Integer Multiplication and Division

COE 301

Computer Organization

Prof. Muhamed Mudawar

College of Computer Sciences and Engineering

King Fahd University of Petroleum and Minerals

Presentation Outline

- **Unsigned Integer Multiplication**
- Signed Integer Multiplication
- Faster Integer Multiplication
- Integer Division
- Integer Multiplication and Division in MIPS

Unsigned Integer Multiplication

❖ Paper and Pencil Example:

Multiplicand		$1100_2 = 12$
Multiplier	×	$1101_2 = 13$
		1100
		0000
		1100
		1100
Product		$10011100_2 = 156$

Binary multiplication is easy

0 × multiplicand = 0

1 × multiplicand = multiplicand

❖ **n-bit multiplicand × n-bit multiplier = (2n)-bit product**

❖ Accomplished via **shifting** and **addition**

❖ Consumes more time and more chip area than addition

Unsigned Sequential Multiplication

- Initialize Product = 0
- Check each bit of the Multiplier
- If Multiplier bit = 1 then **Product = Product + Multiplicand**
- Rather than shifting the multiplicand to the left,

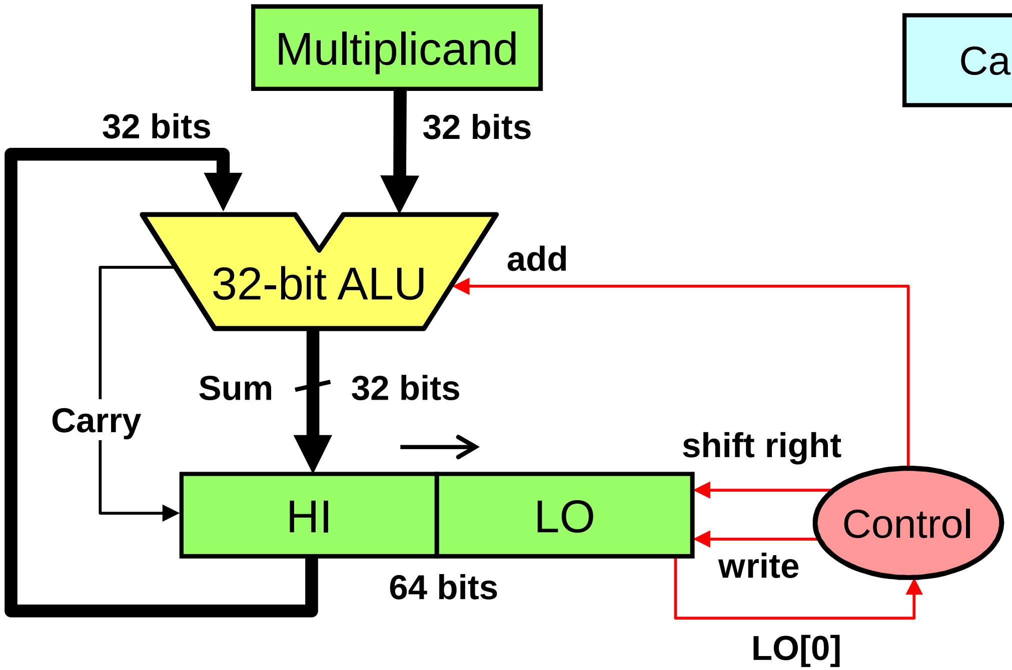
  Shift the Product to the Right

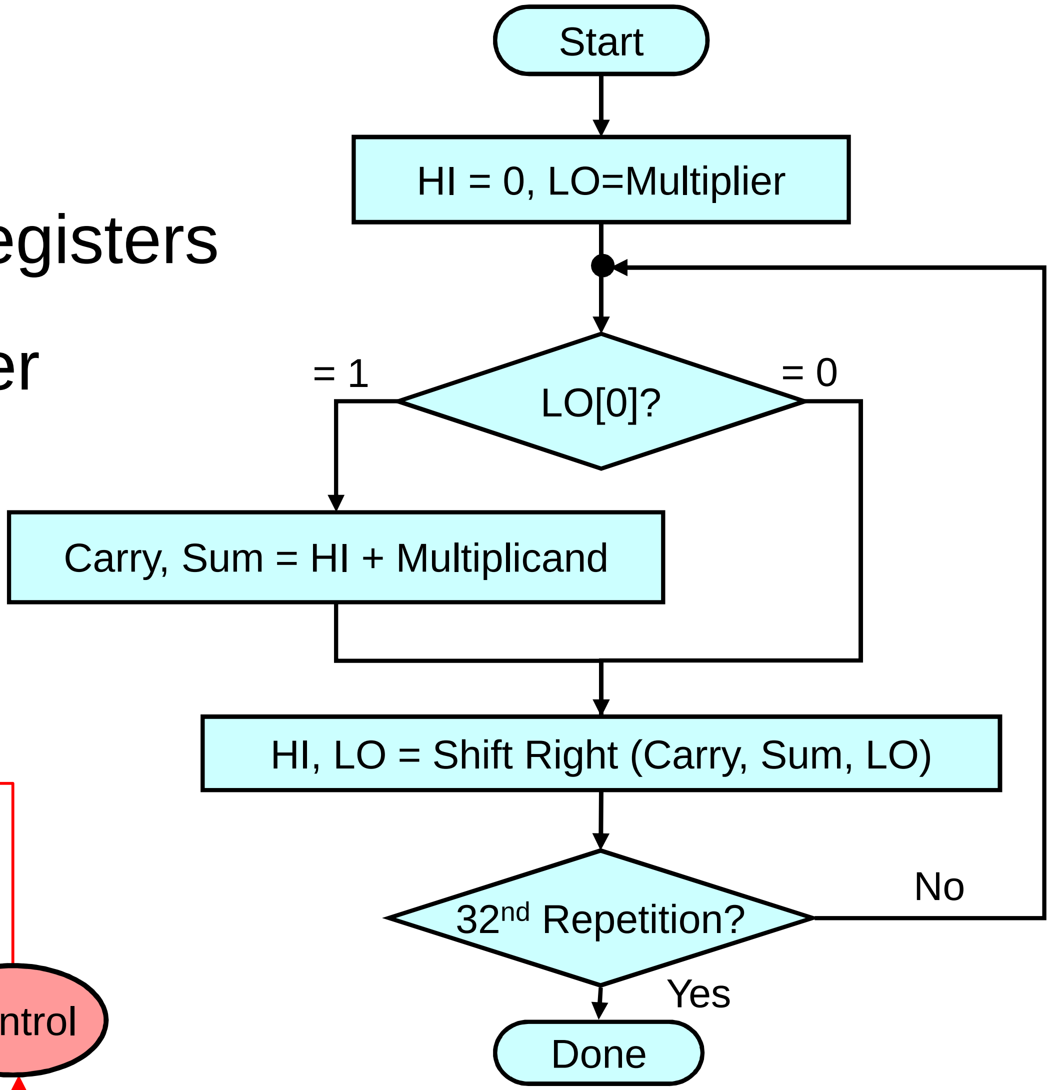
  Has the same net effect and produces the same result

