = \underbrace{2.0131228}_{\textbf{mantissa}}\text{E+}\underbrace{03}_{\textbf{exponent}}$$

- 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**

s	E (exponent)	F (fraction)
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)

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

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 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

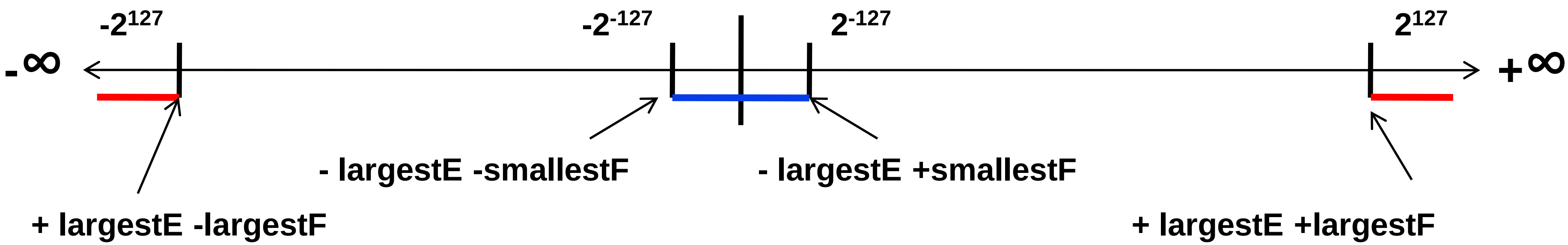
- Largest+: 0 11111110 1.11111111111111111111111

  $= (2-2^{-23}) \times 2^{254-127}$

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**

s	E (exponent)	F (fraction)
1 bit	11 bits	20 bits

F (fraction continued)
32 bits

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 $\geq$ 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.XXXXX …)**
    - **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

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

  - **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**

    $\mathbf{0.001 \times 2^{-1} = 0.010 \times 2^{-2} = .. = 1.000 \times 2^{-4}}$

  - **Step 4:** **The sum is already rounded, so we're done**
  - **Step 5:** **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.XXXXX …)**
    - **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

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

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

Floating Point Multiplication Example

- Multiply

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

- **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 x 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 and Round bit – 2 additional bits to increase 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 = ?