CS / EE 320
Computer Organization and Assembly Language

Lecture 10

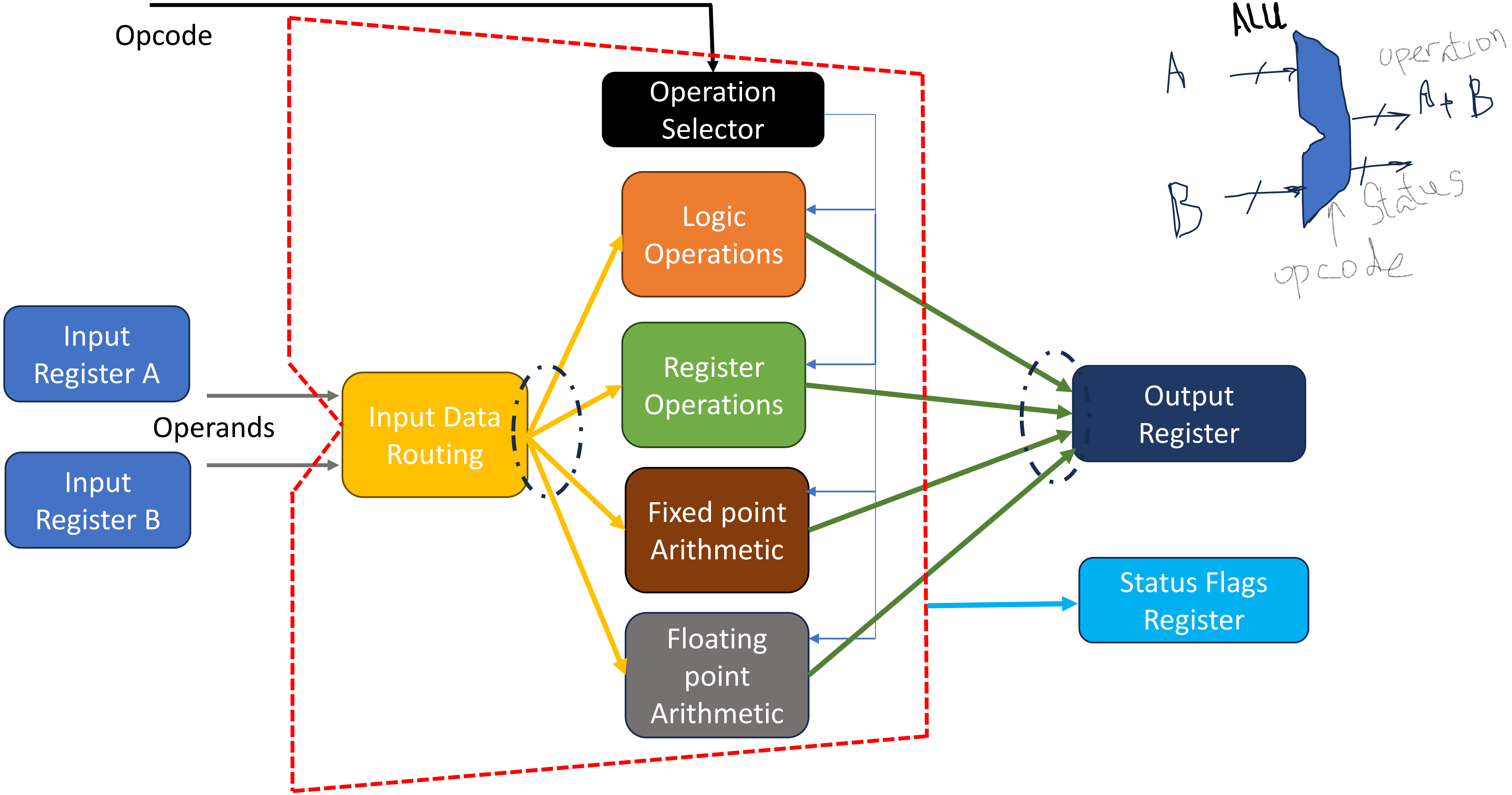
Shahid Masud

Topics: Arithmetic and Logic Operations, Multipliers, Dividers, Simple ALU Design

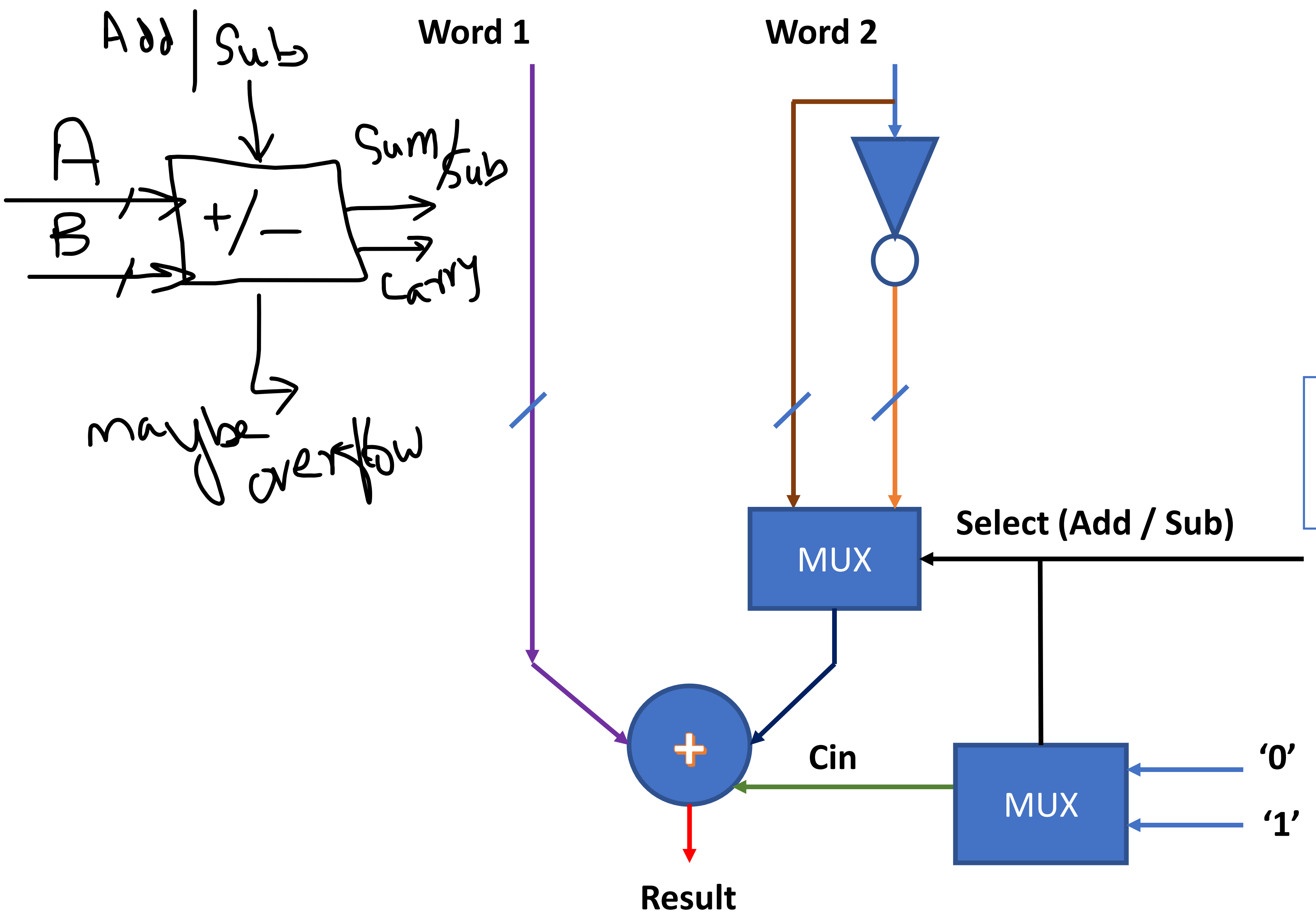
Topics

- Recap – Basic ALU operations, digital logic building blocks
- Binary Multiplier – Pencil and paper method
- Binary Multiplier Hardware
- Binary Divider – Method
- Binary Divider Hardware
- SIMD and Sub-Word Parallelism
- 32 – bit ALU Design with functions AND, OR, NAND, NOR, NOT, Adder, A+B, A-B, A<B, Check Overflow

Dissecting an ALU



Combined Adder and 2’s Complement Subtraction



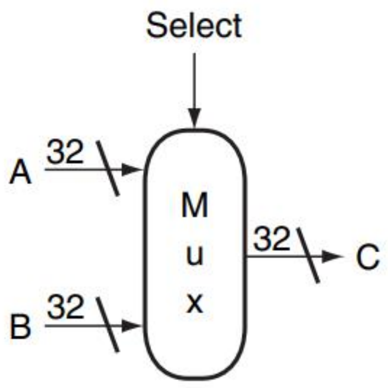
For 2’s Complement,
Select the path through inverter and
Cin = ‘1’

