Computer Architecture & Assembly Language 14:332:331

Lecture 5

Arithmetic for Computers

Naghmeh Karimi

Fall 16

Adapted from *Computer Organization and Design*, *5th Edition*, Patterson & Hennessy, © 2013, Elsevier,
and Computer Organization and Design, *4th Edition*, Patterson & Hennessy, © 2008, Elsevier
and Mary Jane Irwin's slides from Penn State University.

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

§ 3.1 Introduction

Machine Number Representation

- Bits are just bits (have no inherent meaning)

– conventions define the relationships between bits and numbers

- Binary numbers (base 2) - integers

0000 → 0001 → 0010 → 0011 → 0100 → 0101 → . . .

– in decimal from 0 to 2^n-1 for n bits

- Of course, it gets more complicated

– storage locations (e.g., register file words) are finite, so have to worry about overflow (i.e., when the number is too big to fit into 32 bits)

– have to be able to represent negative numbers, e.g., how do we specify -8 in

```
        addi   $sp, $sp, -8        #$sp = $sp - 8
```

– in real systems have to provide for more than just integers, e.g., fractions and real numbers (and floating point) and alphanumeric (characters)

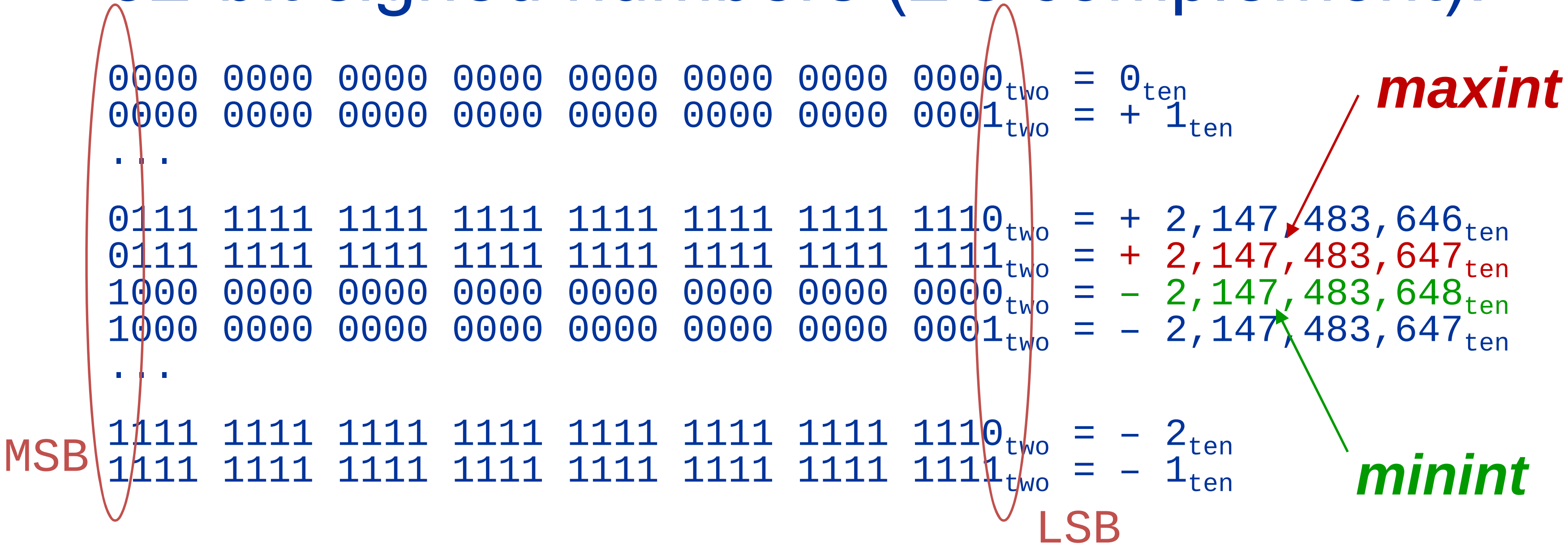
Possible Representations

Sign Mag.	Two's Comp.	One's Comp.
	1000 = -8	
1111 = -7	1001= -7	1000 = -7
1110 = -6	1010 = -6	1001 = -6
1101 = -5	1011 = -5	1010 = -5
1100 = -4	1100 = -4	1011 = -4
1011 = -3	1101 = -3	1100 = -3
1010 = -2	1110 = -2	1101 = -2
1001 = -1	1111 = -1	1110 = -1
1000 = -0		1111 = -0
0000 = +0	0000 = 0	0000 = +0
0001 = +1	0001 = +1	0001 = +1
0010 = +2	0010 = +2	0010 = +2
0011 = +3	0011 = +3	0011 = +3
0100 = +4	0100 = +4	0100 = +4
0101 = +5	0101 = +5	0101 = +5
0110 = +6	0110 = +6	0110 = +6
0111 = +7	0111 = +7	0111 = +7

- Issues:
  - balance
  - number of zeros
  - ease of operations
- Which one is best? Why?

Number Representations

- 32-bit signed numbers (2’s complement):



0000 0000 0000 0000 0000 0000 0000 0000$_{two}$ = 0$_{ten}$

0000 0000 0000 0000 0000 0000 0000 0001$_{two}$ = + 1$_{ten}$

. . .

0111 1111 1111 1111 1111 1111 1111 1110$_{two}$ = + 2,147,483,646$_{ten}$

0111 1111 1111 1111 1111 1111 1111 1111$_{two}$ = + 2,147,483,647$_{ten}$

1000 0000 0000 0000 0000 0000 0000 0000$_{two}$ = – 2,147,483,648$_{ten}$

1000 0000 0000 0000 0000 0000 0000 0001$_{two}$ = – 2,147,483,647$_{ten}$

. . .

1111 1111 1111 1111 1111 1111 1111 1110$_{two}$ = – 2$_{ten}$

1111 1111 1111 1111 1111 1111 1111 1111$_{two}$ = – 1$_{ten}$

Two's Complement Operations

- Negating a two's complement number: complement all the bits and then add a 1
  - remember: “negate” and “invert” are quite different!

- Converting n-bit numbers into numbers with more than n bits:
  - MIPS 16-bit immediate gets converted to 32 bits for arithmetic
  - **sign extend** - copy the most significant bit (the sign bit) into the empty bits

```
0010  -> 0000 0010
1010  -> 1111 1010
```

  - sign extension versus zero extend `(lb vs. lbu)`

Addition & Subtraction

- Just like in grade school (carry/borrow 1s)

```
    0111            0111            0110
  + 0110          - 0110          - 0101
  ------          ------          ------
    1101            0001            0001
```

- Two's complement operations are easy
  - do subtraction by negating and then adding

```
   0111    →       0111
 - 0110    →     + 1010
 ------          ------
   0001           1 0001
```

- Overflow (result too large for finite computer word)
  - e.g., adding two n-bit numbers does not yield an n-bit number

```
    0111
  + 0001
  ------------
   1000
```

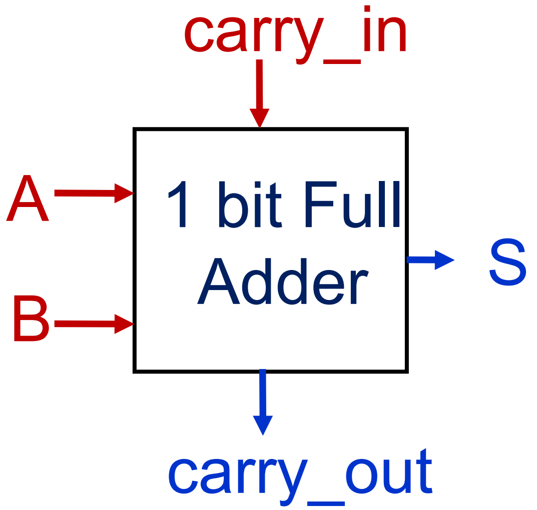
Dealing with Overflow

- 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

- MIPS signals overflow with an exception (aka interrupt) – an unscheduled procedure call where the EPC contains the address of the instruction that caused the exception

Building a 1-bit Binary Adder

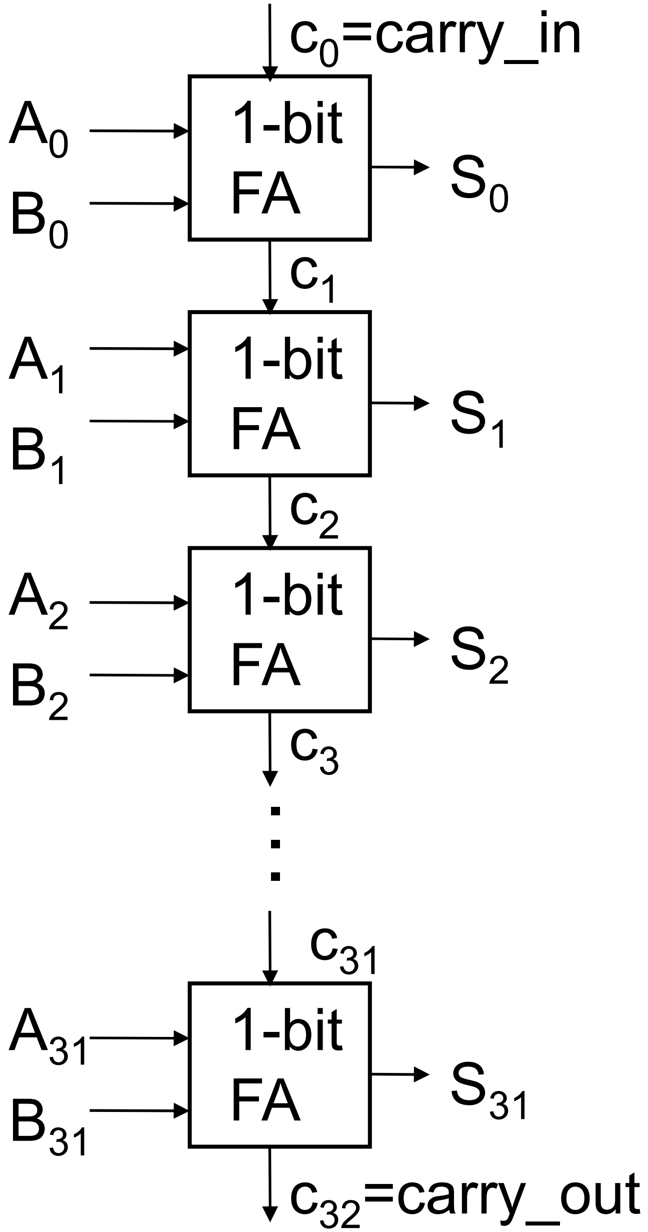


A	B	carry_in	carry_out	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S = A xor B xor carry_in
carry_out = A&B | A&carry_in | B&carry_in
(majority function)

- ❑ How can we use it to build a 32-bit adder?
- ❑ How can we modify it easily to build an adder/subtractor?

Building 32-bit Adder (Ripple Carry adder)

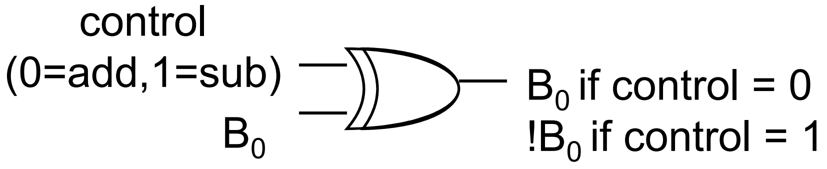


- Just connect the carry-out of each FA to the carry-in of the next level

- Ripple Carry Adder (RCA)
  - **advantage:** simple logic, so small (low cost)
  - **disadvantage:** slow and lots of glitching (so lots of energy consumption)