  Minimizes the hardware resources
- One cycle per iteration (for each bit of the Multiplier)
  - Addition and shifting can be done simultaneously

Unsigned Sequential Multiplier

- Initialize: **HI = 0**
- Initialize: **LO = Multiplier**
- **Final Product** in **HI** and **LO** registers
- Repeat for each bit of Multiplier





Sequential Multiplier Example

- Consider: $1100_2 \times 1101_2$, Product = 10011100_2
- 4-bit multiplicand and multiplier are used in this example
- 4-bit adder produces a **4-bit Sum + Carry bit**

Iteration		Multiplicand	Carry	Product = HI, LO
0	Initialize (HI = 0, LO = Multiplier)	1 1 0 0		0 0 0 0 1 1 0 **1**
1	LO[0] = 1 => ADD	+	**0**	**1 1 0 0** 1 1 0 1
	Shift Right (Carry, Sum, LO) by 1 bit	1 1 0 0		**0 1 1 0 0** 1 1 **0**
2	LO[0] = 0 => NO addition			
	Shift Right (HI, LO) by 1 bit	1 1 0 0		**0 0 1 1 0 0** 1 **1**
3	LO[0] = 1 => ADD	+	**0**	**1 1 1 1 0 0** 1 1
	Shift Right (Carry, Sum, LO) by 1 bit	1 1 0 0		**0 1 1 1 1 0 0 1**
4	LO[0] = 1 => ADD	+	**1**	**0 0 1 1 1 0 0** 1
	Shift Right (Carry, Sum, LO) by 1 bit	1 1 0 0		**1 0 0 1 1 1 0 0**

Next . . .

- Unsigned Integer Multiplication
- **Signed Integer Multiplication**
- Faster Integer Multiplication
- Integer Division
- Integer Multiplication and Division in MIPS

Signed Integer Multiplication

- First attempt:
  - Convert multiplier and multiplicand into positive numbers
    - If negative then obtain the 2's complement and remember the sign
  - Perform unsigned multiplication
  - Compute the sign of the product
  - If product sign < 0 then obtain the 2's complement of the product
  - Drawback: additional steps to compute the 2's complement
- Better version:
  - Use the unsigned multiplication hardware
  - When shifting right, **extend the sign** of the product
  - If multiplier is negative, the **last step** should be a **subtract**

Signed Multiplication (Paper & Pencil)

- Case 1: Positive Multiplier

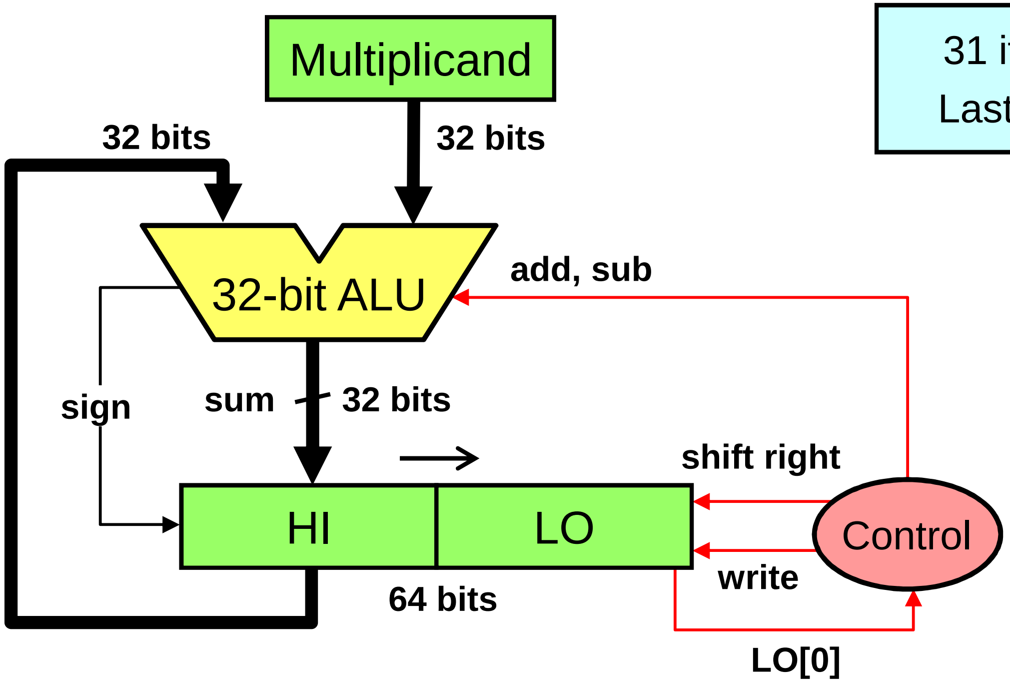
```
Multiplicand            1100₂ = -4
Multiplier         ×    0101₂ = +5
                   ----------------
Sign-extension    { 11111100
                  { 111100
                   ----------------
Product            11101100₂ = -20
```