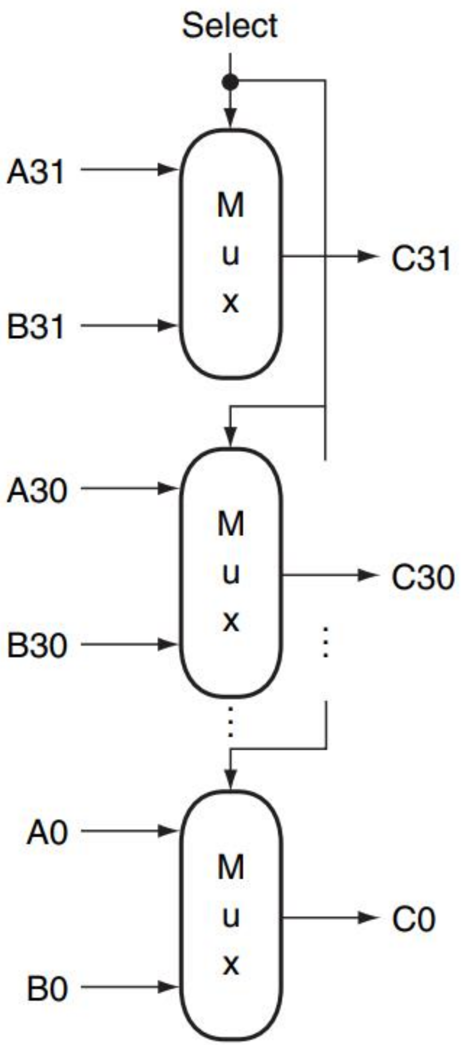
Select ‘1’ for 2’s Complement

Other Digital Components in CPU Design

- Decoders
  - Select (turn to ‘1’) one out of eight available outputs. Rest of the outputs remain at ‘0’.
  - Used in selecting a particular I/O Device or Memory based on Address lines
- Multiplexers
  - Has several inputs and one output
  - Based on Select lines, connects one of the inputs to the output while other inputs are isolated from the output
- Registers
  - Temporary Storage made out of flipflop devices
  - The data moves from input to the output only at the edge of Clock
  - The output is isolated from input when there is no clock; state is retained

Multiplexers and Array of Multiplexers

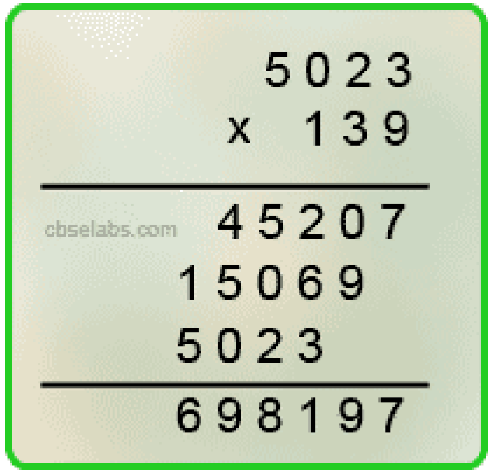




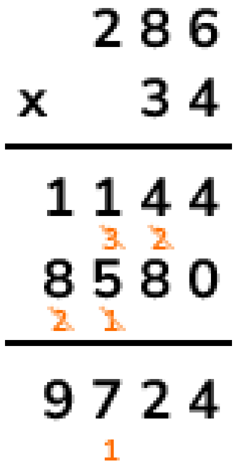
a. A 32-bit wide 2-to-1 multiplexor

b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors

Binary Multiplication – Pencil and Paper Method



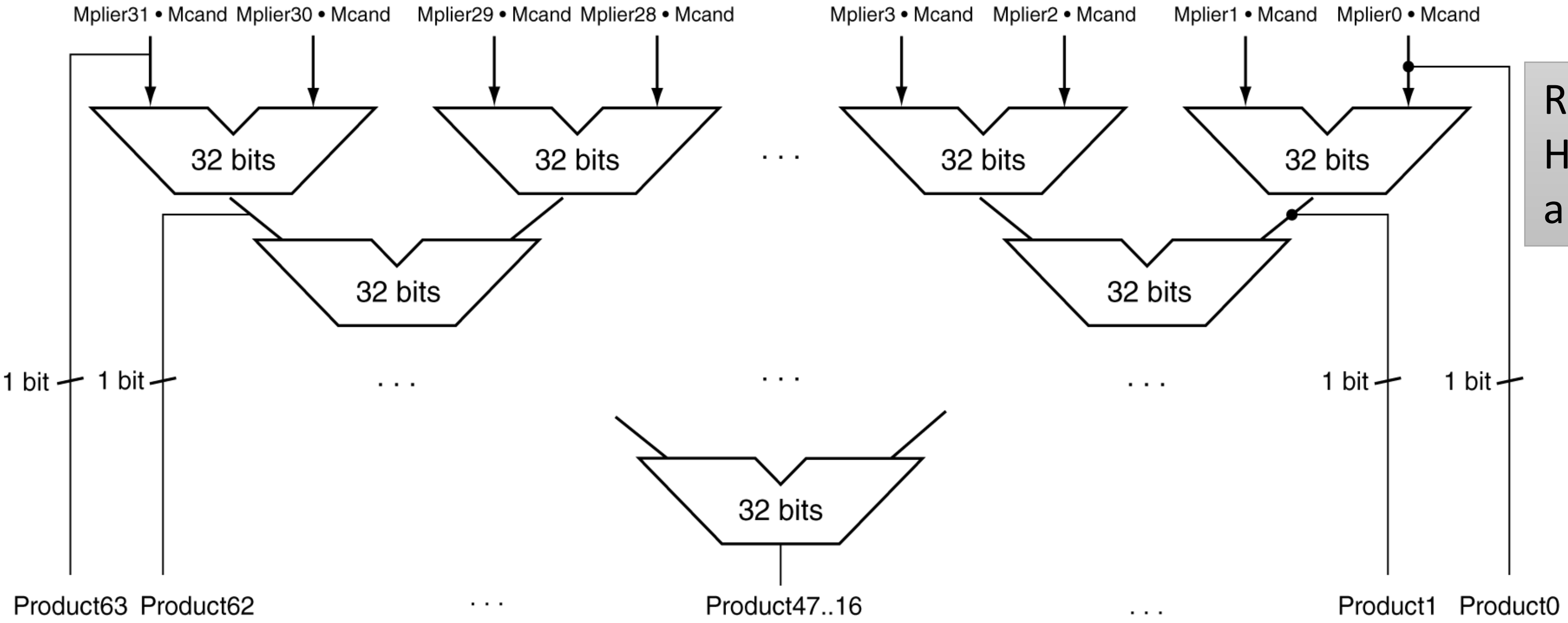
```
     5023
   x  139
  -------
    45207
   15069
   5023
  -------
   698197
```



```
     286
 x    34
 -------
    1144
     3 2
    8580
   2 1
 -------
    9724
     1
```