A 32-bit Ripple Carry Adder/Subtractor

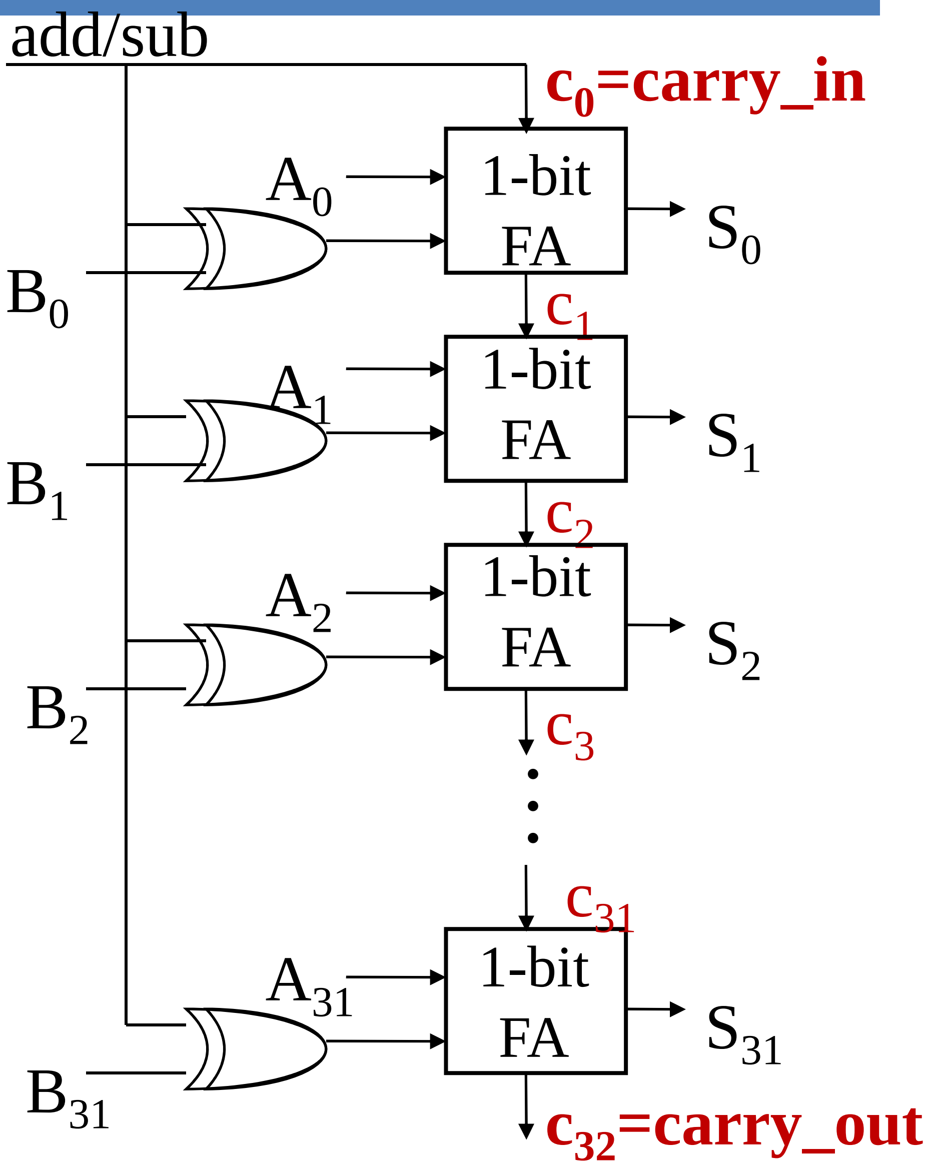
- Remember 2's complement subtraction is just
  - complement all the bits



  - add a 1 in the least significant bit

```
A     0111   →     0111
B  -  0110   →  + 1001
                      1
      0001      1 0001
```

- add/sub=0 → A+B
- add/sub=1 → A -B

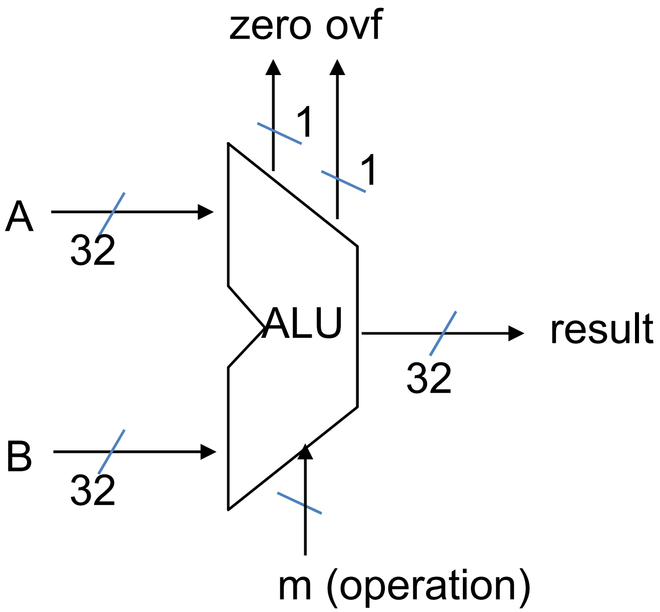


Dealing with Overflow

- Some languages (e.g., C) ignore overflow
  - Use MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS `add`, `addi`, `sub` instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

Arithmetic

- We Discussed
  - Instruction Set Architecture (ISA)
  - Assembly and machine language
- What's up ahead:
  - Implementing the architecture



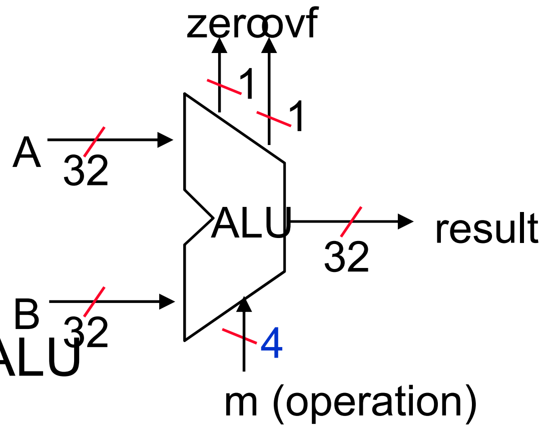
Goal: Design an ALU for the MIPS ISA

- Must support the Arithmetic/Logic operations of the ISA

- Tradeoffs of cost and speed based on frequency of occurrence, hardware budget

Review: MIPS Arithmetic Instructions

	31	25	20	15		5 0
R-Type:	op	Rs	Rt	Rd		funct
I-Type:	op	Rs	Rt	Immed 16		



- expand immediates to 32 bits before ALU
- 10 operations so can encode in 4 bits

Type	op	funct
ADD	00	100000
ADDU	00	100001
SUB	00	100010
SUBU	00	100011
AND	00	100100
OR	00	100101
XOR	00	100110
NOR	00	100111

Type	op	funct
	00	101000
	00	101001
SLT	00	101010
SLTU	00	101011
	00	101100

0	add
1	addu
2	sub
3	subu
4	and
5	or
6	xor
7	nor
a	slt
b	sltu

Design Trick: Divide & Conquer

- Break the problem into simpler problems, solve them and glue together the solution
- Example: assume the immediates have been taken care of before the ALU
  - now down to 10 operations
  - can encode in 4 bits

00	add
01	addu
02	sub
03	subu
04	and
05	or
06	xor
07	nor
12	slt
13	sltu

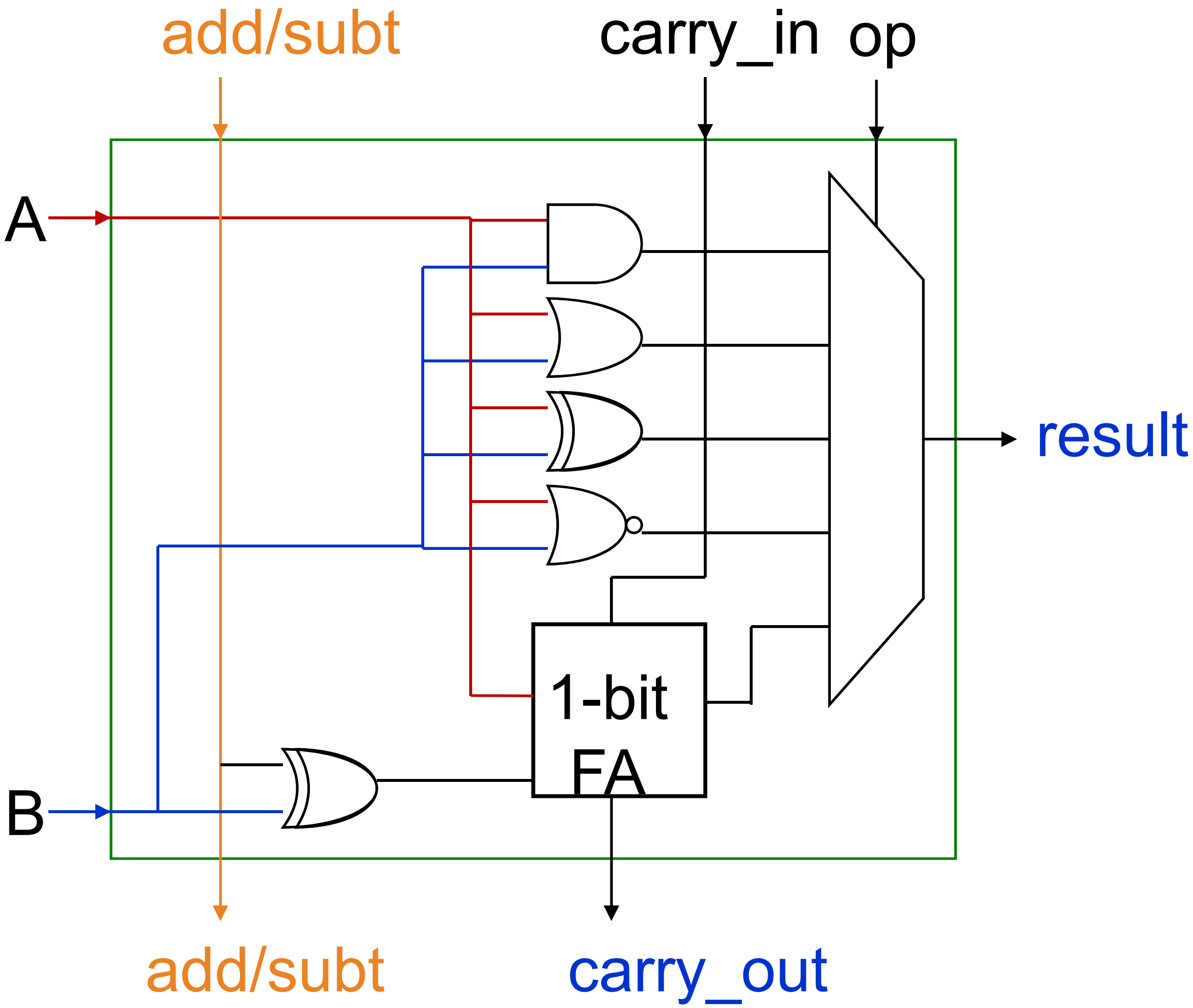
Logic Operations

- Logic operations operate on individual bits of the operand.

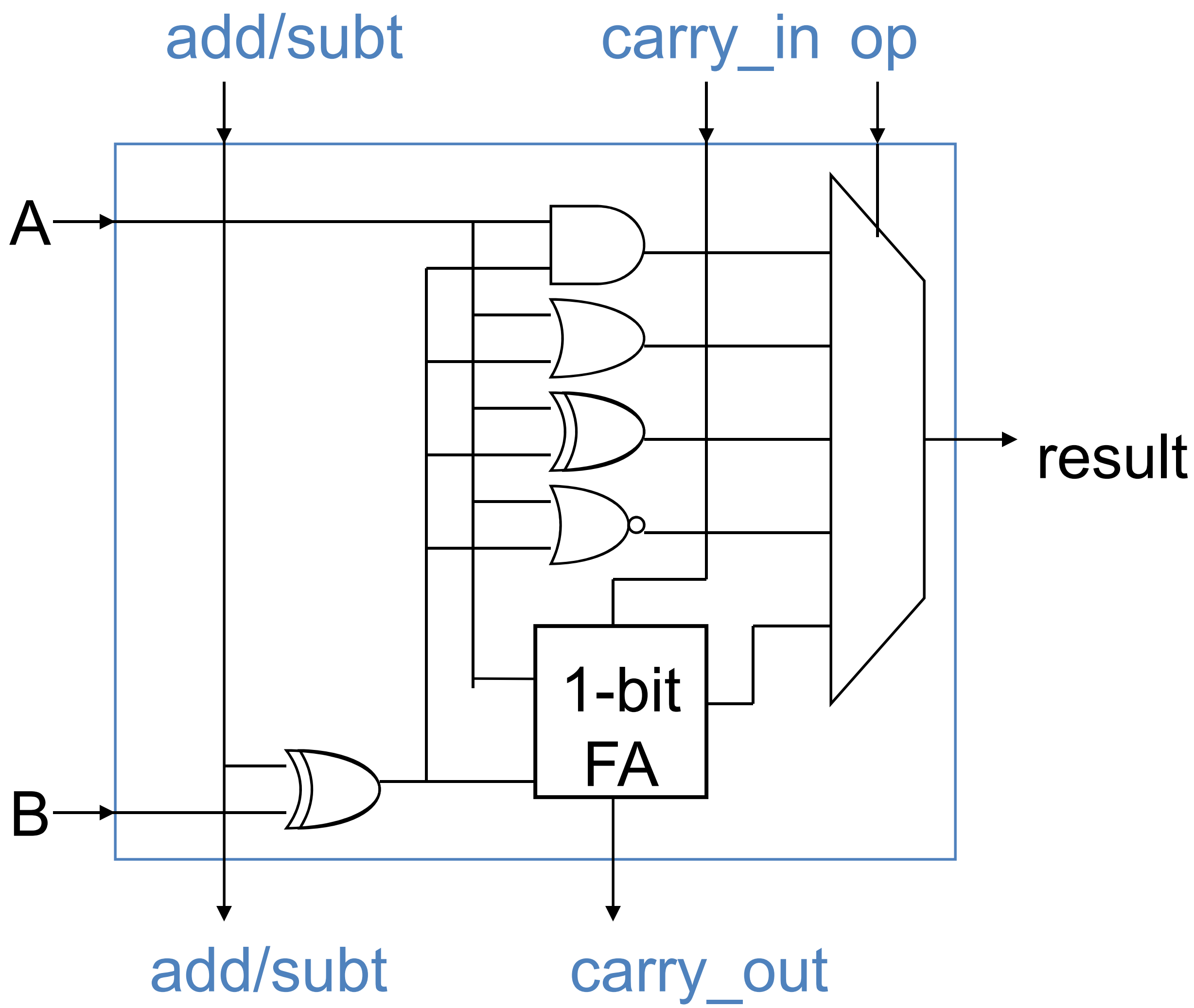
```
                   $t2 = 0…0 0000 1101 0000
                   $t1 = 0…0 0011 1100 0000
and  $t0, $t1, $t2 $t0 = ?

or   $t0, $t1 $t2  $t0 = ?

xor  $t0, $t1, $t2 $t0 = ?

nor  $t0, $t1, $t2 $t0 = ?
```

- How do we expand our FA design to handle the logic operations - and, or, xor, nor ?