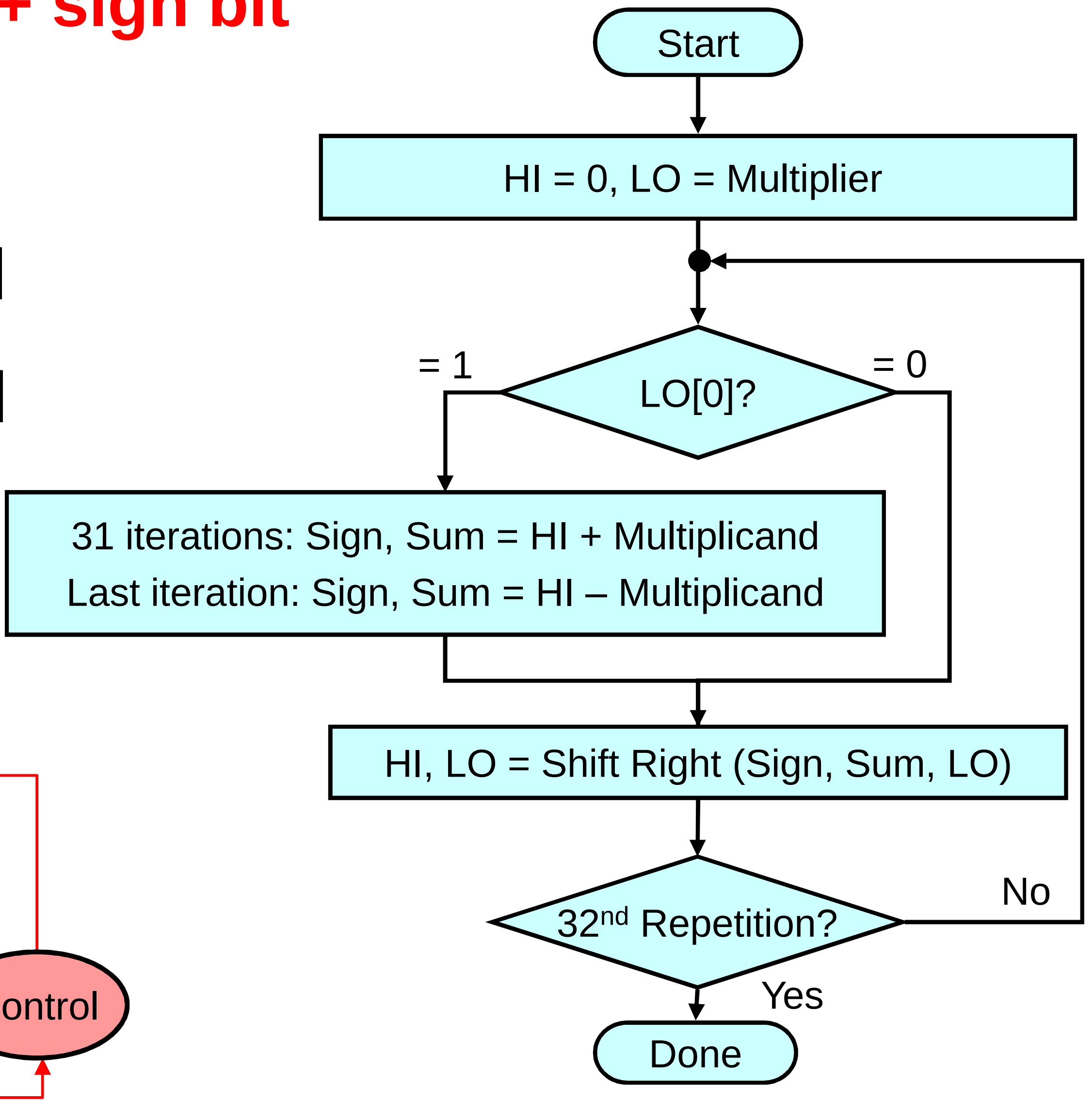
- Case 2: Negative Multiplier

```
Multiplicand            1100₂ = -4
Multiplier         ×    1101₂ = -3
                   ----------------
Sign-extension    { 11111100
                  { 111100
                    00100     (2's complement of 1100)
                   ----------------
Product            00001100₂ = +12
```

Signed Sequential Multiplier

- ALU produces: **32-bit sum + sign bit**
- Sign bit can be computed:
  - **No overflow: sign = sum[31]**
  - **If Overflow: sign = ~sum[31]**





Signed Multiplication Example

- Consider: 1100_2 (-4) × 1101_2 (-3), Product = 00001100_2
- Check for overflow: No overflow ➔ Extend sign bit
- Last iteration: add 2's complement of Multiplicand

Iteration		Multiplicand	Sign	Product = HI, LO
0	Initialize (HI = 0, LO = Multiplier)	1 1 0 0		0 0 0 0 1 1 0 1
1	LO[0] = 1 => ADD	+	1	1 1 0 0 1 1 0 1
	Shift Right (Sign, Sum, LO) by 1 bit	1 1 0 0		1 1 1 0 0 1 1 0
2	LO[0] = 0 => NO addition			
	Shift Right (Sign, HI, LO) by 1 bit	1 1 0 0		1 1 1 1 0 0 1 1
3	LO[0] = 1 => ADD	+	1	1 0 1 1 0 0 1 1
	Shift Right (Sign, Sum, LO) by 1 bit	1 1 0 0		1 1 0 1 1 0 0 1
4	LO[0] = 1 => SUB (ADD 2's compl)	0 1 0 0 +	0	0 0 0 1 1 0 0 1
	Shift Right (Sign, Sum, LO) by 1 bit			0 0 0 0 1 1 0 0

Next . . .