Faster Multiplier in Hardware

- Uses multiple adders
  - Cost/performance tradeoff

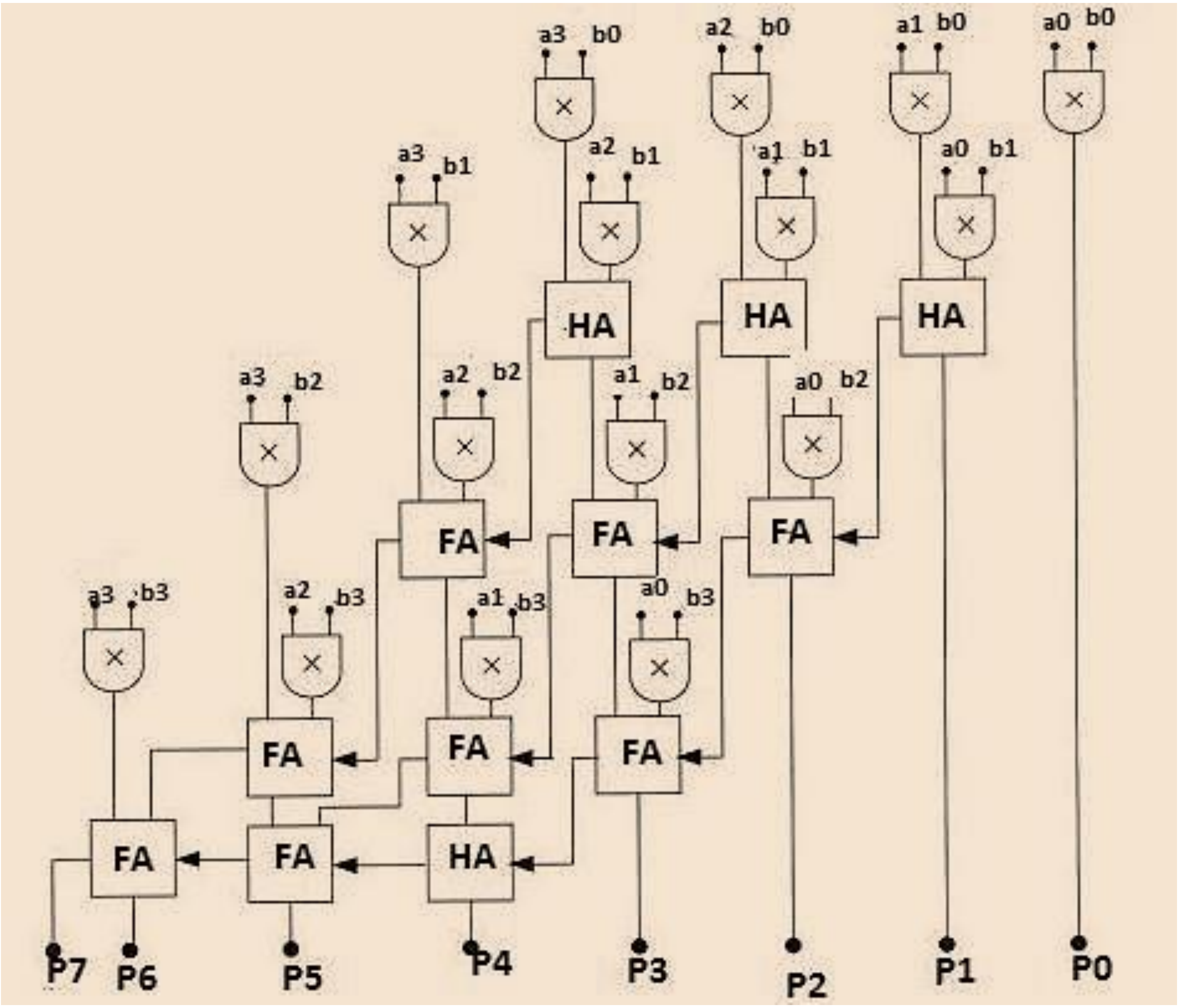


Rather that using a single 32-bit Adder, this Hardware 'unrolls the loop' to use 31 Adders and then organizes them to minimize delay

- Can be pipelined
  - Several multiplication performed in parallel

4-Bit Array Multiplier connected as rows of AND and ADDER

Multiplicand				A_3	A_2	A_1	A_0
Multiplier			×	B_3	B_2	B_1	B_0
				A_3B_0	A_2B_0	A_1B_0	A_0B_0
			A_3B_1	A_2B_1	A_1B_1	A_0B_1	0
		A_3B_2	A_2B_2	A_1B_2	A_0B_2	0	0
	A_3B_3	A_2B_3	A_1B_3	A_0B_3	0	0	0
Cout	P_6	P_5	P_4	P_3	P_2	P_1	P_0



HA = Half Adder

FA = Full Adder

Px = Partial Product

Examples continued

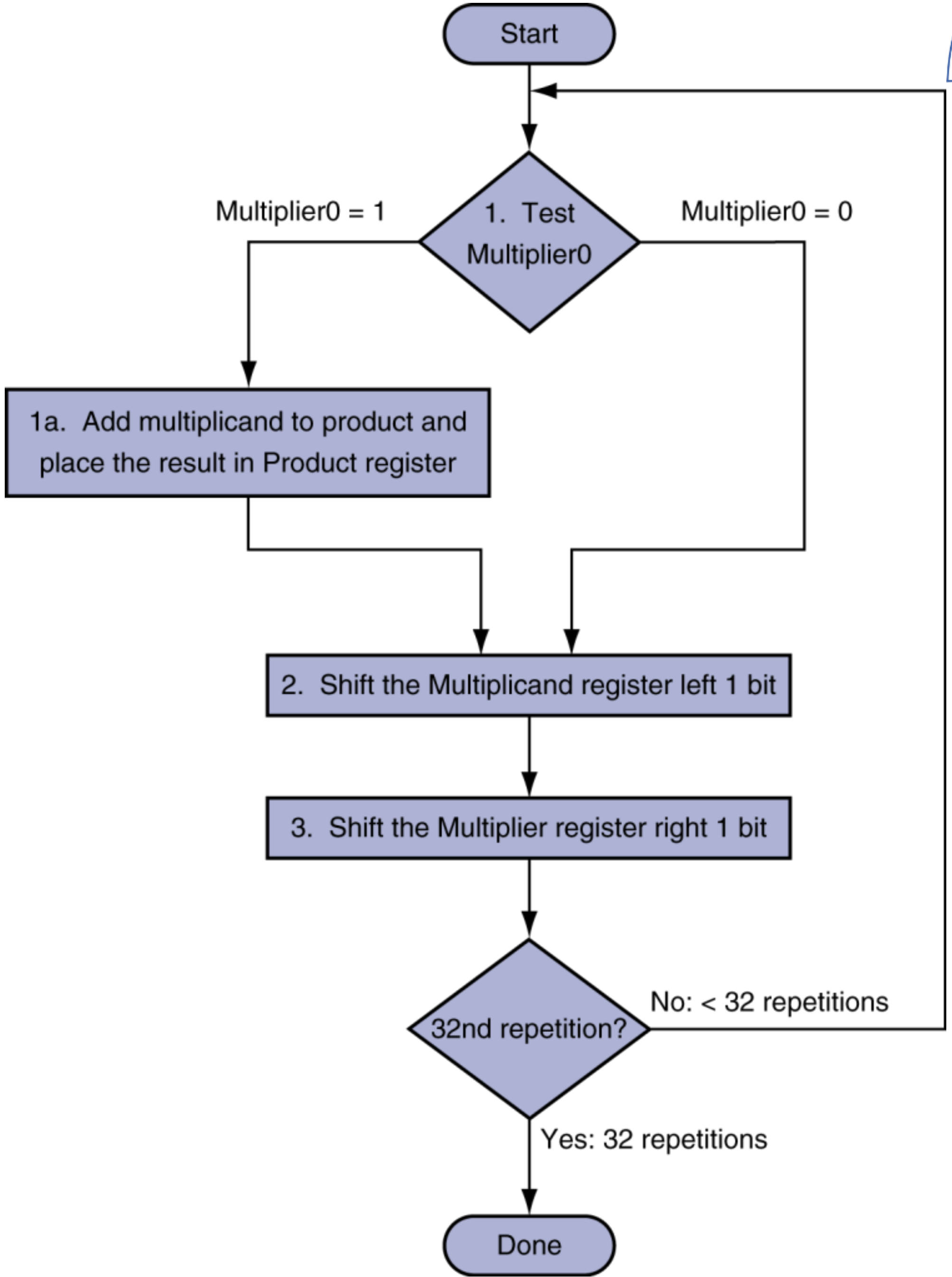
				1	1	0	1	Multiplicand
			x	1	0	1	1	Multiplier
				1	1	0	1	
			1	1	0	1	X	Shift Left by one
		1	0	0	1	1	1	Partial product after first step
		0	0	0	0	X	X	Another shift left
		1	0	0	1	1	1	Partial product after second step
	1	1	0	1	X	X	X	Another shift left
1	0	0	0	1	1	1	1	Partial product after final step