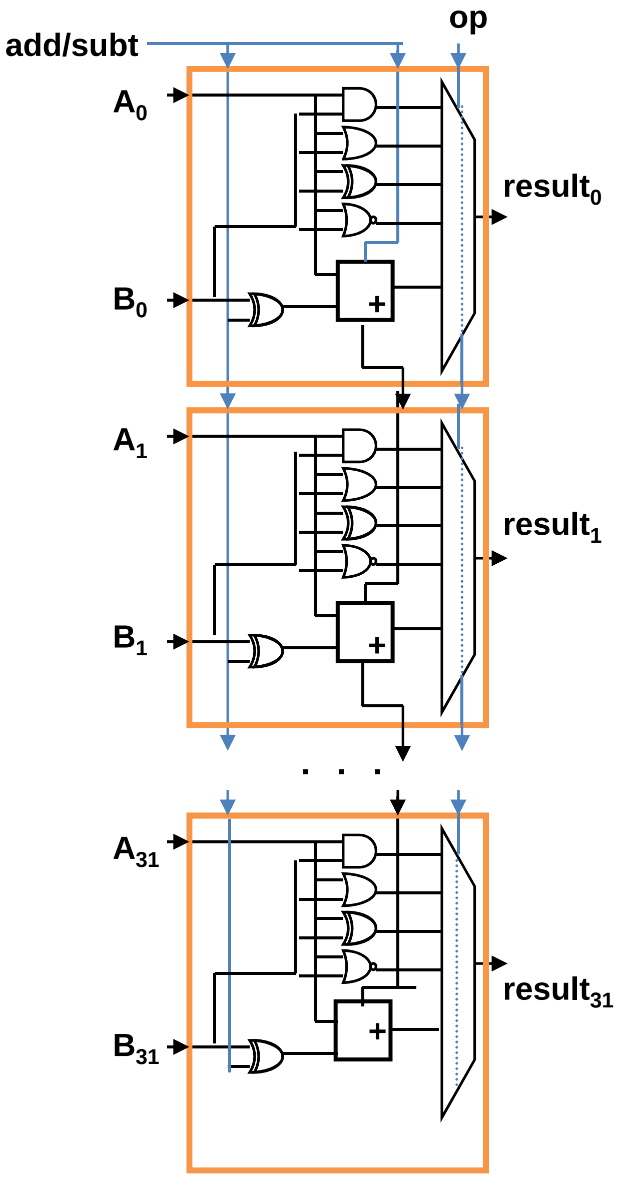
A Simple ALU Cell



A Simple ALU Cell (version 2)



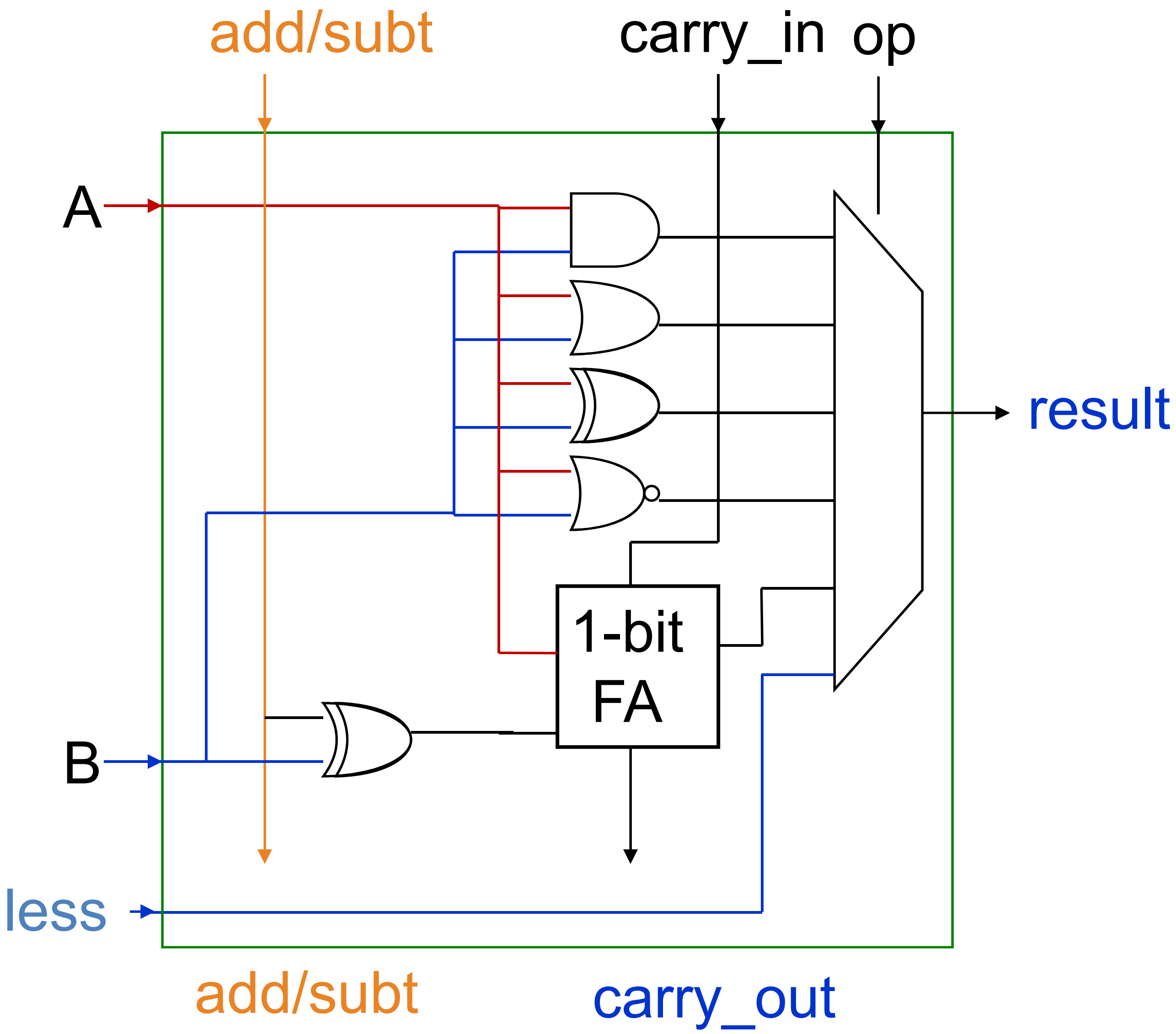
A Simple 32-bit ALU



Tailoring the ALU to the MIPS ISA

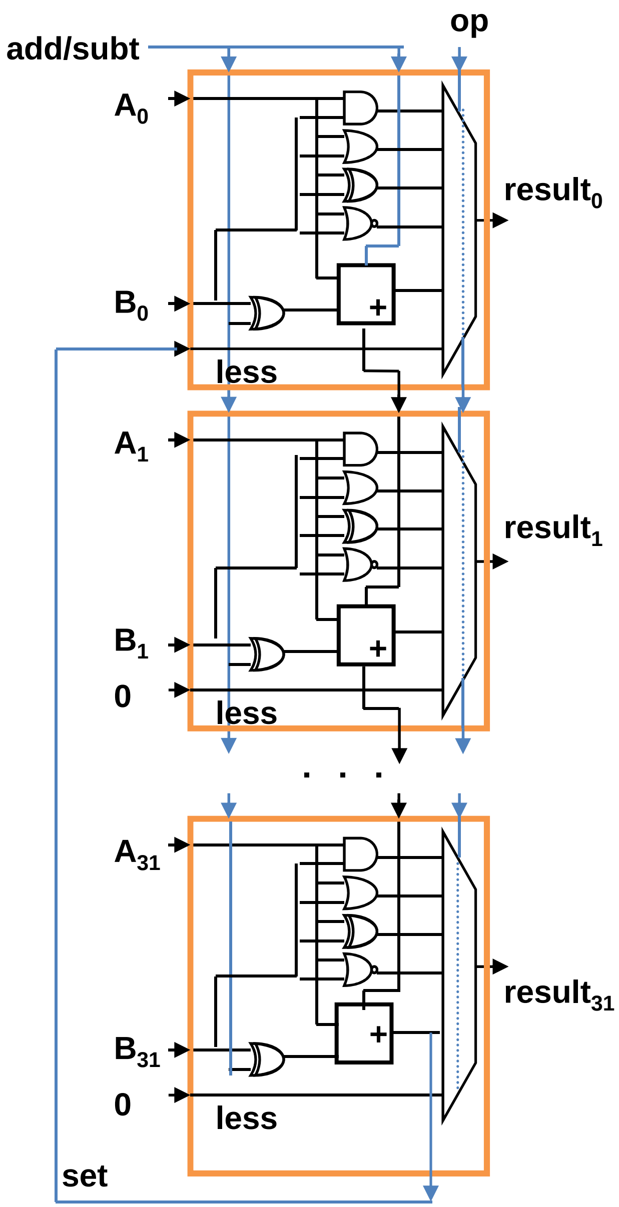
- ❑Need to support the set-on-less-than instruction (`slt`)
  - `slt` is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a - b) < 0 implies a < b
- ❑Need to support test for equality (`beq`)
  - use subtraction: (a - b) = 0 implies a = b
- ❑Need to add the overflow detection hardware

Modifying the ALU Cell for slt



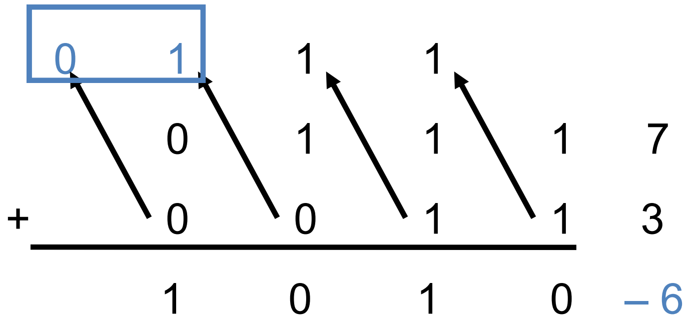
A MIPS ALU Implementation (supporting SLT)

- First perform a subtraction
- Make the result 1 if the subtraction yields a negative result
- Make the result 0 if the subtraction yields a positive result

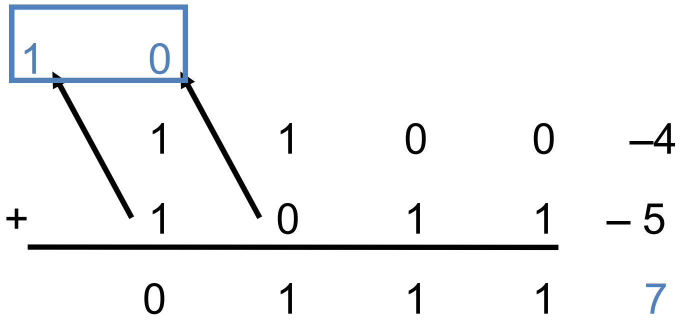


Overflow Detection

- Overflow occurs when the result is too large to represent in the number of bits allocated
  - adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive gives a negative
  - or, subtract a positive from a negative gives a positive
- On your own: Prove you can **detect overflow** by:
  - **Carry into MSB xor Carry out of MSB**