- Unsigned Integer Multiplication
- Signed Integer Multiplication
- **Faster Integer Multiplication**
- Integer Division
- Integer Multiplication and Division in MIPS

Faster Multiplier

- Suppose we want to multiply two numbers A and B
  - Example on 4-bit numbers: $A = a_3\ a_2\ a_1\ a_0$ and $B = b_3\ b_2\ b_1\ b_0$
- Step 1: AND (multiply) each bit of A with each bit of B
  - Requires n^2 AND gates and produces n^2 product bits
  - Position of $a_ib_j = (i+j)$. For example, Position of $a_2b_3 = 2+3 = 5$

A × B

			a_3b_0	a_2b_0	a_1b_0	a_0b_0
		a_3b_1	a_2b_1	a_1b_1	a_0b_1	
	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
a_3b_3	a_2b_3	a_1b_3	a_0b_3			

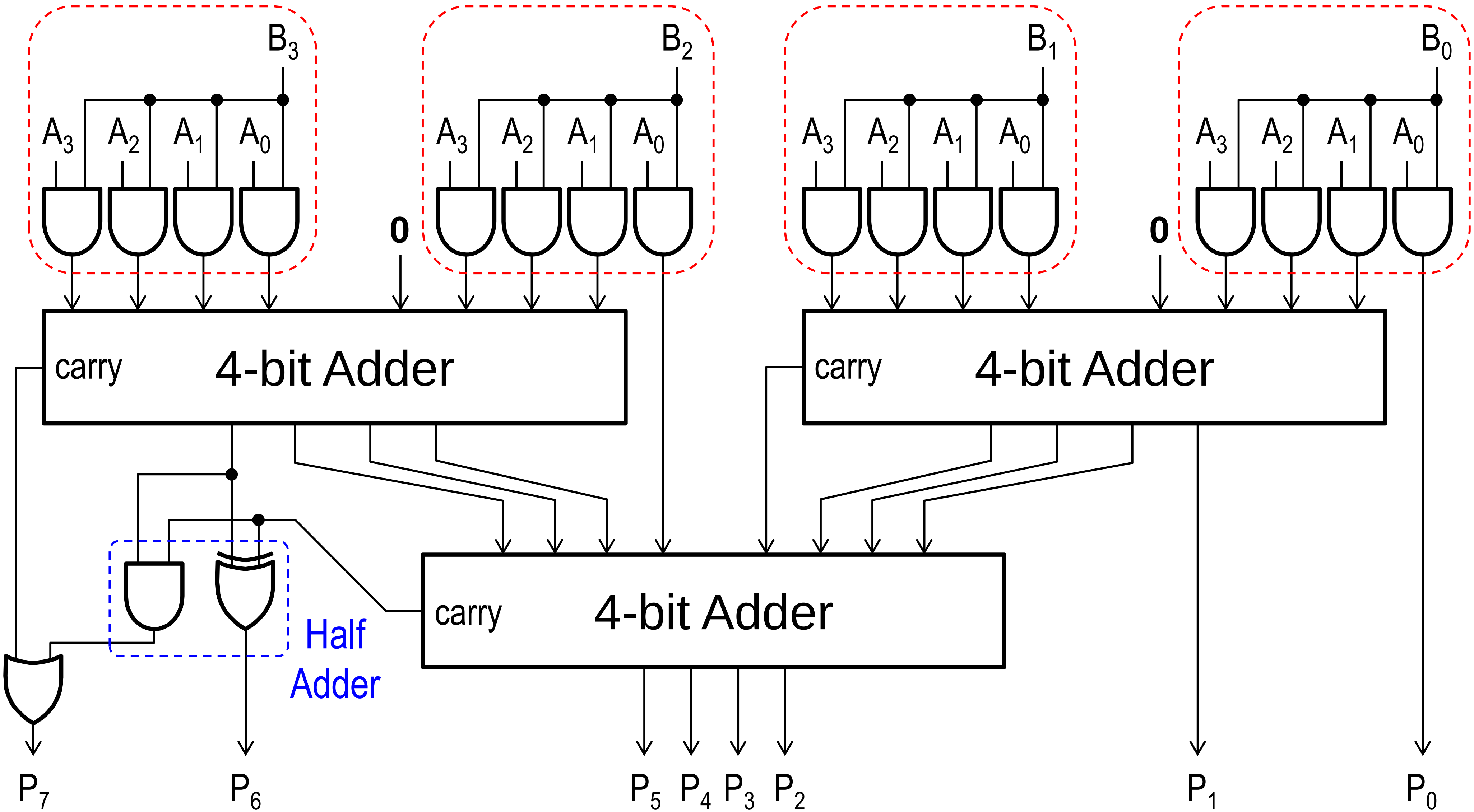
Adding the Partial Products

- Step 2: Add the partial products
  - The partial products are shifted and added to compute the product **P**
  - The partial products can be added in parallel
  - Different implementations are possible

4-bit Multiplicand					A_3	A_2	A_1	A_0
4-bit Multiplier				×	B_3	B_2	B_1	B_0
Partial Products are shifted and added					A_3B_0	A_2B_0	A_1B_0	A_0B_0
Can be added in parallel				A_3B_1	A_2B_1	A_1B_1	A_0B_1	
			A_3B_2	A_2B_2	A_1B_2	A_0B_2		
		A_3B_3	A_2B_3	A_1B_3	A_0B_3			
8-bit Product	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