Answer = $(10001111)_2 = (143)_{10}$

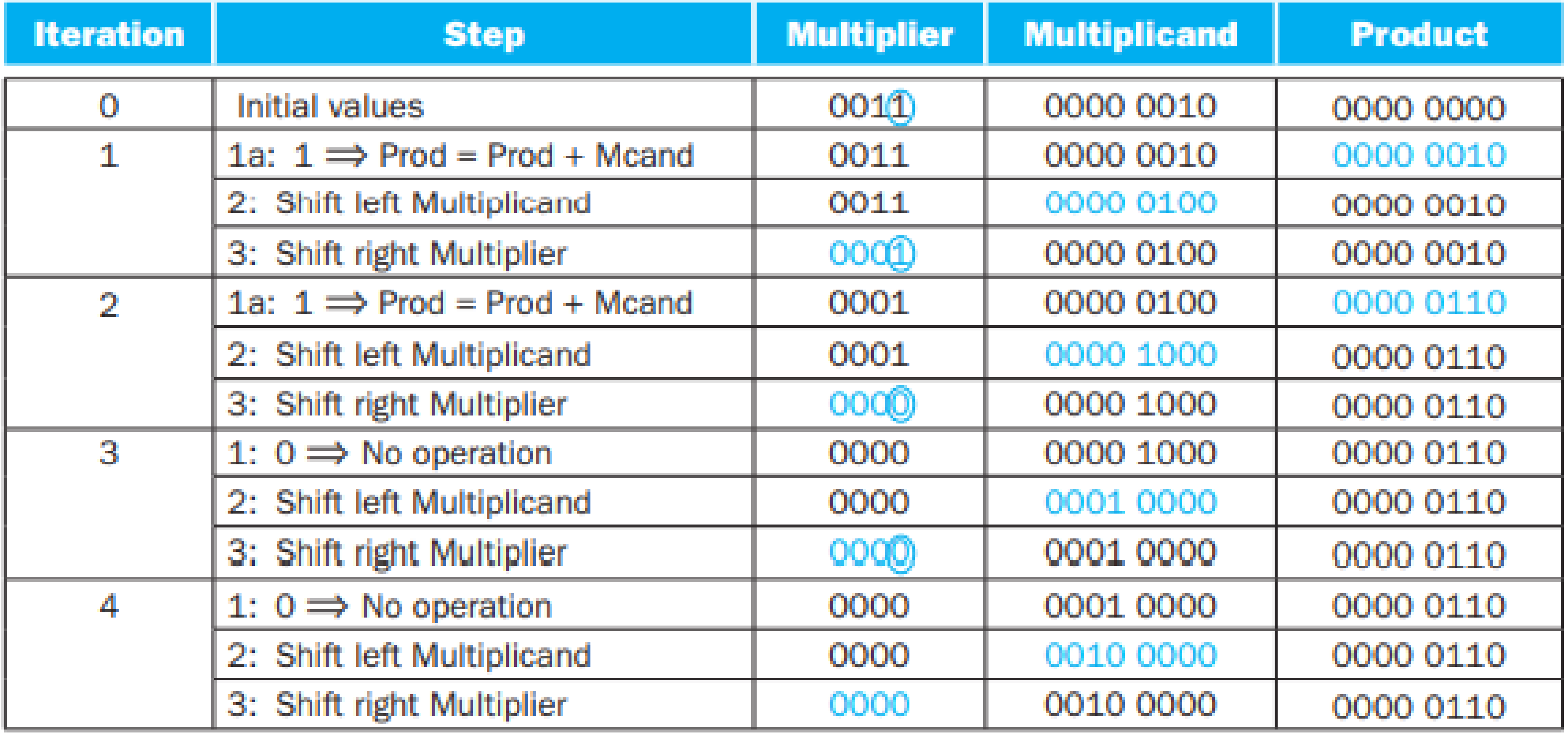
Direct Multiplication Hardware

Basic Multiplication Algorithm using Hardware

- If the least significant bit of the multiplier is ‘1’, add the multiplicand to the product
- If not, go to the next step
- Shift the multiplicand left and the multiplier to the right in the next two steps
- These steps are repeated 32 times


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Application of Multiplication Algorithm



Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⟹ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⟹ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⟹ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⟹ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

The bit examined to determine the next step is circled

Execution of Sequential Multiplication using Add/Shift

(unsigned numbers e.g.)

1011 Multiplicand (11 dec)

x 1101 Multiplier (13 dec)

1011 Partial products

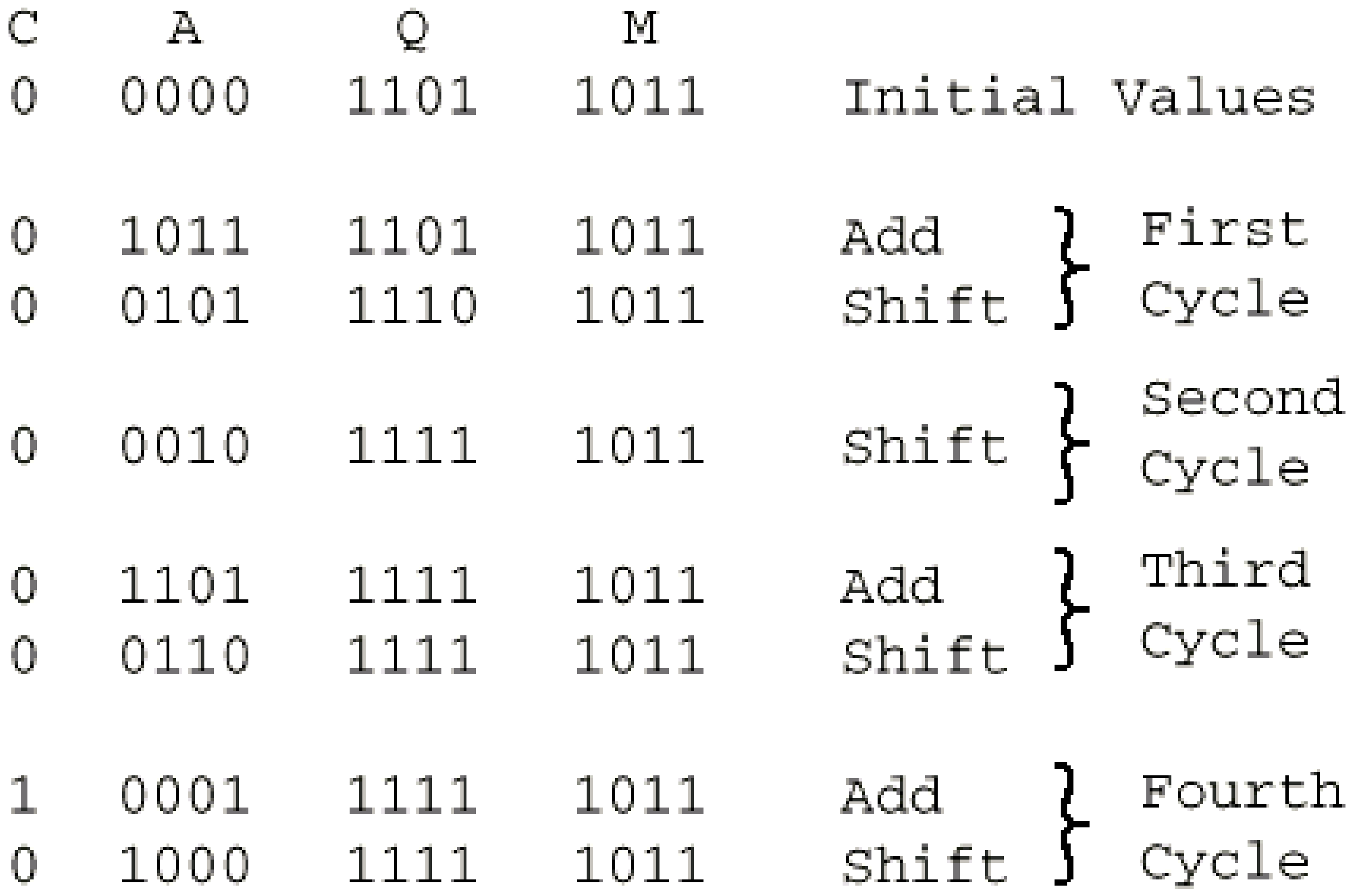
0000 Note: if multiplier bit is 1 copy

1011 multiplicand (place value)

1011 otherwise zero

10001111 Product (143 dec)

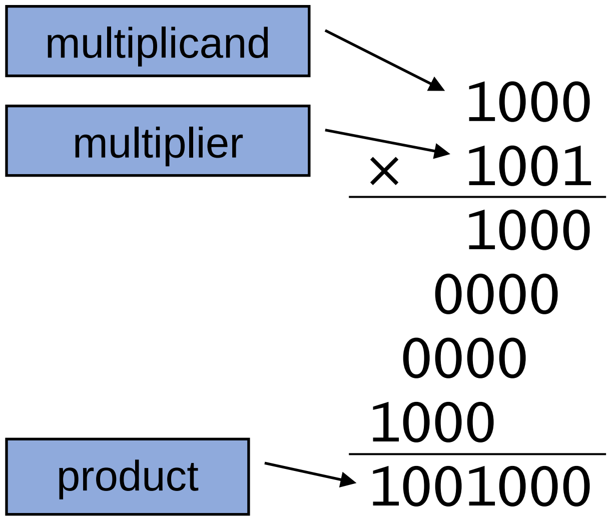
Note: need double length result



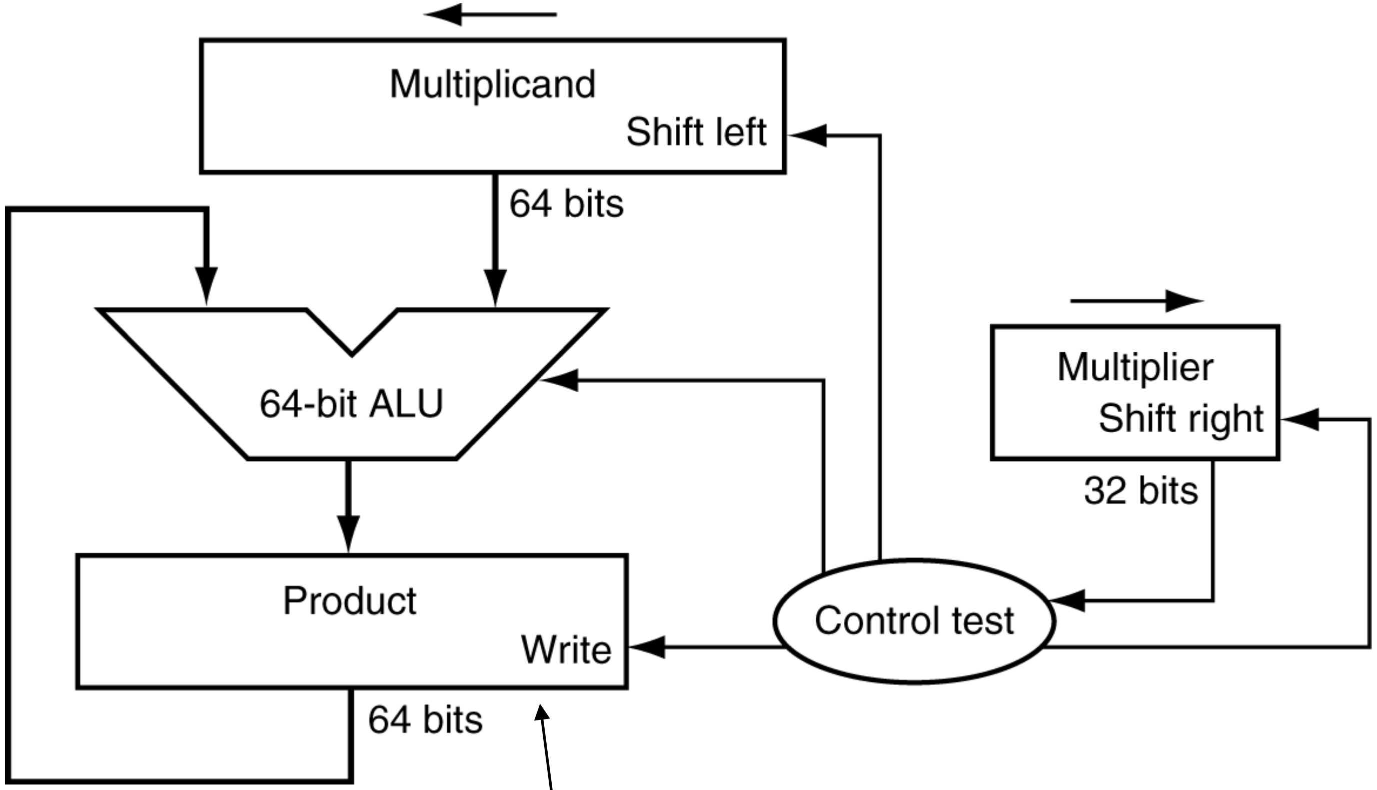
C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