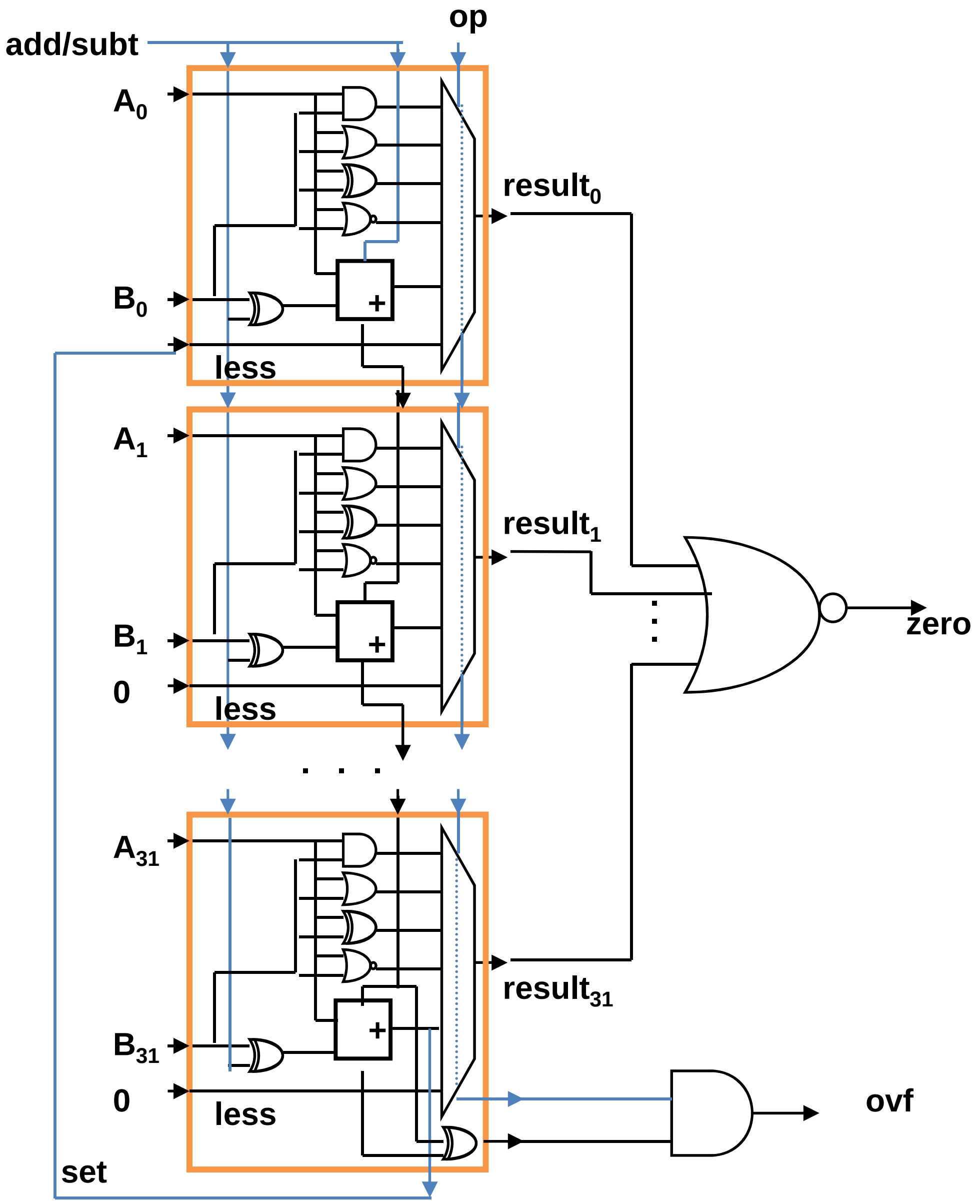


```
0   1   1   1
    0   1   1   1    7
+   0   0   1   1    3
    1   0   1   0   – 6
```



```
1   0
    1   1   0   0   –4
+   1   0   1   1   – 5
    0   1   1   1    7
```

A MIPS ALU Implementation

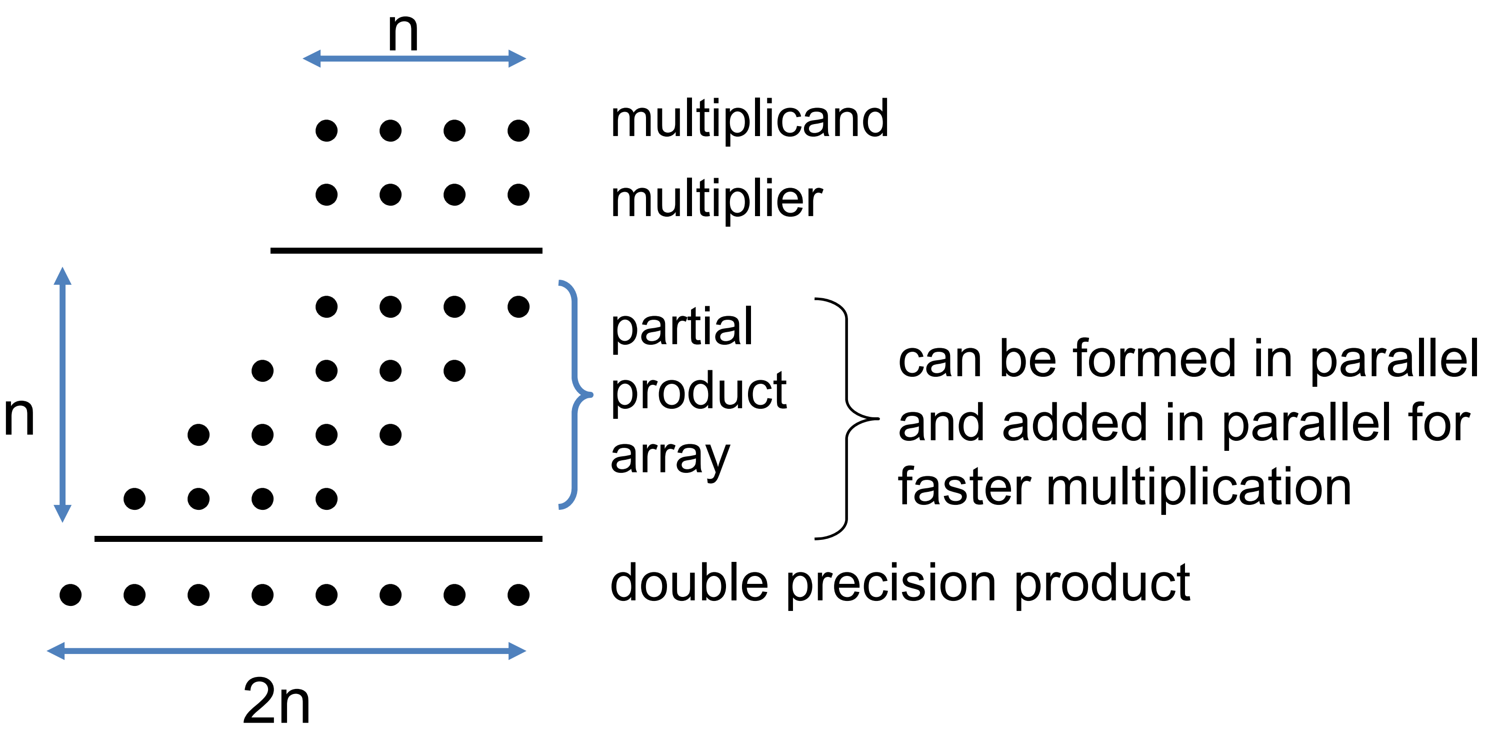


- Zero detect (`slt, slti, sltiu, sltu, beq, bne`)

- Enable overflow bit setting for signed arithmetic (`add, addi, sub`)

Multiply

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


n
multiplicand
multiplier
n
partial product array
can be formed in parallel and added in parallel for faster multiplication
double precision product
2n

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 Multiply Instruction

```
mult    $s2, $s3  # hi||lo = $s2 * $s3
```

op	rs	rt	rd	shamt	funct
000000	$s2	$s3	00000	00000	24

- Low-order word of the product is placed in *processor dedicated* register **`lo`** and the high-order word is placed in processor register **`hi`**
- **`mult`** uses signed integers and result is a signed 64-bit number. Overflow is checked in software
- MIPS uses **`multu`** for unsigned products

000000	$s2	$s3	00000	00000	25

Multiply Instruction

- The product needs to be moved to general purpose registers to become available for other operations. Instructions **mfhi** $s0 and **mflo** $s5 are provided for this.
- **mfhi** $s0

000000	00000	00000	$s0	00000	16

- **mflo** $s5

000000	00000	00000	$s5	00000	18

- Multiplication is more complicated than addition - via shifting and addition

$$\begin{array}{r l}
0010_{ten} & \text{(multiplicand)}\\
\text{x}\ \underline{1011}_{ten} & \text{(multiplier)}\\
0010\phantom{_{ten}} & \\
0010\phantom{0_{ten}} & \text{(partial product}\\
0000\phantom{00_{ten}} & \text{array)}\\
\underline{0010\phantom{000_{ten}}} & \\
00010110_{ten} & \text{(product)}
\end{array}$$

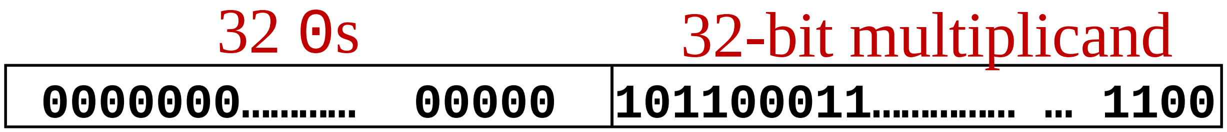
- m bits × n bits = m+n bit product 32+32=64 bits double precision product produced – more time to compute

Multiply Instruction

- Binary numbers make it easy:

  0 => place 0s (0x multiplicand) in the proper place

  1 => place a copy of multiplicand in the proper place



32 0s

32-bit multiplicand

0000000………. 00000	101100011…………. … 1100

- At each stage shift the multiplicand left (x 2)
- Use next LSB of b to determine whether to add in the shifted multiplicand
- Accumulate 2n bit partial product at each stage
- The process is repeated 32 times in MIPS

Multiplication

- Start with long-multiplication approach

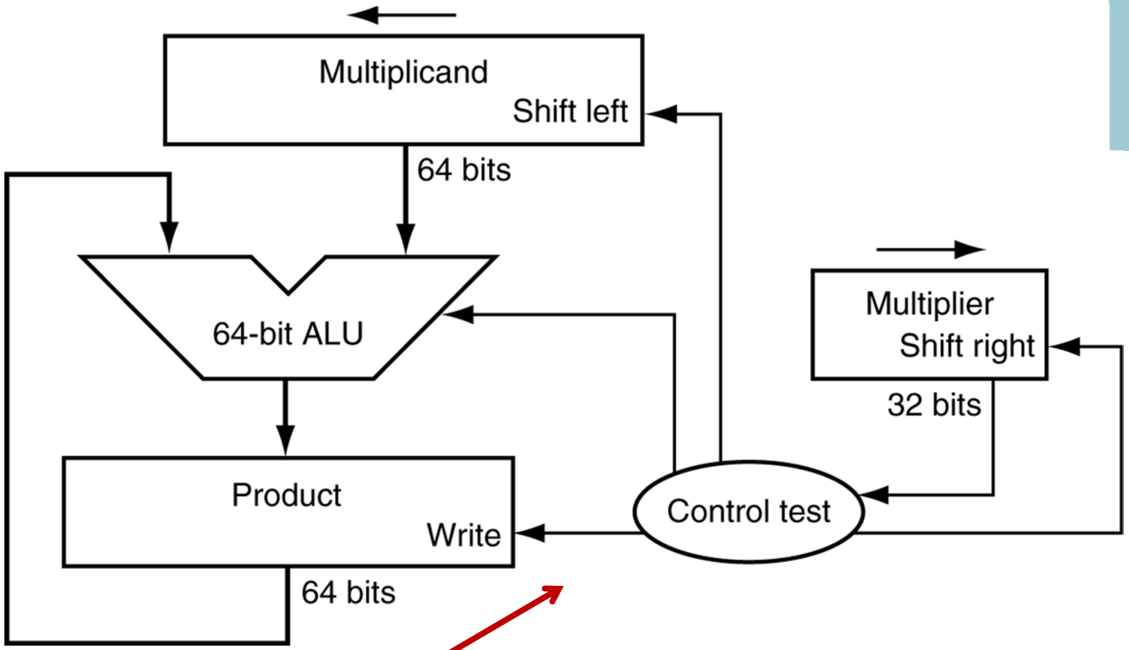
32 0s	32-bit multiplicand

multiplicand

multiplier

```
      1000
  ×   1001
  --------
      1000
     0000
    0000
   1000
  --------
   1001000
```