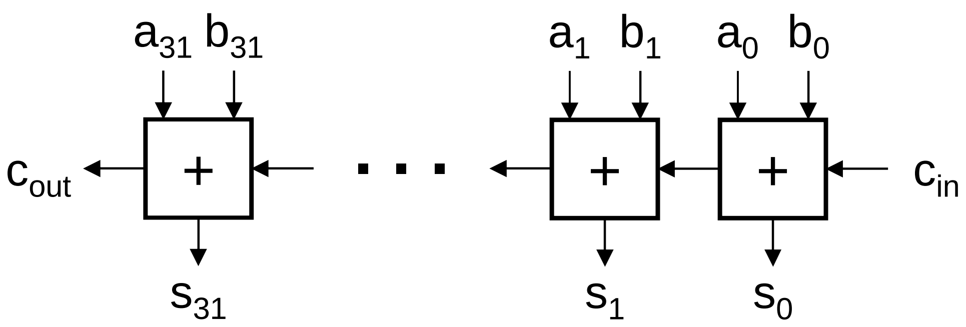
4-bit × 4-bit Binary Multiplier

16 AND gates, **Three** 4-bit adders, a half-adder, and an OR gate

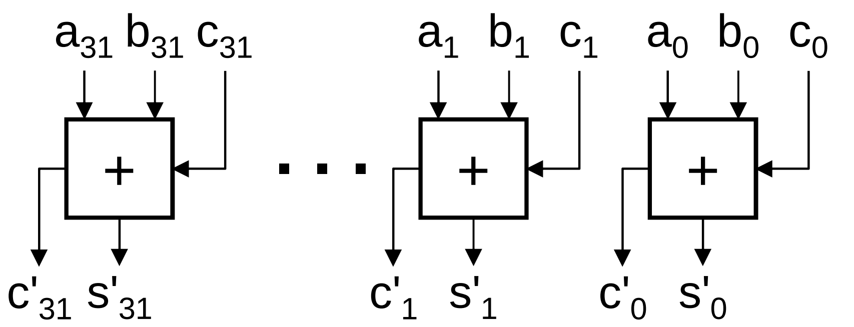


Carry Save Adders

- A n-bit carry-save adder produces two n-bit outputs
  - n-bit partial sum bits and n-bit carry bits
- All the n bits of a carry-save adder work in parallel
  - The carry does not propagate as in a carry-propagate adder
  - This is why a carry-save is faster than a carry-propagate adder
- Useful when adding multiple numbers (as in multipliers)



Carry-Propagate Adder

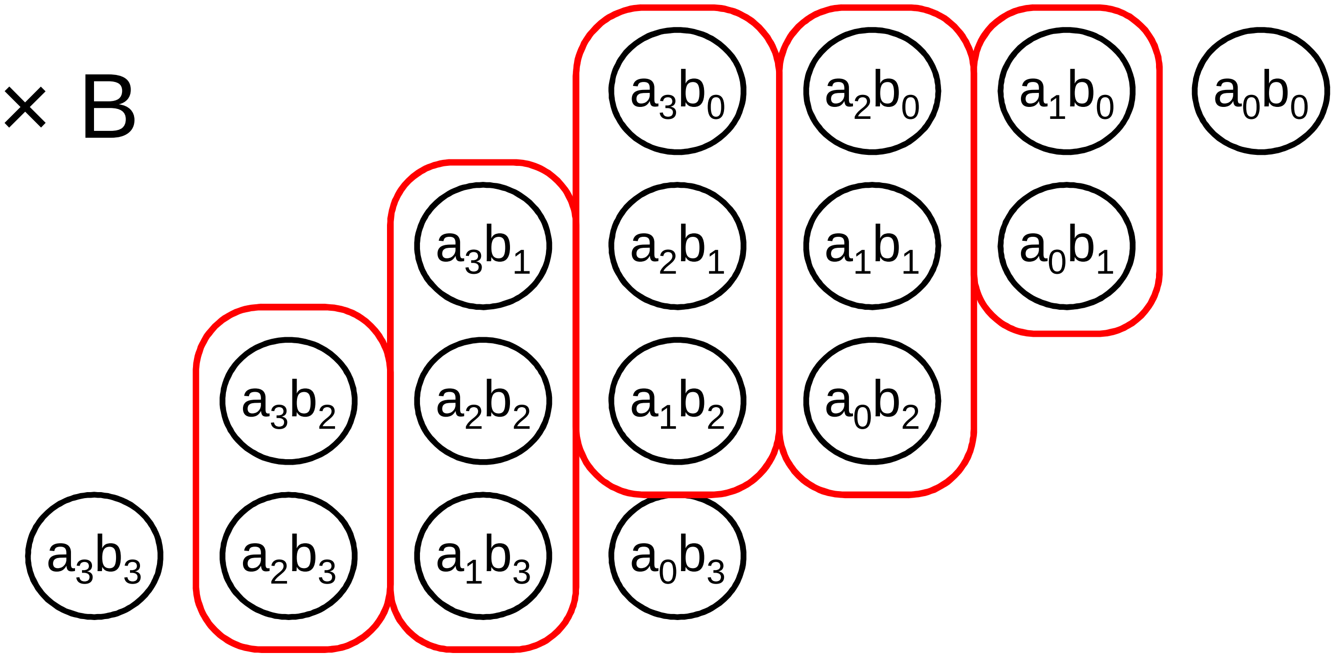


Carry-Save Adder

Carry-Save Adders in a Multiplier

- ADD the product bits vertically using **Carry-Save adders**
  - Full Adder adds three vertical bits
  - Half Adder adds two vertical bits
  - Each adder produces a partial sum and a carry
- Use **Carry-propagate adder** for final addition