Multiplication Hardware

- Start with long-multiplication approach



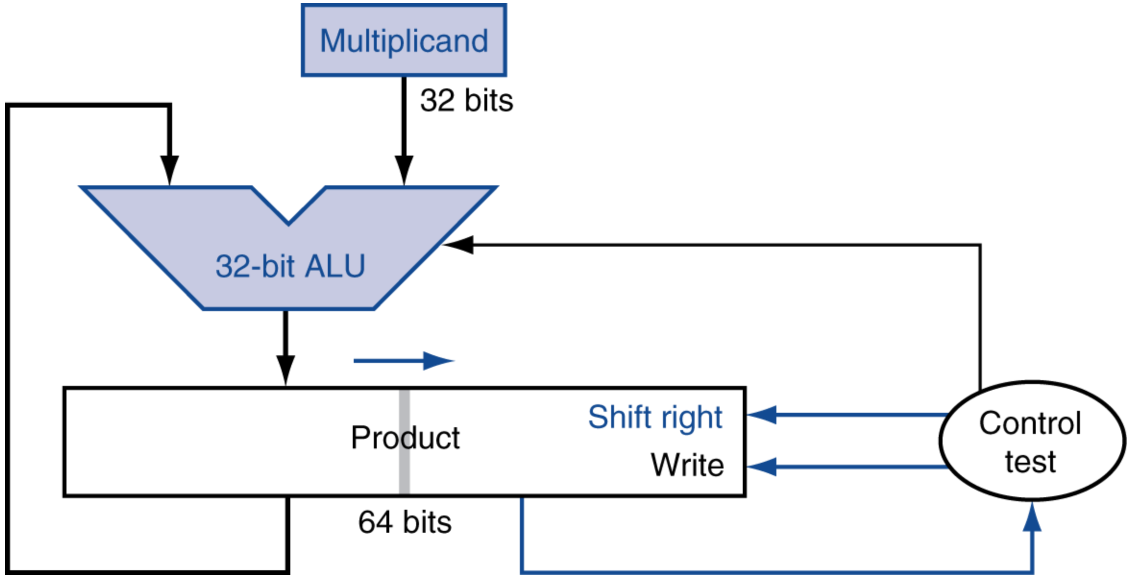
Length of product is the sum of operand lengths



Initially 0

Refined Multiplication Hardware

- Perform steps in parallel: add/shift

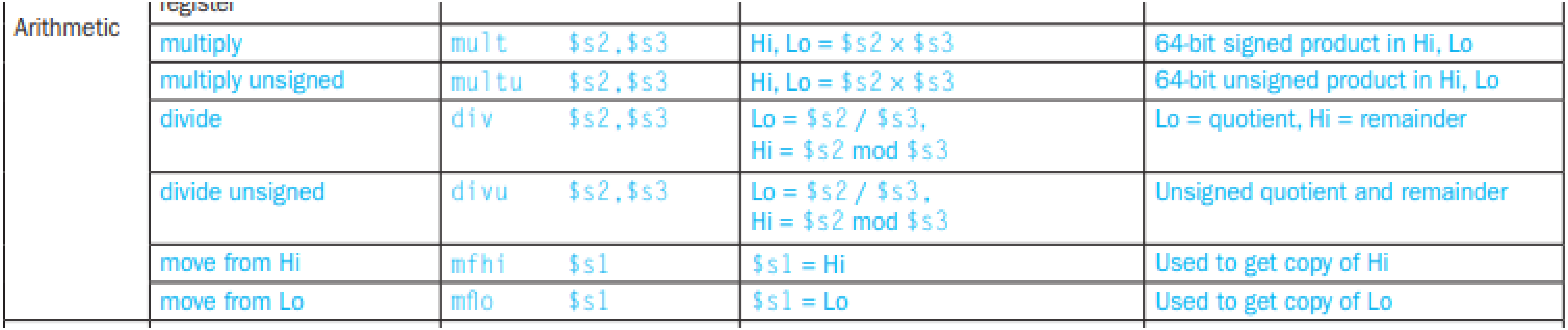


Improved Version of Multiplication Hardware

- The Multiplicand Register, ALU and Multiplier Register are 32 bits wide
- Only the Product Register is 64 bits
- Now the Product is shifted right
- Separate Multiplier register is not required, the multiplier is placed in the right half of the Product register

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

Assembly Instructions for Multiplication



	register				
Arithmetic	multiply	mult	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide	div	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
	move from Hi	mfhi	$s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo	$s1	$s1 = Lo	Used to get copy of Lo

Note: Use of Hi and Lo registers in CPU

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

- MIPS provides a separate pair of 32-bit registers to contain the 64 bit Product, called Hi and Lo registers
- MIPS has two instructions: multiply (mult) and multiply unsigned (multu)
- To fetch the integer 32 bit Product, the programmer uses move from lo (mflo).
- MIPS Assembler also generates a pseudo instruction for multiplier that specifies mflo and mfhi registers

Binary Division – Pencil and Paper Method

Division Operation in Decimal Numbers

Division of 274 ÷ 13

				2	1	Quotient	
Divisor	1	3	2	7	4		Dividend
		-	-2	6			
				1	4		
			-	1	3		
		Remainder			1	Rem	

Decimal Division – another example

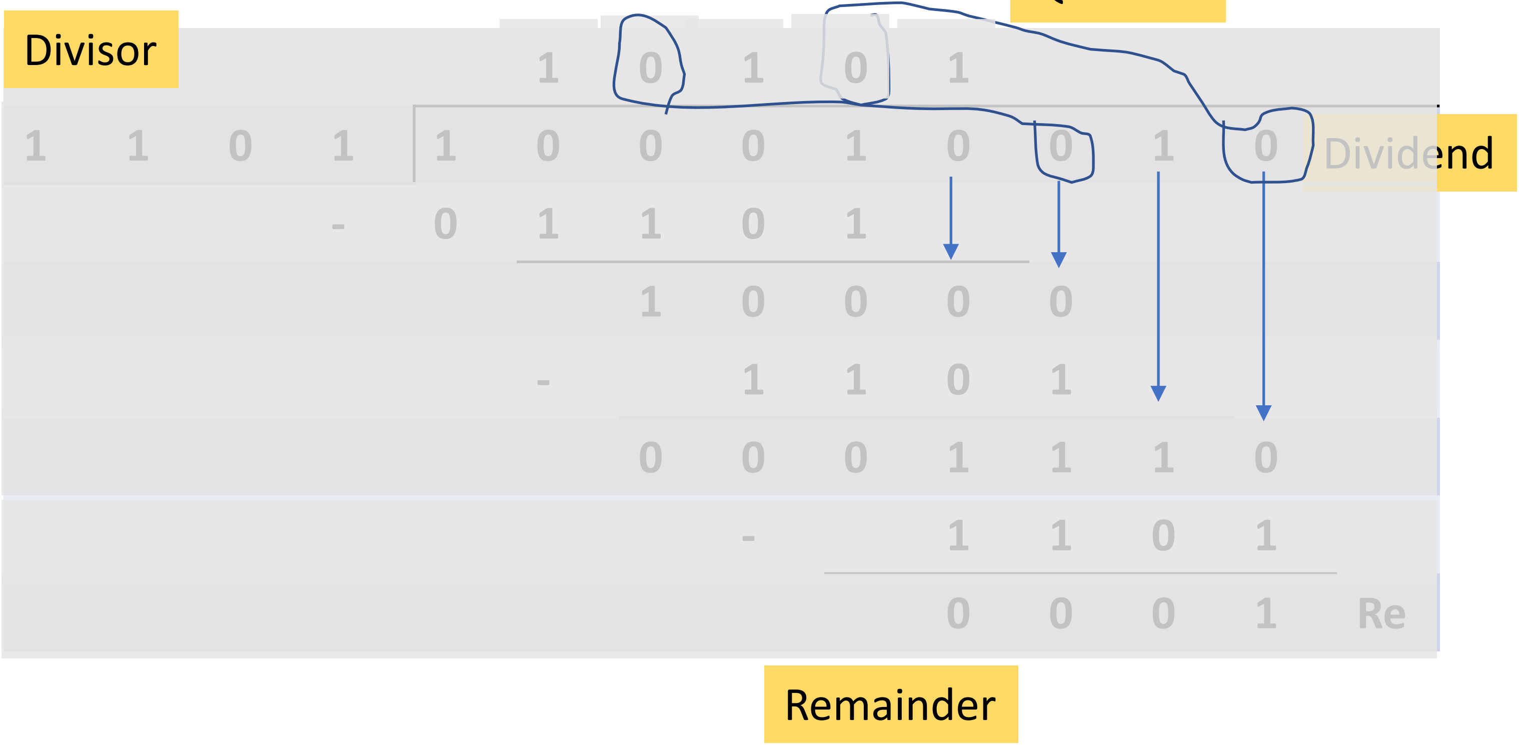
		1	0	0	4	Quotient
Divisor	8	8	0	3	5	Dividend
	-	8				
		0	0	3	5	
		-		3	2	
Remainder					3	