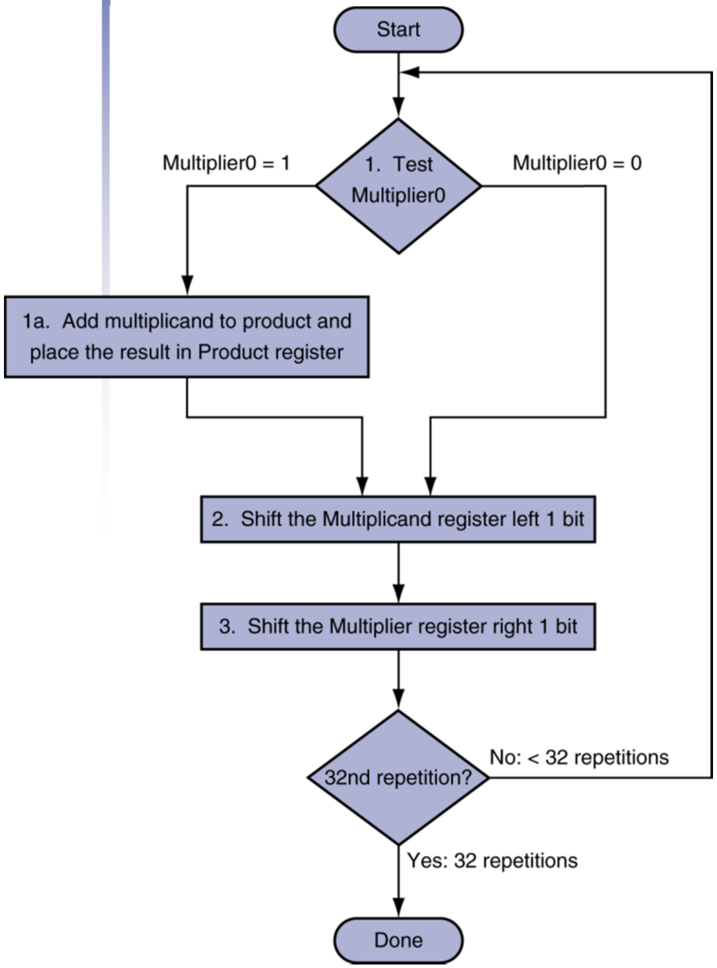
product

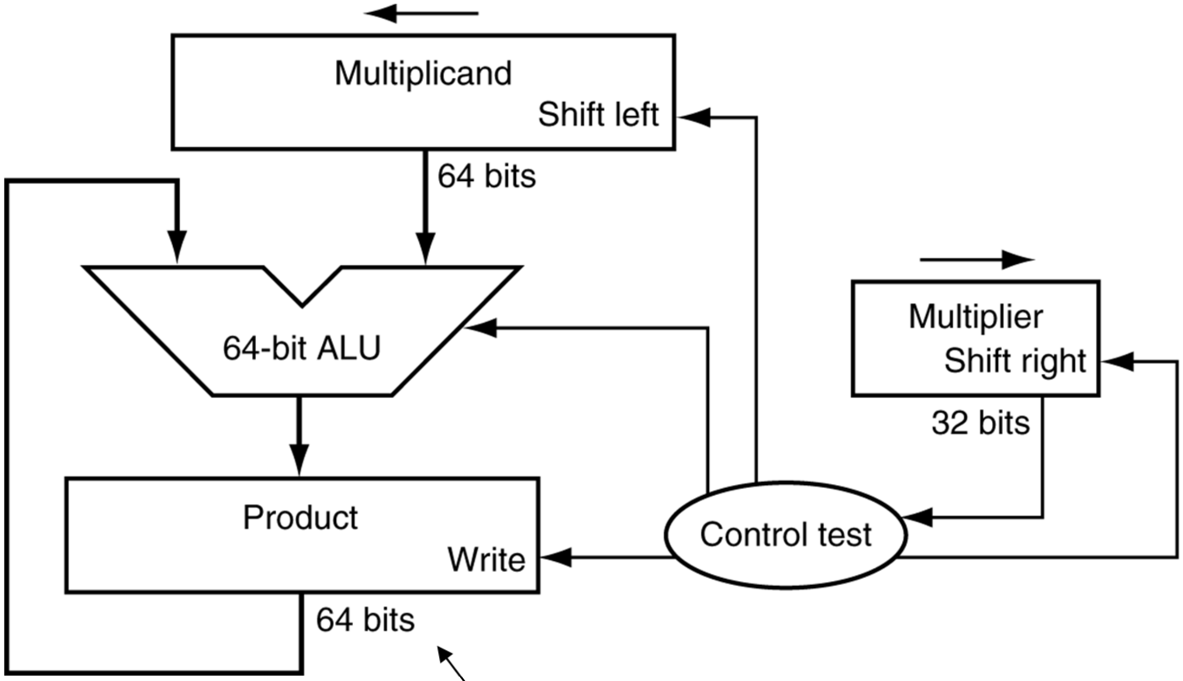


Length of product is the sum of operand lengths

If Multiplier0 = 1 add multiplicand to product register

Multiplication Hardware

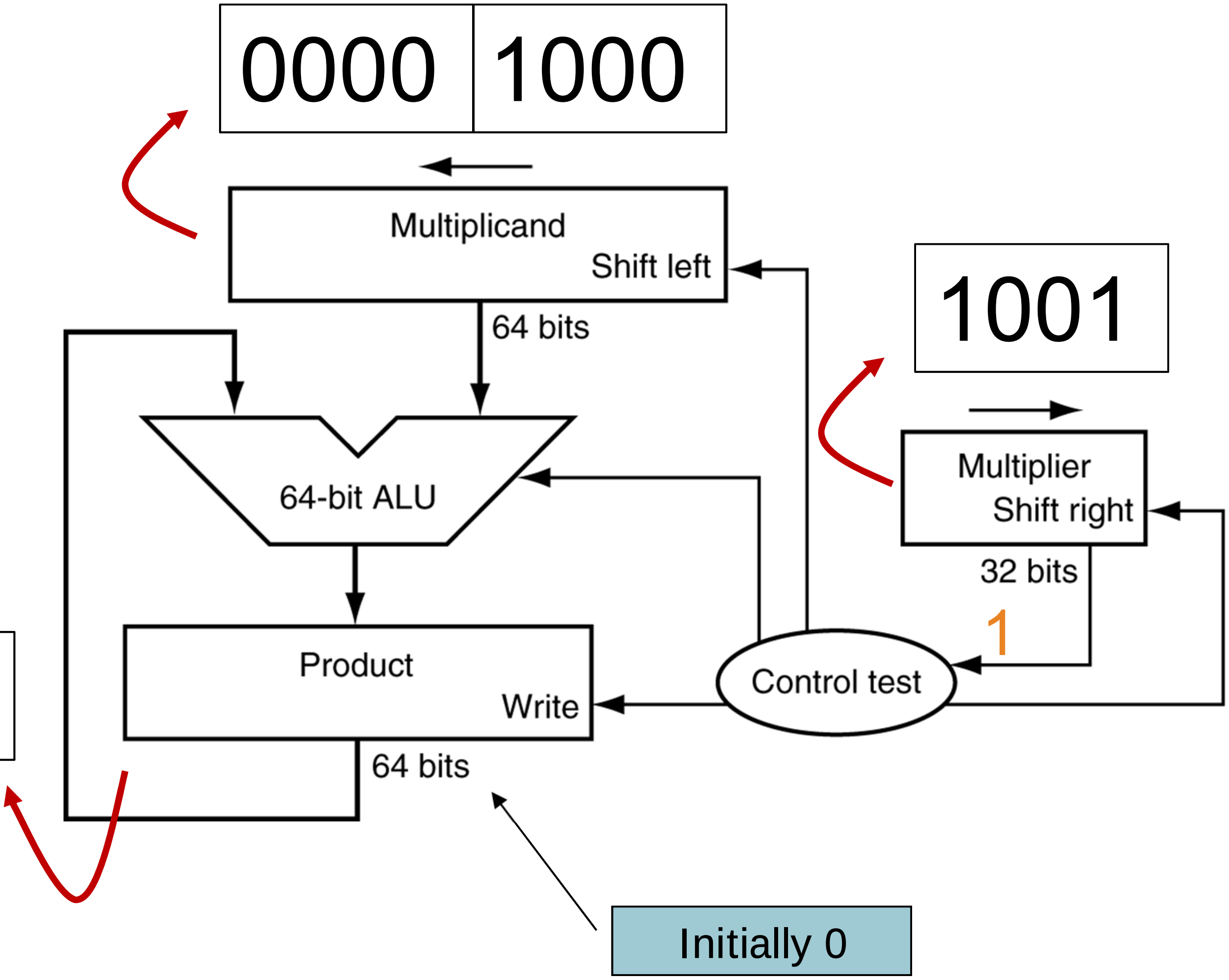

Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done



Initially 0

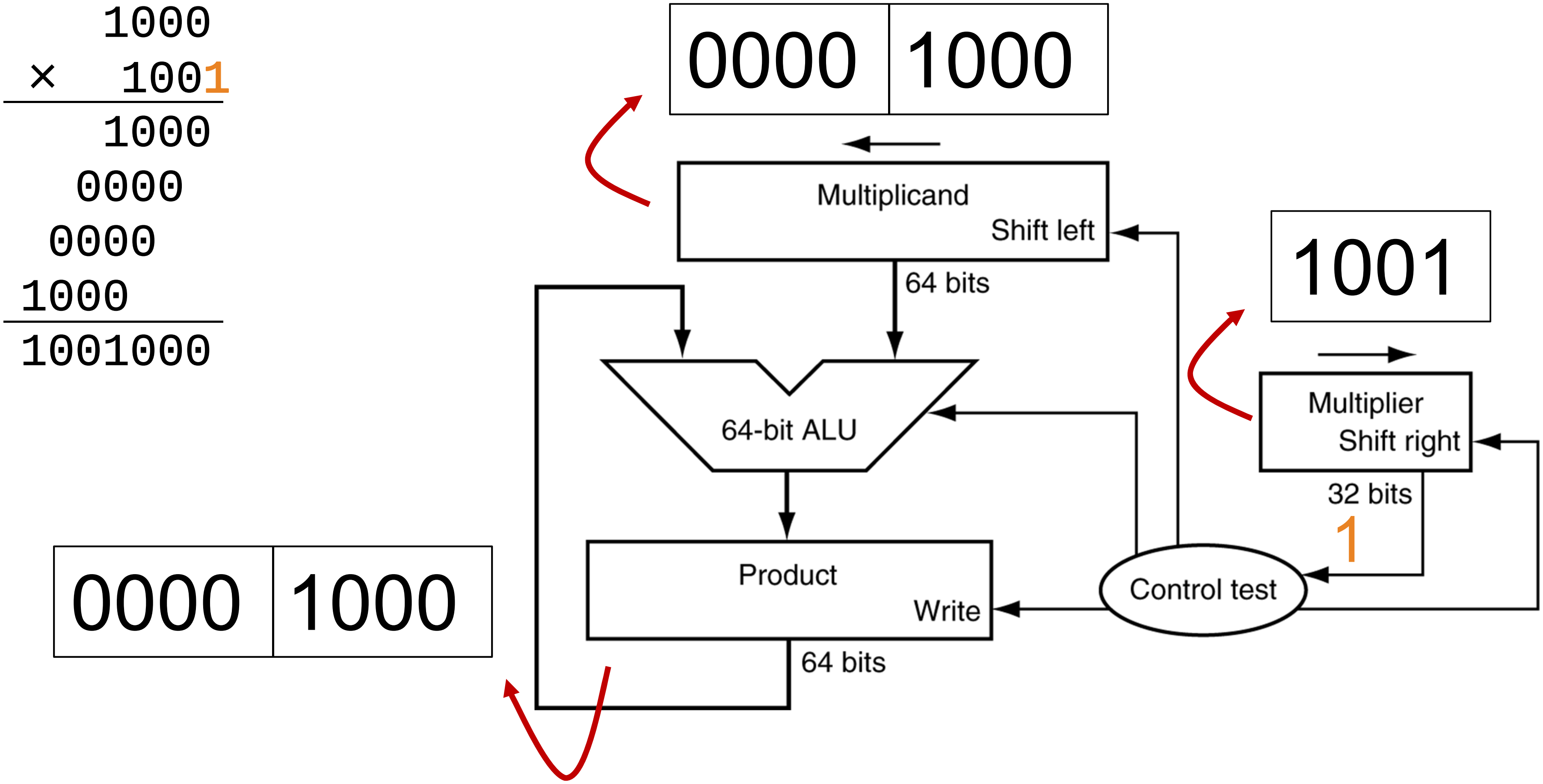
Multiplication Hardware

```
    1000
 ×   1001
    1000
   0000
  0000
 1000
 1001000
```