$A \times B$

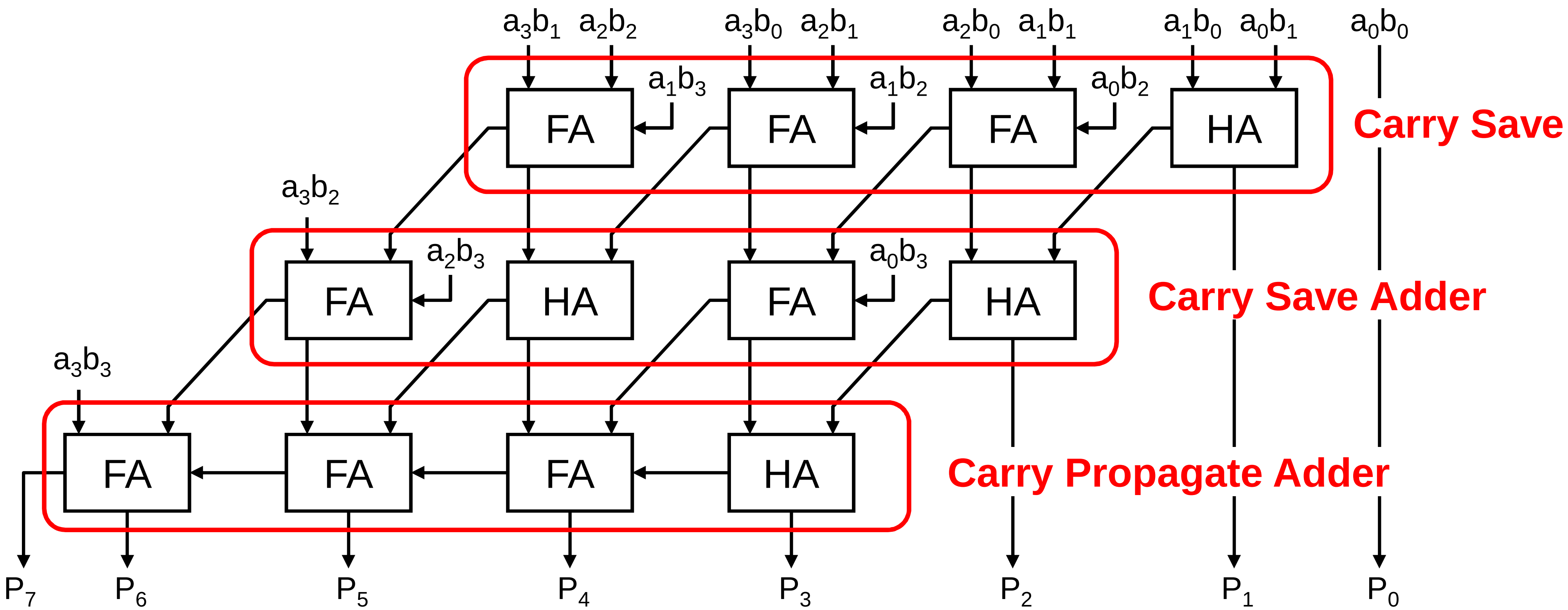


Carry-Save Adders in a Multiplier

Step 1: Use **carry save adders** to add the partial products

- ✧ Reduce the partial products to just two numbers

Step 2: Use **carry-propagate adder** to add last two numbers



Summary of a Fast Multiplier

- A fast n-bit × n-bit multiplier requires:
  - n^2 AND gates to produce n^2 product bits in parallel
  - Many adders to perform additions in parallel
- Uses carry-save adders to reduce delays
- Higher cost (more chip area) than sequential multiplier
- Higher performance (faster) than sequential multiplier

Next . . .

- Unsigned Integer Multiplication
- Signed Integer Multiplication
- Faster Integer Multiplication
- **Integer Division**
- Integer Multiplication and Division in MIPS

Unsigned Division (Paper & Pencil)

10011_2 = 19 **Quotient**

Divisor 1011_2) 11011001_2 = 217 **Dividend**

```
 -1011
 -----
    10
    101
    1010
    10100
    -1011
    -----
     1001
     10011
     -1011
     -----
      1000
```

1000_2 = 8 **Remainder**

Check how big a number can be subtracted, creating a bit of the quotient on each attempt