Division Operation in Binary – Example 1

Remainder

Division of 274 ÷ 13

```
                 2    1
 1    3   |  2   7    4
      -     -2   6
                 1    4
           -     1    3
                      1    Rem
```



Quotient

Divisor

Dividend

```
                      1   0   1   0   1
1   1   0   1  |  1   0   0   0   1   0   0   1   0
            -     0   1   1   0   1
                          1   0   0   0   0
                      -       1   1   0   1
                          0   0   0   1   1   1   0
                                  -       1   1   0   1
                                          0   0   0   1   Re
```

Remainder

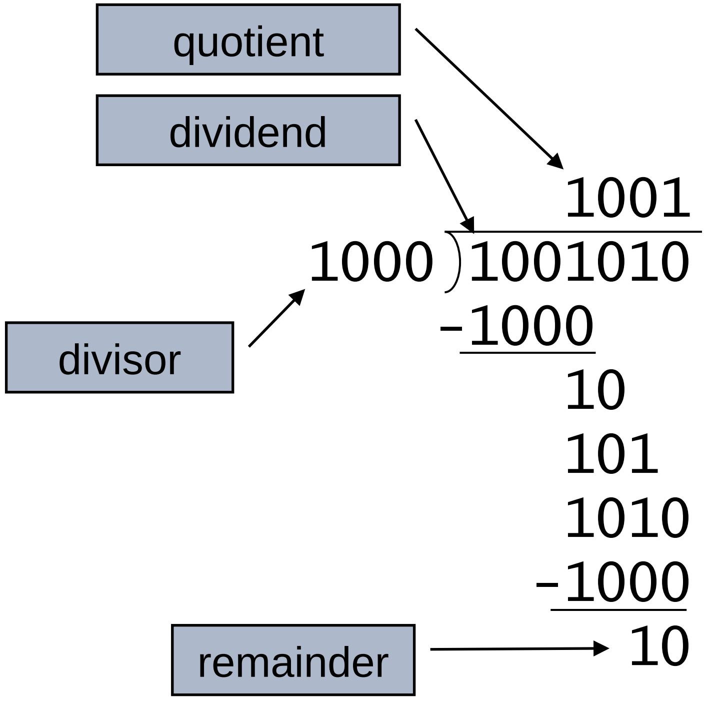
Division Operation in Binary – Example 2

Division of 299 ÷ 15

```
                1   9
          -------------
 1   5  | 2   9   9
--------
      -   2   8   5
          -----------
              1   4   Rem
```

```
                                 Quotient
Divisor             1   0   0   1   1
                  ----------------------------------
 1   1   1   1  | 1   0   0   1   0   1   0   1   1   Dividend
----------------
             -      1   1   1   1
                  ----------------
                    0   0   1   1   1   0   1
                    -   0   0   1   1   1   1
                        --------------------------
                        0   0   1   1   1   0   1
                            -       1   1   1   1
                                ----------------
                Remainder           1   1   1   0   Re
```

Division – developing an Algorithm



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

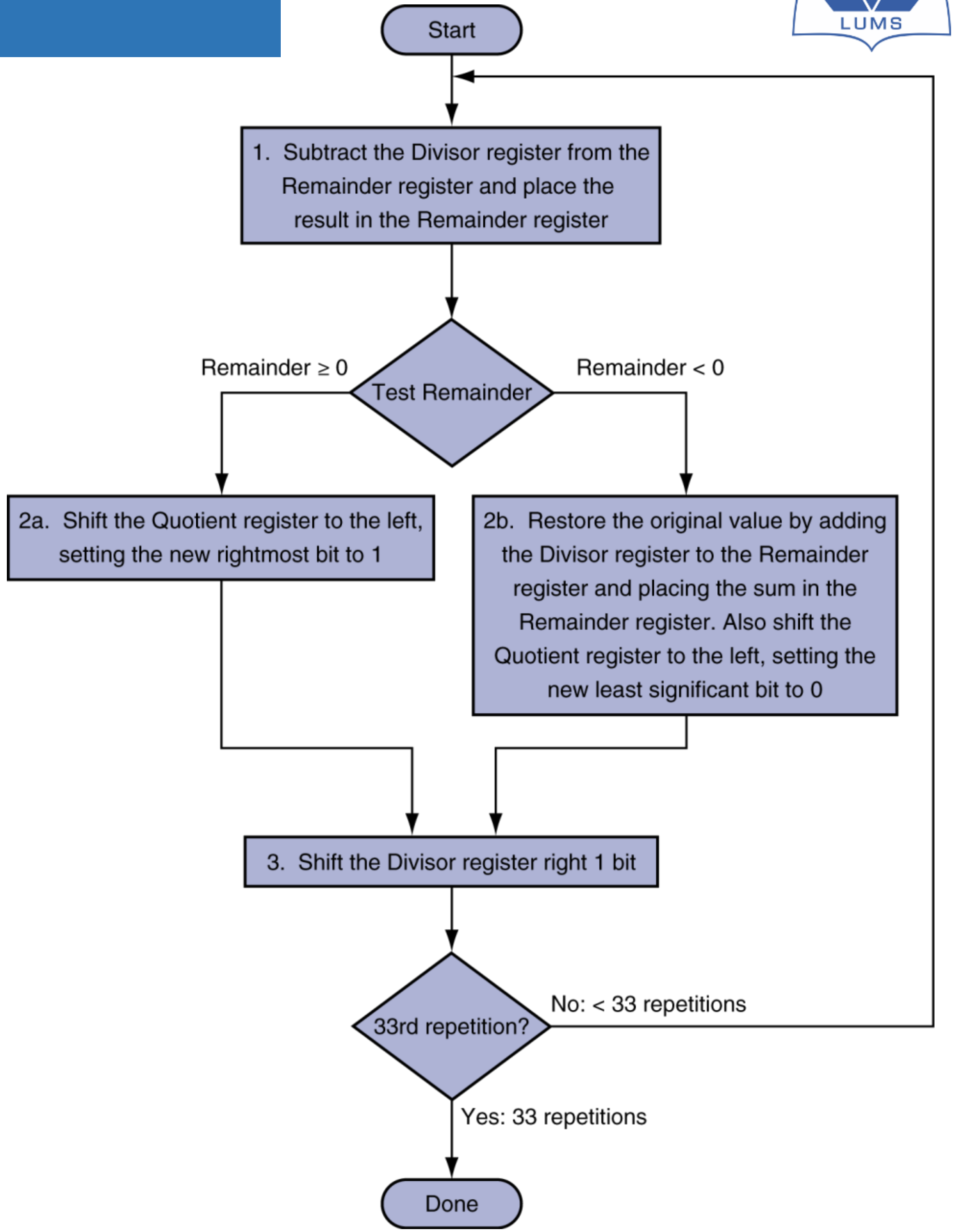
- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

SUBTRACTION: Use 2's Compliment and ADD Circuits

Simple Division Algorithm

Division Algorithm using the hardware

- If the remainder is positive, the divisor goes into the dividend, so step 2a generates a ‘1’ in the quotient
- A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and
- Adds the divisor to the remainder. This reverses the subtraction of step 1.
- The final shift in step 3 aligns the divisor properly, relative to the dividend for the next iteration
- These steps are repeated 33 times