Multiplication Hardware

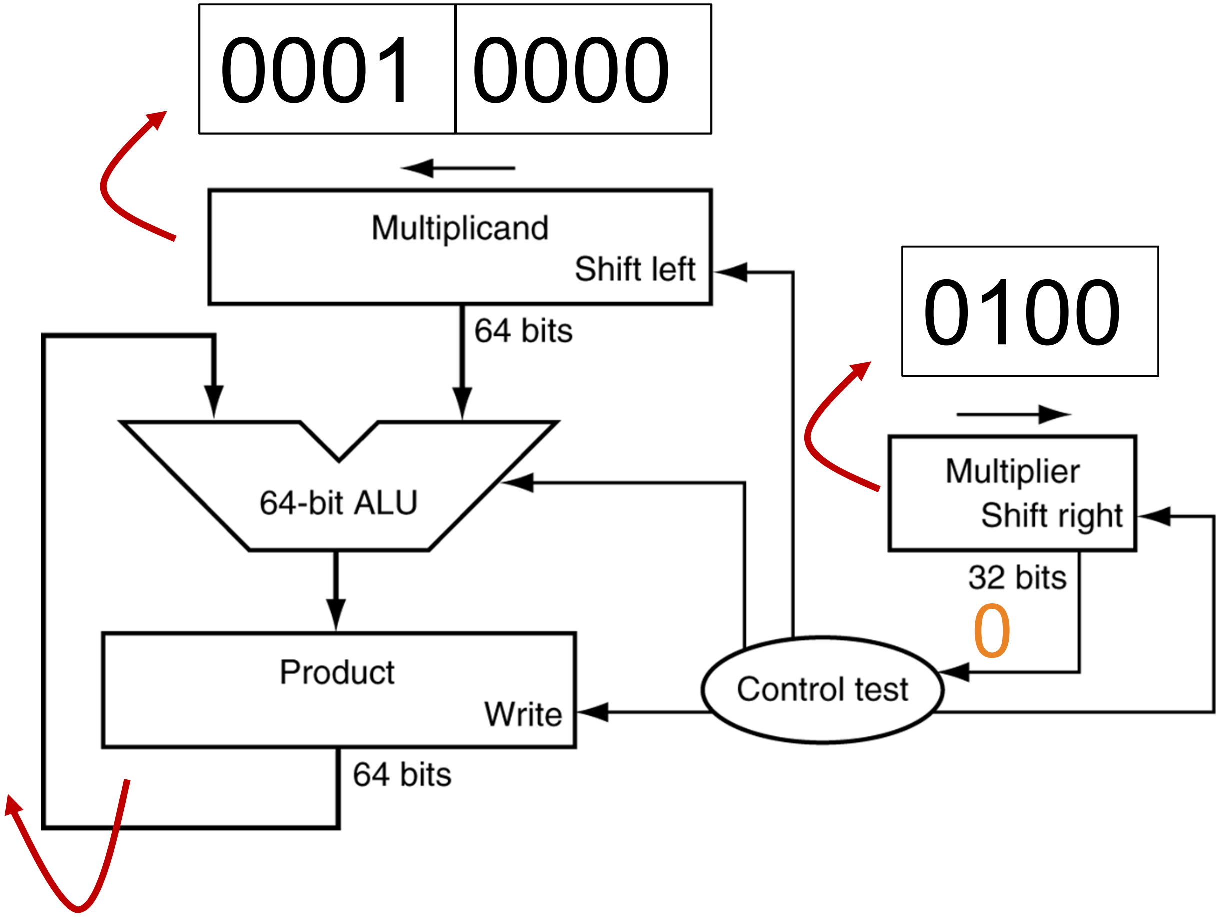
```
    1000
×   1001
-------
    1000
   0000
  0000
 1000
-------
 1001000
```



Multiplication Hardware

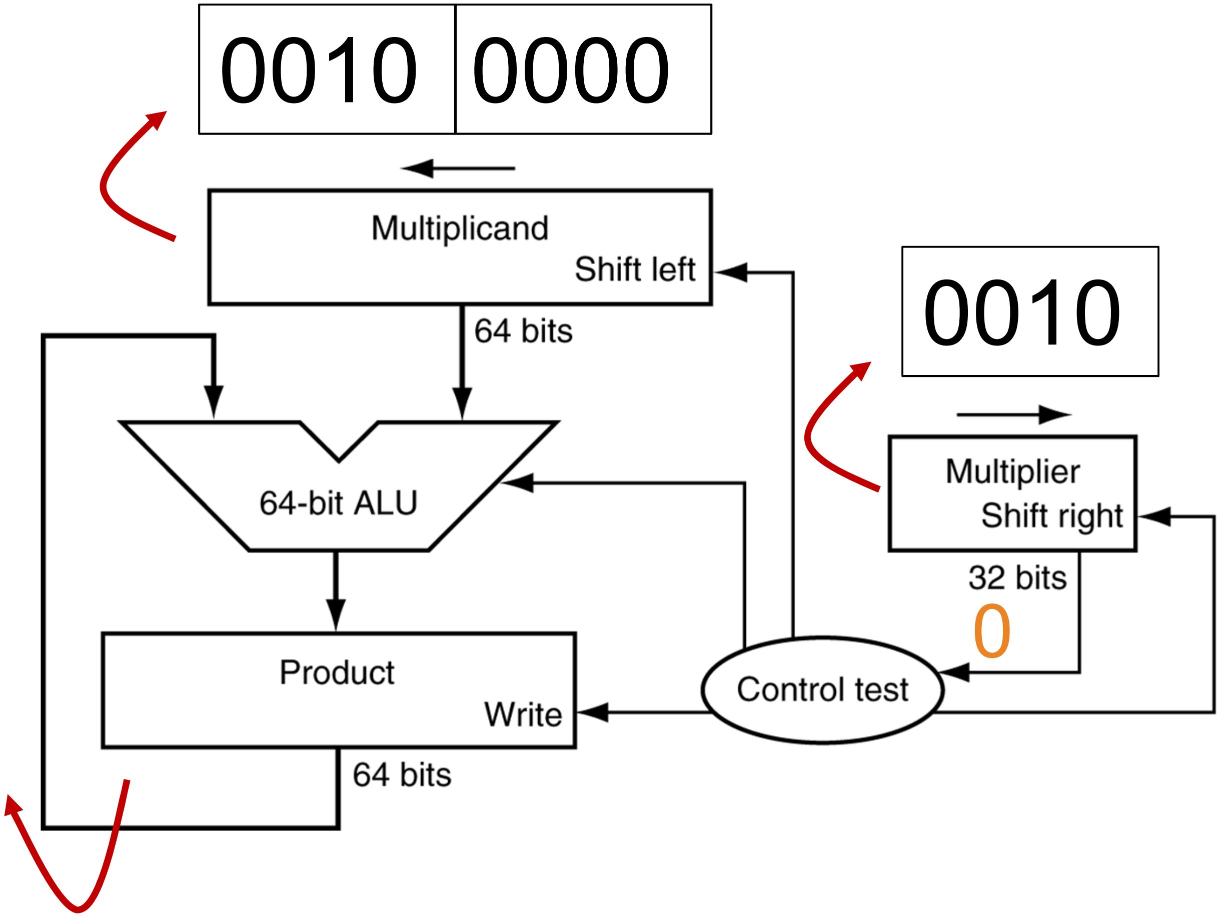
```
     1000
 ×    1001
     1000
    0000
   0000
  1000
  1001000
```

0000 | 1000



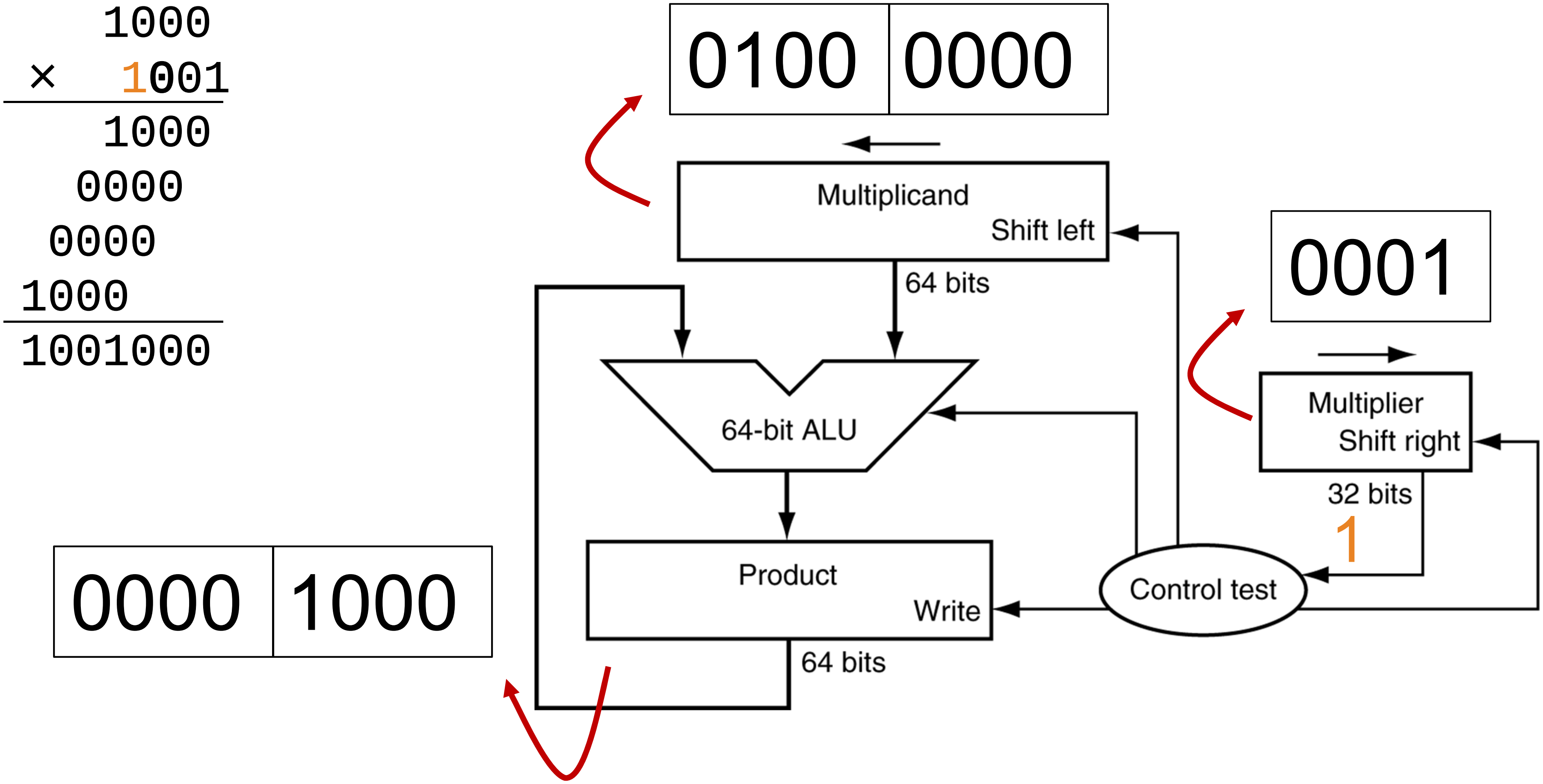
Multiplication Hardware

```
     1000
×     1001
     1000
    0000
   0000
  1000
  1001000
```



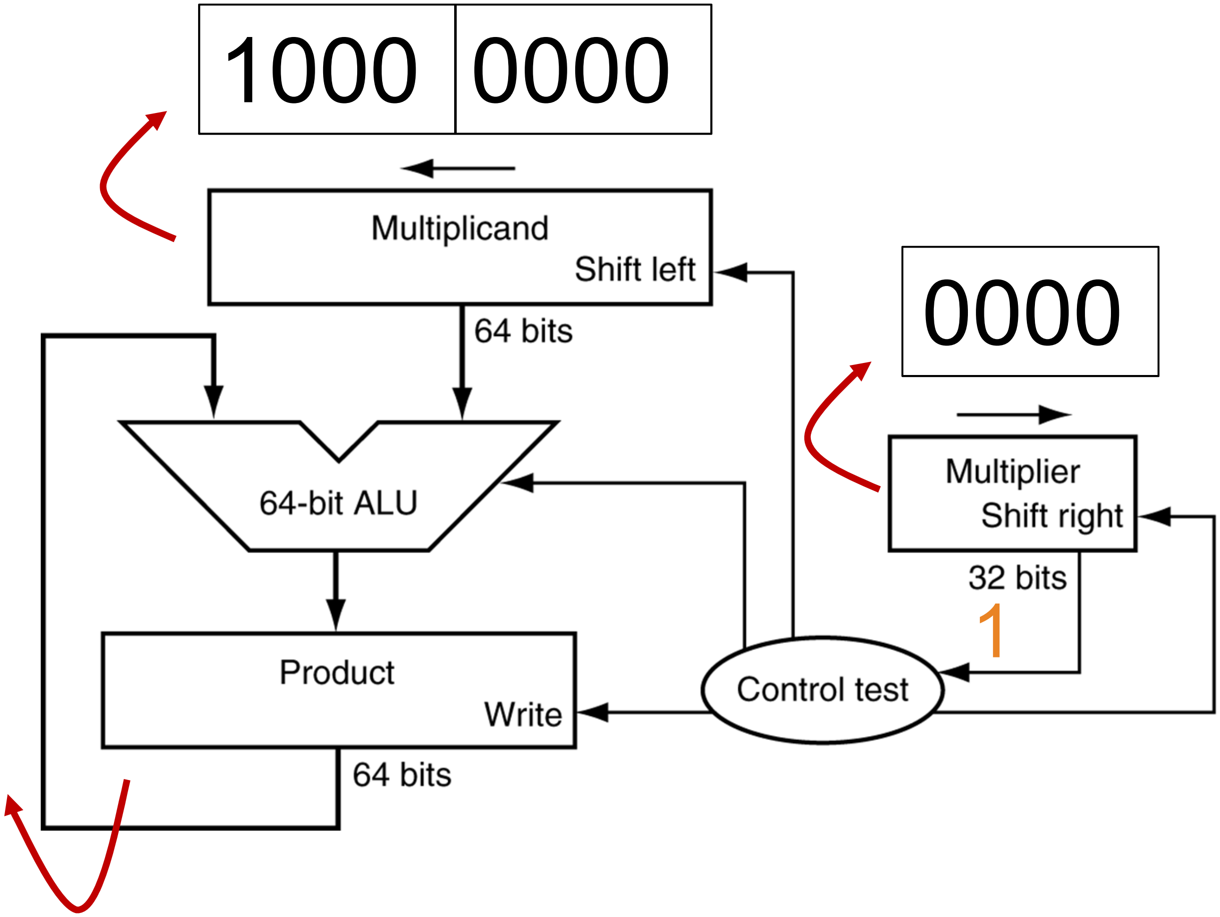
Multiplication Hardware

```
   1000
×   1001
   1000
  0000
 0000
1000
1001000
```