Binary division is done via shifting and subtraction

Dividend = Quotient × Divisor + Remainder

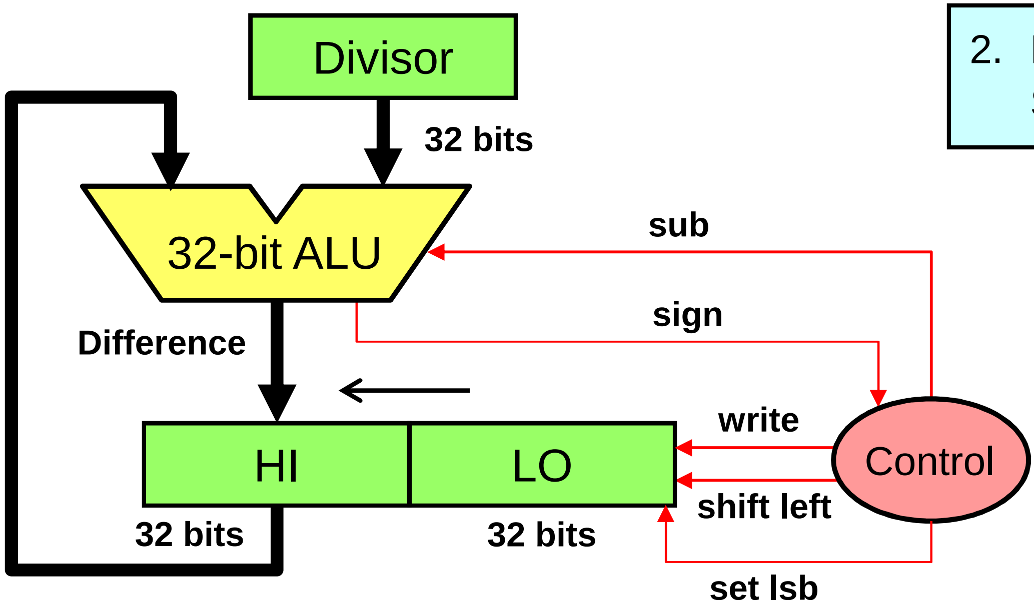
217 = 19 × 11 + 8

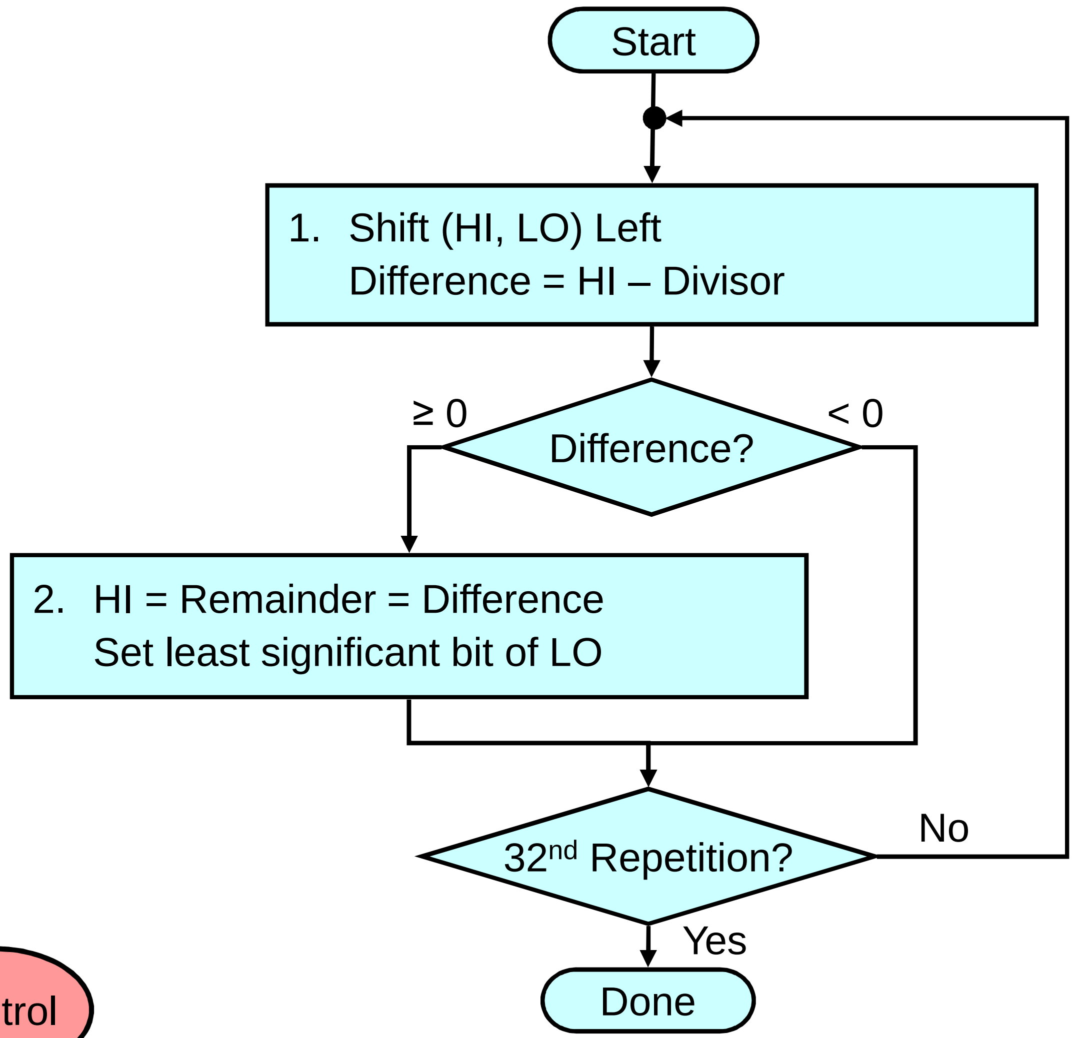
Sequential Division

- Uses two registers: HI and LO
- Initialize: HI = Remainder = 0 and LO = Dividend
- Shift (HI, LO) LEFT by 1 bit (also Shift Quotient LEFT)
  - Shift the remainder and dividend registers together LEFT
  - Has the same net effect of shifting the divisor RIGHT
- Compute: Difference = Remainder – Divisor
- If (Difference ≥ 0) then
  - Remainder = Difference
  - Set Least significant Bit of Quotient
- Observation to Reduce Hardware:
  - LO register can be also used to store the computed Quotient

Sequential Division Hardware

- Initialize:
  - HI = 0, LO = Dividend
- Results:
  - HI = Remainder
  - LO = Quotient





Unsigned Integer Division Example

- Example: **1110**$_2$ / **0100**$_2$ (4-bit dividend & divisor)
- Result Quotient = **0011**$_2$ and Remainder = **0010**$_2$
- 4-bit registers for Remainder and Divisor (4-bit ALU)

Iteration		HI	LO	Divisor	Difference
0	Initialize	0 0 0 0	**1 1 1 0**	**0 1 0 0**	
1	Shift Left, Diff = HI - Divisor	0 0 0 **1** ←	**1 1 0 0**	**0 1 0 0**	**< 0**
	Diff < 0 => Do Nothing				
2	Shift Left, Diff = HI - Divisor	0 0 **1 1** ←	**1 0 0 0**	**0 1 0 0**	**< 0**
	Diff < 0 => Do Nothing				
3	Shift Left, Diff = HI - Divisor	0 **1 1 1** ←	**0 0 0 0**	**0 1 0 0**	**0 0 1 1**
	HI = Diff, set **lsb** of LO	**0 0 1 1**	**0 0 0 1**		
4	Shift Left, Diff = HI - Divisor	**0 1 1 0** ←	**0 0 1 0**	**0 1 0 0**	**0 0 1 0**
	HI = Diff, set **lsb** of LO	**0 0 1 0**	**0 0 1 1**		

Signed Integer Division

- Simplest way is to remember the signs
- Convert the dividend and divisor to positive
  - Obtain the 2's complement if they are negative
- Do the unsigned division
- Compute the signs of the quotient and remainder
  - Quotient sign = Dividend sign XOR Divisor sign
  - Remainder sign = Dividend sign
- Negate the quotient and remainder if their sign is negative
  - Obtain the 2's complement to convert them to negative

Signed Integer Division Examples

1. **Positive** Dividend and **Positive** Divisor
   - ✧ Example: +17 / +3 Quotient = +5 Remainder = +2
2. **Positive** Dividend and **Negative** Divisor
   - ✧ Example: +17 / –3 Quotient = –5 Remainder = +2
3. **Negative** Dividend and **Positive** Divisor
   - ✧ Example: –17 / +3 Quotient = –5 Remainder = –2
4. **Negative** Dividend and **Negative** Divisor
   - ✧ Example: –17 / –3 Quotient = +5 Remainder = –2

The following equation must always hold:

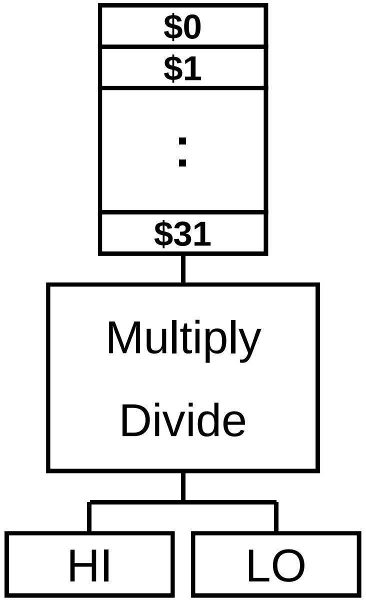
Dividend = Quotient × Divisor + Remainder

Next . . .