Start

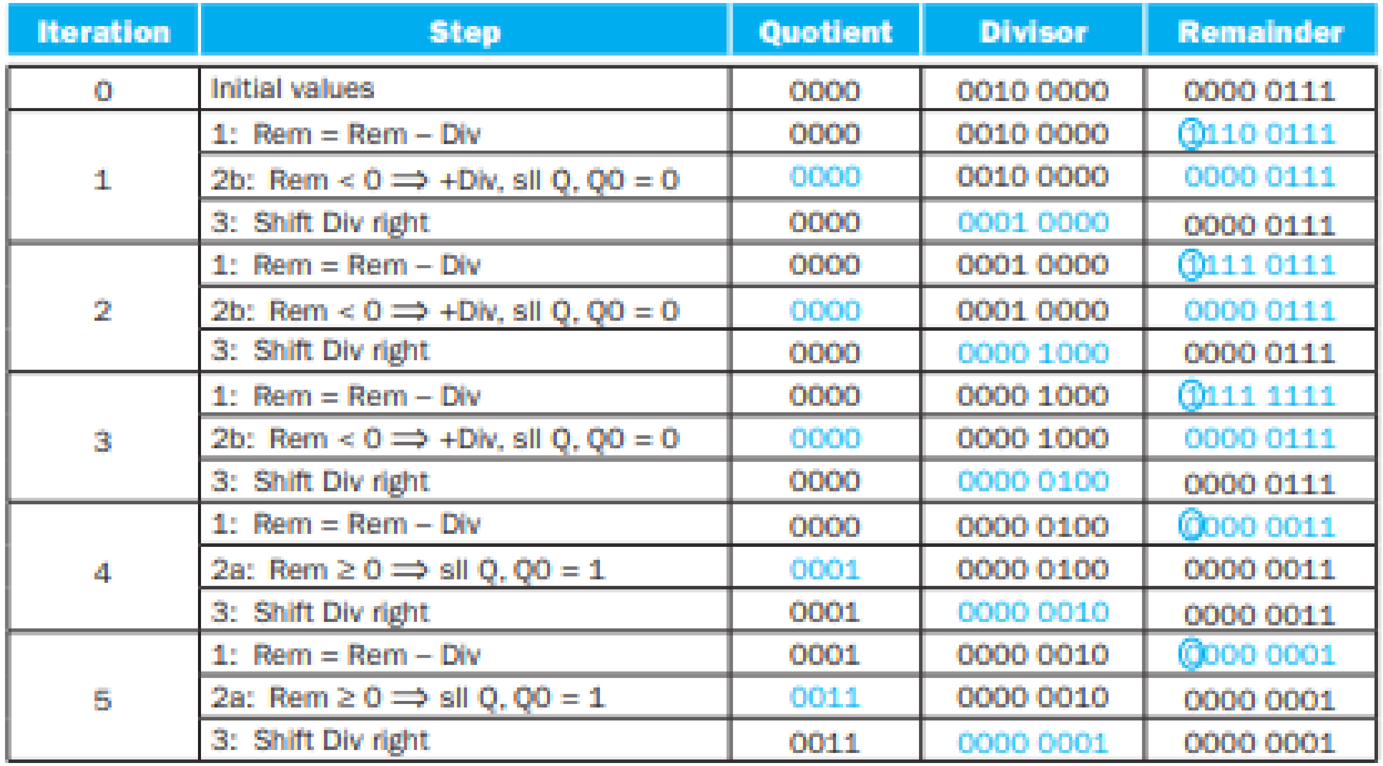
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register

2a. Shift the Quotient register to the left, setting the new rightmost bit to 1

2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0

3. Shift the Divisor register right 1 bit

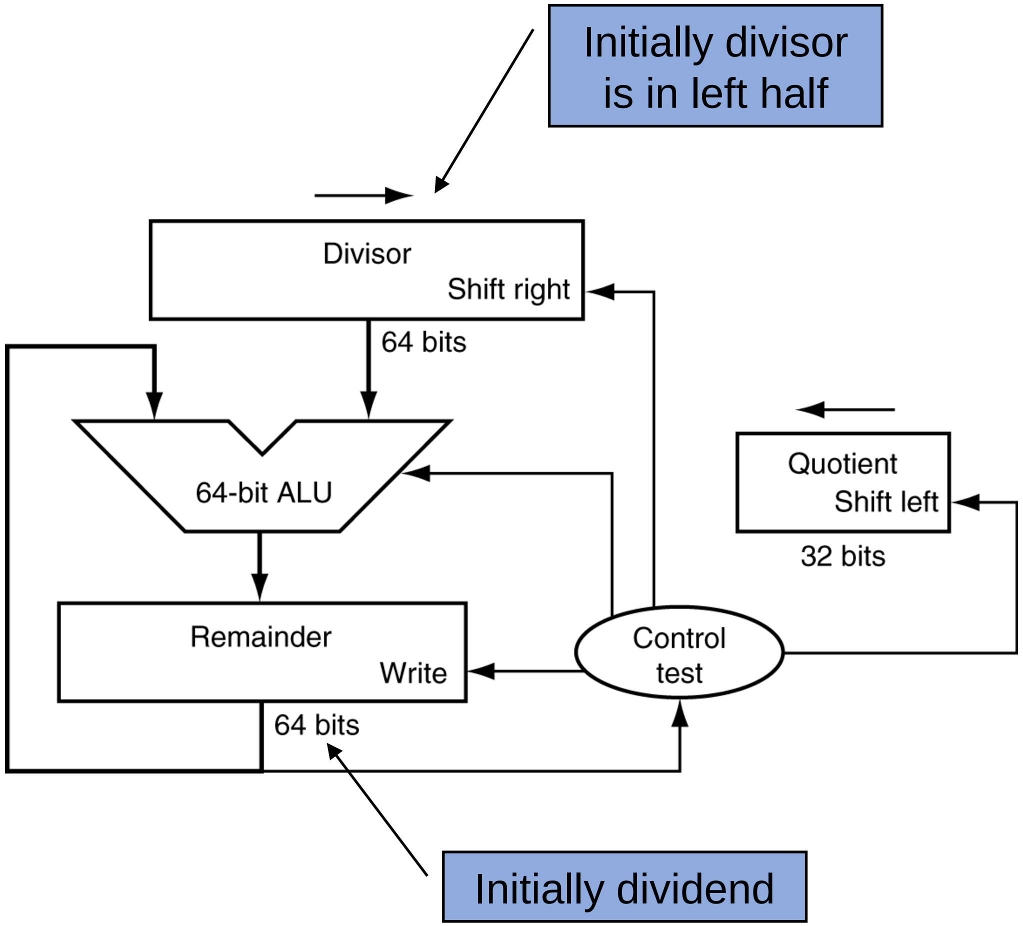
Applying Sequential Division Algorithm



Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

The bit examined to determine the next step is encircled

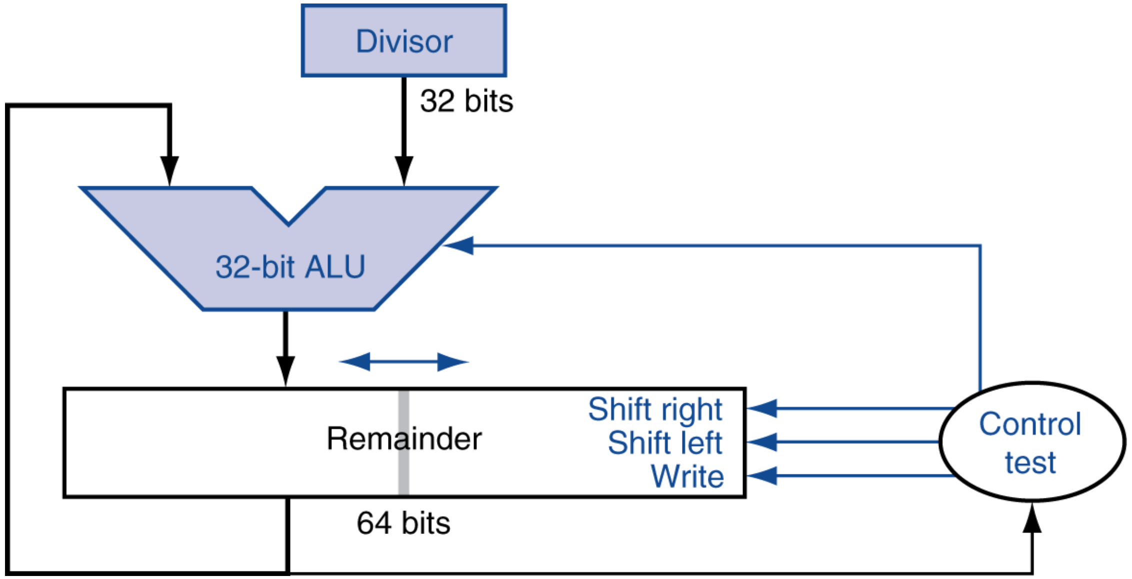
Simple Division Hardware Implementation



Simple Division Hardware

- The Divisor register, ALU and Remainder register are all 64 bits wide
- Only the Quotient register is 32 bits
- The 32 bits Divisor starts in the left half of the Divisor register and it is shifted right 1 bit in each iteration
- The Remainder is initialized with the Dividend
- Control decides when to shift the Divisor and Quotient registers and when to write the new value into the remainder register

Optimized Divider



Improved Version of Division Hardware

- The Divisor register, ALU, and Quotient register are all 32 bits wide
- Only the Remainder register is 64 bits
- The Quotient Register is combined with the right half of the Remainder register