Multiplication Hardware

```
    1000
×    1001
    1000
   0000
  0000
 1000
 1001000
```

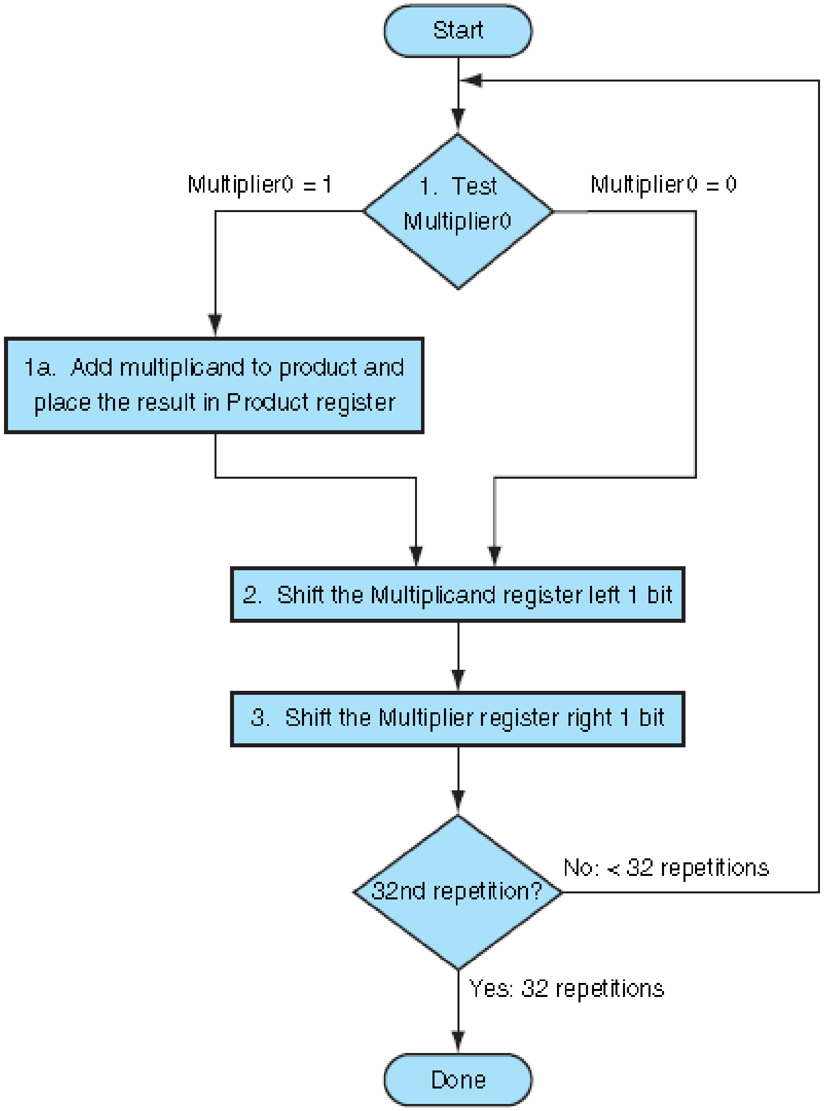


0100 | 1000

Final Result

```
   01000000
+  00001000
   01001000
```

Multiply Algorithm Version 1


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

1001_{two} x 1001_{two}

•	Multiplier	Multiplicand	Product
0	1001	00001001	00000000
1	1001	00001001	00000000
	1001	00010010	00001001
	0100	00010010	00001001
2	0100	00010010	00001001
	0010	00100100	00001001
3	0010	00100100	00001001
	0001	01001000	00001001
4	0001	01001000	00001001
	0000	10010000	01010001

Observations on Multiply Version 1

- 1 clock cycle per step => 100 clocks per multiply

  Ratio of multiply to add 1:5 to 1:100

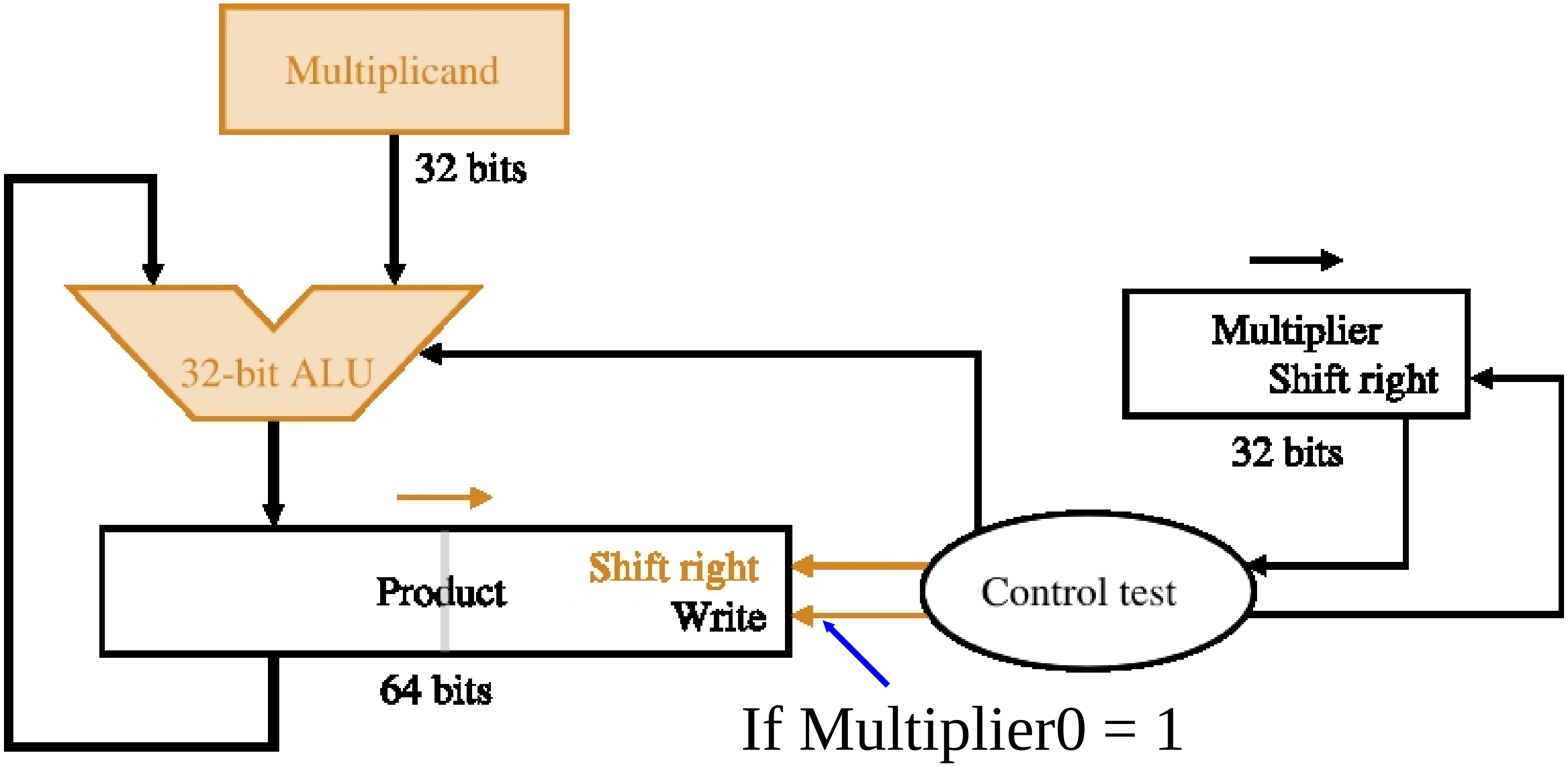
- 1/2 bits in multiplicand always 0
  => 64-bit adder is wasted

- 0’s inserted in right of multiplicand as it is shifted left
  => least significant bits of product never changed once formed

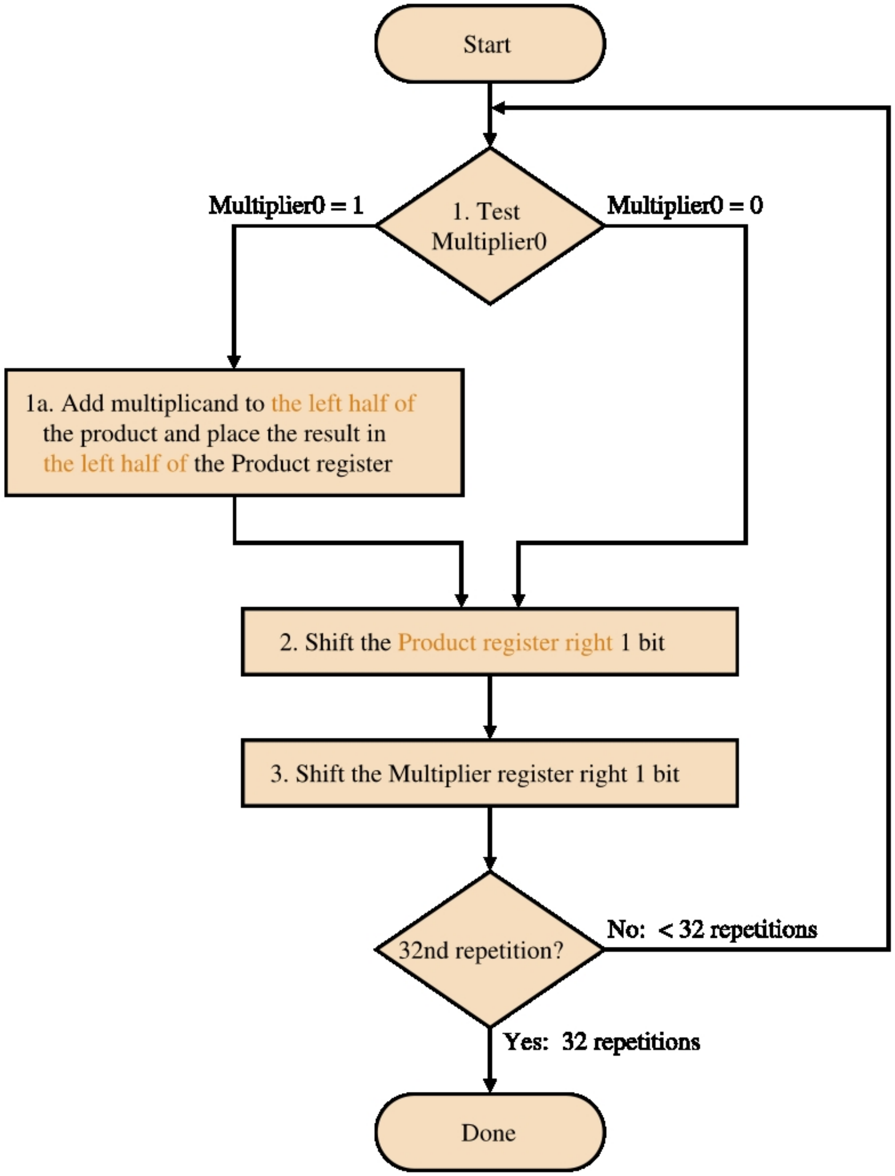
- Instead of shifting multiplicand to left, shift product to right?