- Unsigned Integer Multiplication
- Signed Integer Multiplication
- Faster Integer Multiplication
- Integer Division
- **Integer Multiplication and Division in MIPS**

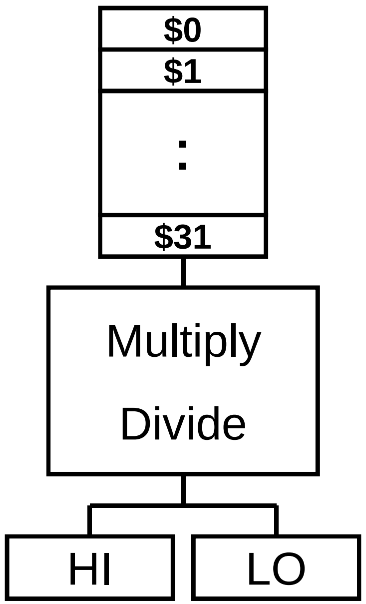
Integer Multiplication in MIPS

- Multiply instructions
  - `mult  Rs, Rt` **Signed multiplication**
  - `multu Rs, Rt` **Unsigned multiplication**
- 32-bit multiplication produces a 64-bit Product
- Separate pair of 32-bit registers
  - **HI = high-order 32-bit of product**
  - **LO = low-order 32-bit of product**
- MIPS also has a special `mul` instruction
  - `mul  Rd, Rs, Rt` `Rd = Rs × Rt`
  - Copy **LO** into destination register **Rd**
  - Useful when the product is small (32 bits) and **HI** is not needed



Integer Division in MIPS

- Divide instructions
  - `div   Rs, Rt` **Signed division**
  - `divu Rs, Rt` **Unsigned division**
- Division produces quotient and remainder
- Separate pair of 32-bit registers
  - **HI = 32-bit remainder**
  - **LO = 32-bit quotient**
  - If divisor is 0 then result is **unpredictable**
- Moving data from **HI, LO** to MIPS registers
  - `mfhi Rd` (**Rd = HI**)
  - `mflo Rd` (**Rd = LO**)



Integer Multiply and Divide Instructions

Instruction	Meaning	Format					
mult Rs, Rt	HI, LO = Rs $\times_s$ Rt	Op = 0	Rs	Rt	0	0	0x18
multu Rs, Rt	HI, LO = Rs $\times_u$ Rt	Op = 0	Rs	Rt	0	0	0x19
mul Rd, Rs, Rt	Rd = Rs $\times_s$ Rt	0x1c	Rs	Rt	Rd	0	2
div Rs, Rt	HI, LO = Rs $/_s$ Rt	Op = 0	Rs	Rt	0	0	0x1a
divu Rs, Rt	HI, LO = Rs $/_u$ Rt	Op = 0	Rs	Rt	0	0	0x1b
mfhi Rd	Rd = HI	Op = 0	0	0	Rd	0	0x10
mflo Rd	Rd = LO	Op = 0	0	0	Rd	0	0x12
mthi Rs	HI = Rs	Op = 0	Rs	0	0	0	0x11
mtlo Rs	LO = Rs	Op = 0	Rs	0	0	0	0x13

$\times_s$ = Signed multiplication, $\times_u$ = Unsigned multiplication

$/_s$ = Signed division, $/_u$ = Unsigned division

NO arithmetic exception can occur

String to Integer Conversion

- Consider the conversion of string "91052" into an integer

`'9'`	`'1'`	`'0'`	`'5'`	`'2'`

- How to convert the string into an integer?
- Initialize: **sum = 0**
- Load each character of the string into a register
  - Check if the character is in the range: **`'0'`** to **`'9'`**
  - Convert the character into a **digit** in the range: **`0`** to **`9`**
  - Compute: **sum = sum * 10 + digit**
  - Repeat until end of string or a non-digit character is encountered
- To convert "91052", initialize sum to 0 then …
  - sum = 9, then 91, then 910, then 9105, then 91052

String to Integer Conversion Function

```
#--------------------------------------------------------------
# str2int:  Convert a string of digits into unsigned integer
# Input:    $a0 = address of null terminated string
# Output:   $v0 = unsigned integer value
#--------------------------------------------------------------
str2int:
      li     $v0, 0           # Initialize: $v0 = sum = 0
      li     $t0, 10          # Initialize: $t0 = 10
L1:   lb     $t1, 0($a0)      # load $t1 = str[i]
      blt    $t1, '0', done   # exit loop if ($t1 < '0')
      bgt    $t1, '9', done   # exit loop if ($t1 > '9')
      addiu  $t1, $t1, -48    # Convert character to digit
      mul    $v0, $v0, $t0    # $v0 = sum * 10
      addu   $v0, $v0, $t1    # $v0 = sum * 10 + digit
      addiu  $a0, $a0, 1      # $a0 = address of next char
      j      L1               # loop back
done: jr     $ra              # return to caller
```

Integer to String Conversion

- Convert an unsigned 32-bit integer into a string
- How to obtain the decimal digits of the number?
  - Divide the number by 10, Remainder = decimal digit (0 to 9)
  - Convert decimal digit into its ASCII representation ('0' to '9')
  - Repeat the division until the quotient becomes zero
  - Digits are computed **backwards** from least to most significant
- Example: convert 2037 to a string
  - Divide 2037/10 quotient = 203 remainder = 7 char = '7'
  - Divide 203/10 quotient = 20 remainder = 3 char = '3'
  - Divide 20/10 quotient = 2 remainder = 0 char = '0'
  - Divide 2/10 quotient = 0 remainder = 2 char = '2'

Integer to String Conversion Function

```
#--------------------------------------------------------------
# int2str:  Converts an unsigned integer into a string
# Input:    $a0 = value, $a1 = buffer address (12 bytes)
# Output:   $v0 = address of converted string in buffer
#--------------------------------------------------------------
int2str:
      li    $t0, 10          # $t0 = divisor = 10
      addiu $v0, $a1, 11     # start at end of buffer
      sb    $zero, 0($v0)    # store a NULL character
L2:   divu  $a0, $t0         # LO  = value/10, HI = value%10
      mflo  $a0              # $a0 = value/10
      mfhi  $t1              # $t1 = value%10
      addiu $t1, $t1, 48     # convert digit into ASCII
      addiu $v0, $v0, -1     # point to previous byte
      sb    $t1, 0($v0)      # store character in memory
      bnez  $a0, L2          # loop if value is not 0
      jr    $ra              # return to caller
```