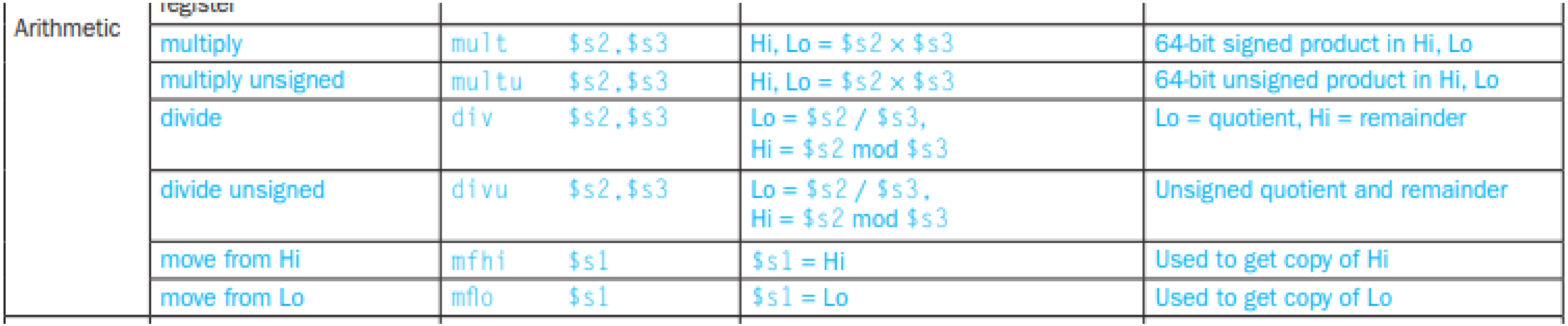
- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

Right Shift and Division

- Left shift by *i* places multiplies an integer by 2^i
- Right shift divides by 2^i **?**
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., –5 / 4
    - 11111011_2 >> 2 = 11111110_2 = –2
    - Rounds toward –∞
  - c.f. 11111011_2 >>> 2 = 00111110_2 = +62

Remember, each left shift is multiplication by 2

MIPS Assembly Instructions for Division



Arithmetic	register			
	multiply	mult $s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo

	divide	div $s2,$s3	Lo = $s2 / $s3,
			Hi = $s2 mod $s3
				Lo = quotient, Hi = remainder

	divide unsigned	divu $s2,$s3	Lo = $s2 / $s3,
			Hi = $s2 mod $s3
				Unsigned quotient and remainder

	move from Hi	mfhi $s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo $s1	$s1 = Lo	Used to get copy of Lo

Note: Use of Hi and Lo registers in CPU

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

Divide instructions in MIPS

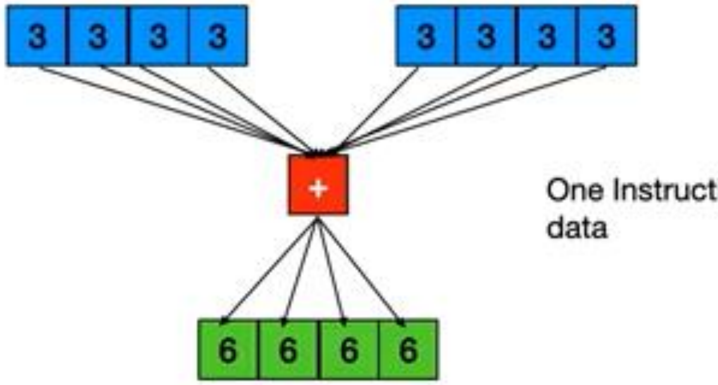
- The same sequential hardware can be used for both multiply and divide
- The only requirement is that 64 bit register than can shift left or right and a 32 bit ALU that adds or subtracts
- MIPS uses the 32 bit Hi and 32 bit Lo registers for both multiply and Divide instructions
- Hi contains the Remainder
- Lo contains the Quotient after the Divide instruction is completed
- div instruction is for signed numbers
- divu instruction is unsigned numbers
- MIPS assembler has pseudo instructions that specify three registers to place desired result in a general-purpose register

Subword Parallellism

- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
  - Example: 128-bit adder:
    - Sixteen 8-bit adds
    - Eight 16-bit adds
    - Four 32-bit adds
- Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD)

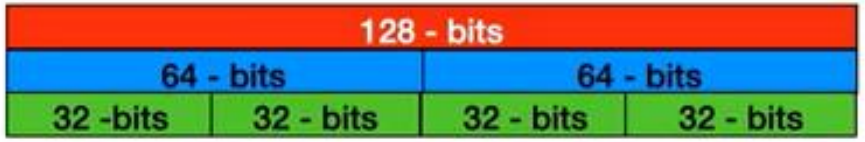
SIMD Introduction

Single Instruction Multi Data

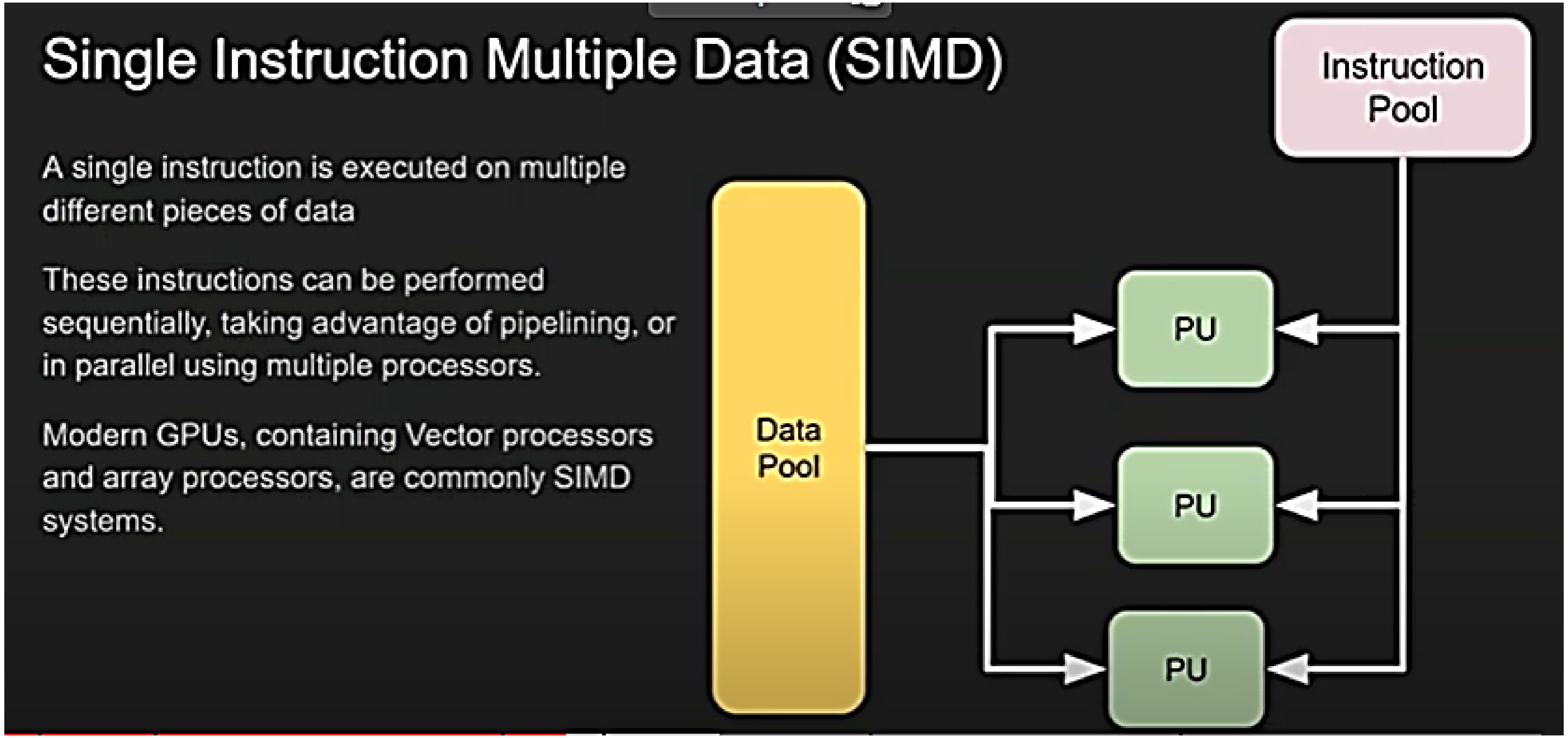


One Instruction per vector of data

SIMD Registers



SIMD Processing



Single Instruction Multiple Data (SIMD)

A single instruction is executed on multiple different pieces of data

These instructions can be performed sequentially, taking advantage of pipelining, or in parallel using multiple processors.

Modern GPUs, containing Vector processors and array processors, are commonly SIMD systems.

Streaming SIMD Extension 2 (SSE2)

- Adds 4 × 128-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 × 64-bit double precision
  - 4 × 32-bit double precision
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data

Arithmetic for Multimedia

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow, result is largest representable value
    - 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent

Readings

- Chap 3 of P&H Textbook
- Appendix C of P&H Textbook