Multiply Hardware Version 2

- 32-bit Multiplicand register, 32 -bit ALU, 64-bit Product register, 32-bit Multiplier register

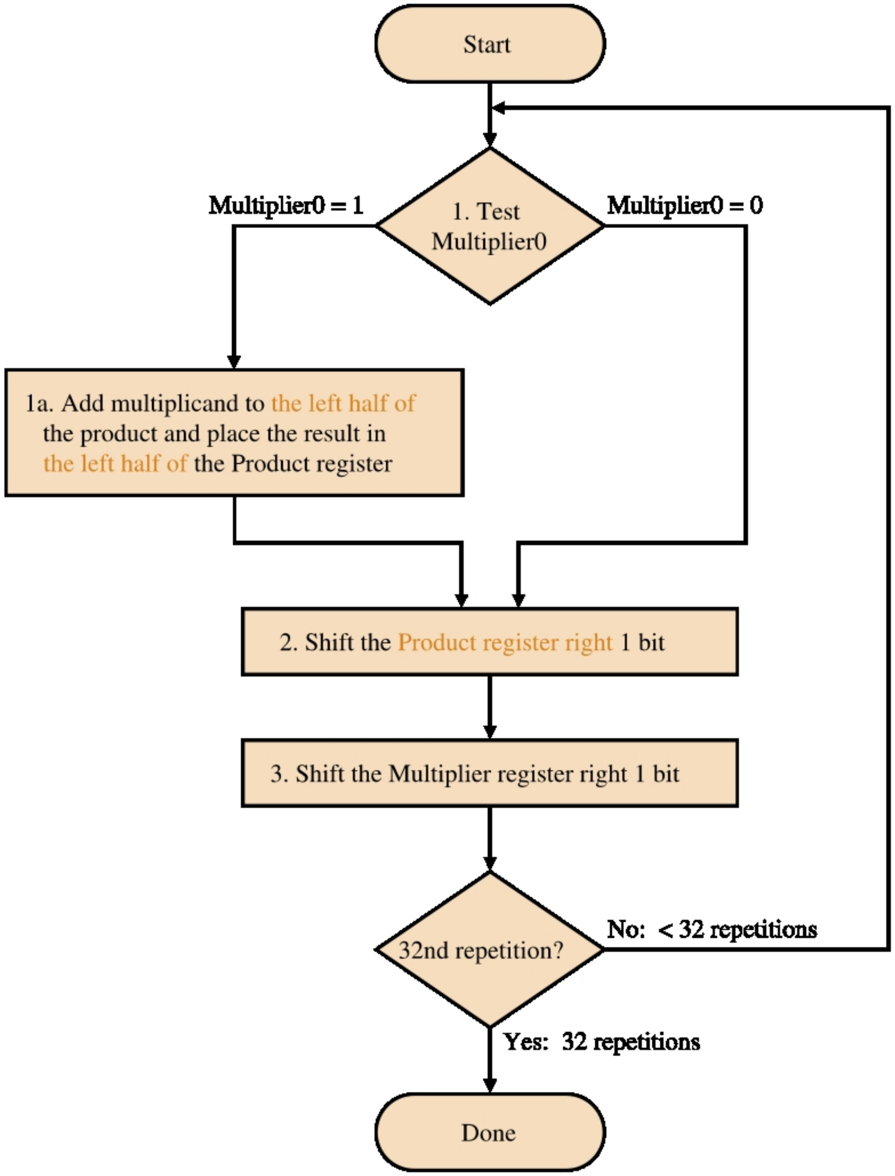


Multiply Algorithm Version 2


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

- Multiplicand stays still and product moves right
- Product register wastes space that exactly matches size of multiplier
- So we can combine Multiplier register and Product register

Multiply Algorithm Version 2


Start

1. Test Multiplier0

Multiplier0 = 1

Multiplier0 = 0

1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register

2. Shift the Product register right 1 bit

3. Shift the Multiplier register right 1 bit

32nd repetition?

No: < 32 repetitions

Yes: 32 repetitions

Done

1001_{two} x 1001_{two}

•	Multiplier	Multiplicand	Product
0	1001	1001	00000000
1	1001	1001	10010000
	1001	1001	01001000
	0100	1001	01001000
2	0100	1001	01001000
	0100	1001	00100100
	0010	1001	00100100
3	0010	1001	00100100
	0010	1001	00010010
	0001	1001	00010010
4	0001	1001	10100010
	0000	1001	01010001

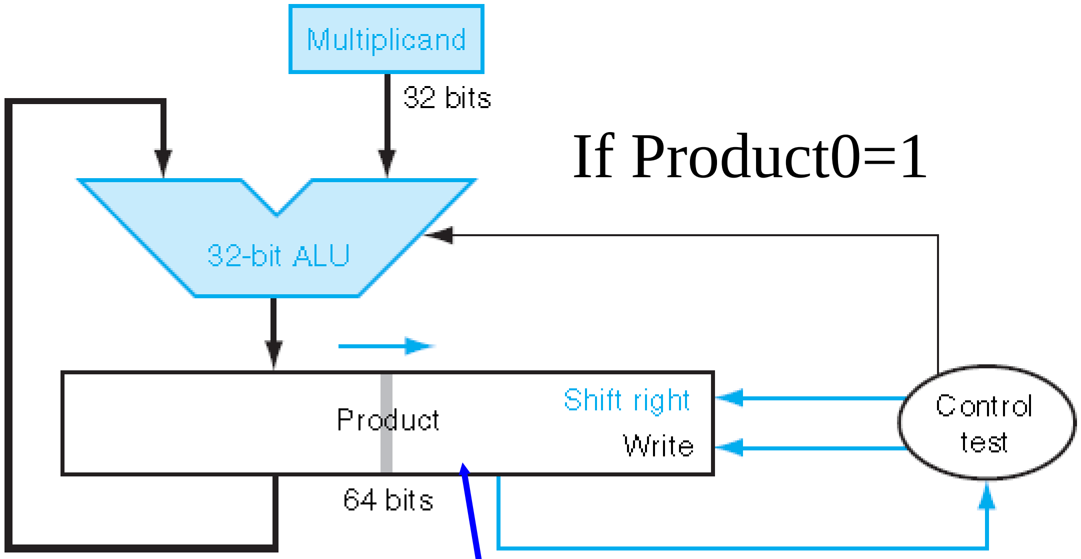
Multiply Hardware Version 3

- 32-bit Multiplicand register, 32 -bit ALU, 64-bit Product register, (0-bit Multiplier register)

- 2 steps per bit because Multiplier & Product combined

- MIPS registers **`Hi`** and **`Lo`** are left and right halves of the Product

- Gives us MIPS instruction MultU



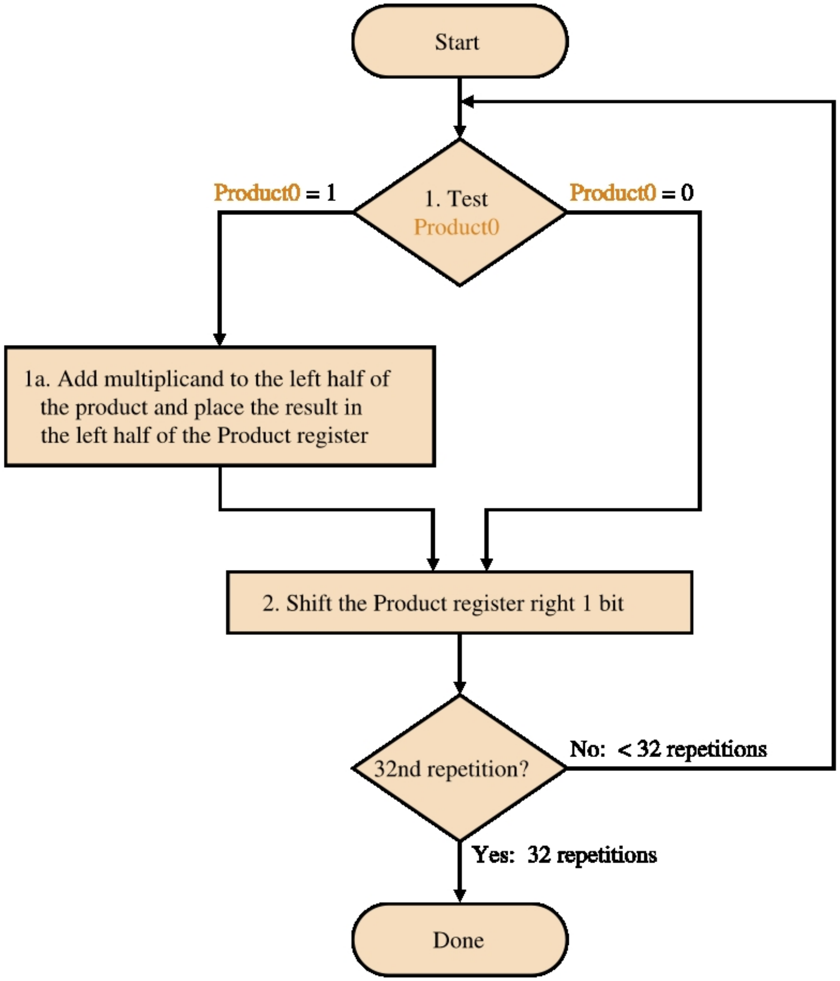
If Product0=1

Initially 32 0s

32-bit multiplier

`0000000………. 00000`	`101100011…………. … 1100`

Multiply Algorithm Version 3


Start
Product0 = 1
1. Test Product0
Product0 = 0
1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register
2. Shift the Product register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

1001_{two} x 1001_{two}

Iter.	Multiplicand	Product
0	1001	0000 1001
1	1001	0000 1001
add	1001	1001 1001
shift	1001	0100 1100
2	1001	0100 1100
shift	1001	0010 0110
3	1001	0010 0110
shift	1001	0001 0011
4	1001	0001 0011
add	1001	1010 0011
shift	1001	0101 0001

Multiplication of signed integers

- What about signed multiplication?
- Easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps)
- Apply definition of 2's complement. Need to sign-extend partial products and subtract at the end
- *Booth's Algorithm* is elegant way to multiply signed numbers using same hardware as before and save cycles
- It can handle multiple bits at a time, thus it is faster

Motivation for Booth’ s Algorithm

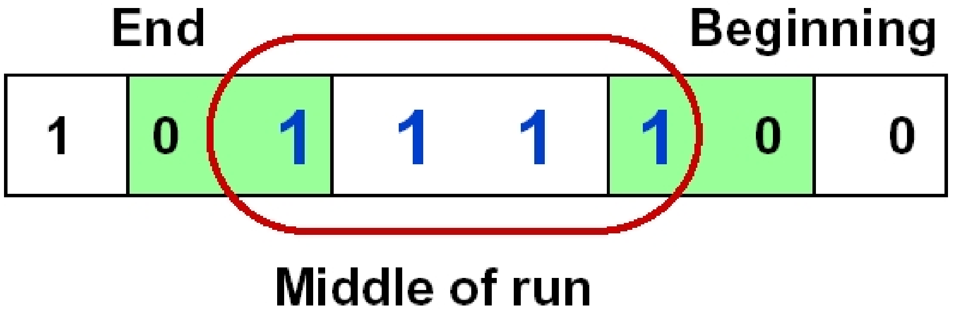
- Example 2 x 6 = 0010 x 0110$_{two}$:

```
        0010
 x      0110
  -----------
+       0000      shift (0 in multiplier)
 +     0010       add (1 in multiplier)
 +    0010        add (1 in multiplier)
 +   0000         shift  (0 in multiplier)
  -----------
     00001100
```

- ALU with add or subtract gets same result in more than one way:

```
6    = – 2 + 8
0110 = – 00010 + 01000 = 11110 + 01000
```

Booth’ s Algorithm



Now the test in the algorithm depends on *two* bits. Results are placed in the left half of the product register.

Current Bit	Bit to the Right	Explanation	Example	Op
1	0	Begins run of 1s	`000111`**`10`**`00`	subtract
1	1	Middle of run of 1s	`00011`**`11`**`000`	no op
0	1	End of run of 1s	`00`**`01`**`111000`	add
0	0	Middle of run of 0s	`0`**`00`**`1111000`	no op

Originally for Speed (when shift was faster than add)

- Replace a string of `1`s in multiplier with an initial subtract when we first see a `10` and then later add for the first `01`

Booths Example (2 x 7)

mythical bit

Operation	Multiplicand	Product register	next operation?
0. initial value	0010	0000 0111 0	10 -> subtract
1a. P = P - m	1110	+	1110
1110 0111 0		shift P (sign extend)	
1b.	0010	1111 0011 1	11 -> nop, shift
2.	0010	1111 1001 1	11 -> nop, shift
3.	0010	1111 1100 1	01 -> add
4a.	0010	+0010	
		0001 1100 1	shift
4b.	0010	0000 1110 0	done

Booths Example (2 x -3) (1111 1010$_{\text{two}}$)

Operation	Multiplicand	Product	next?
0. initial value	**0010**	**0000 1101 0**	**10 -> subtract**
1a. P = P - m	1110	+	1110
1110 1101 0	shift P (sign ext)		
1b.	0010	1111 0110 1	01 -> add multiplicand
		+ 0010	
2a.		0001 0110 1	shift P and sign ext.
2b.	0010	0000 1011 0 +	10 -> sub multiplicand 1110
3a.	0010	1110 1011 0	shift and sign ext
3b.	0010	1111 0101 1	11 -> no op
4a		1111 0101 1	shift
4b.	0010	1111 1010 1	done

Division: Paper & Pencil

```
                  1001ten   Quotient
                ___________
Divisor1000ten |1001110ten  Dividend
               -1000
                   11
                   111
                   1110
                  -1000
                    110   Remainder (or Modulo result)
```

A number can be subtracted, creating quotient bit on each step

Dividend = Quotient x Divisor + Remainder

- We assume for now unsigned 32-bit integers (dividend, divisor, quotient and remainder are all 32 bit integers
- Different versions of divide algorithm, successive refinement

MIPS Divide Instruction

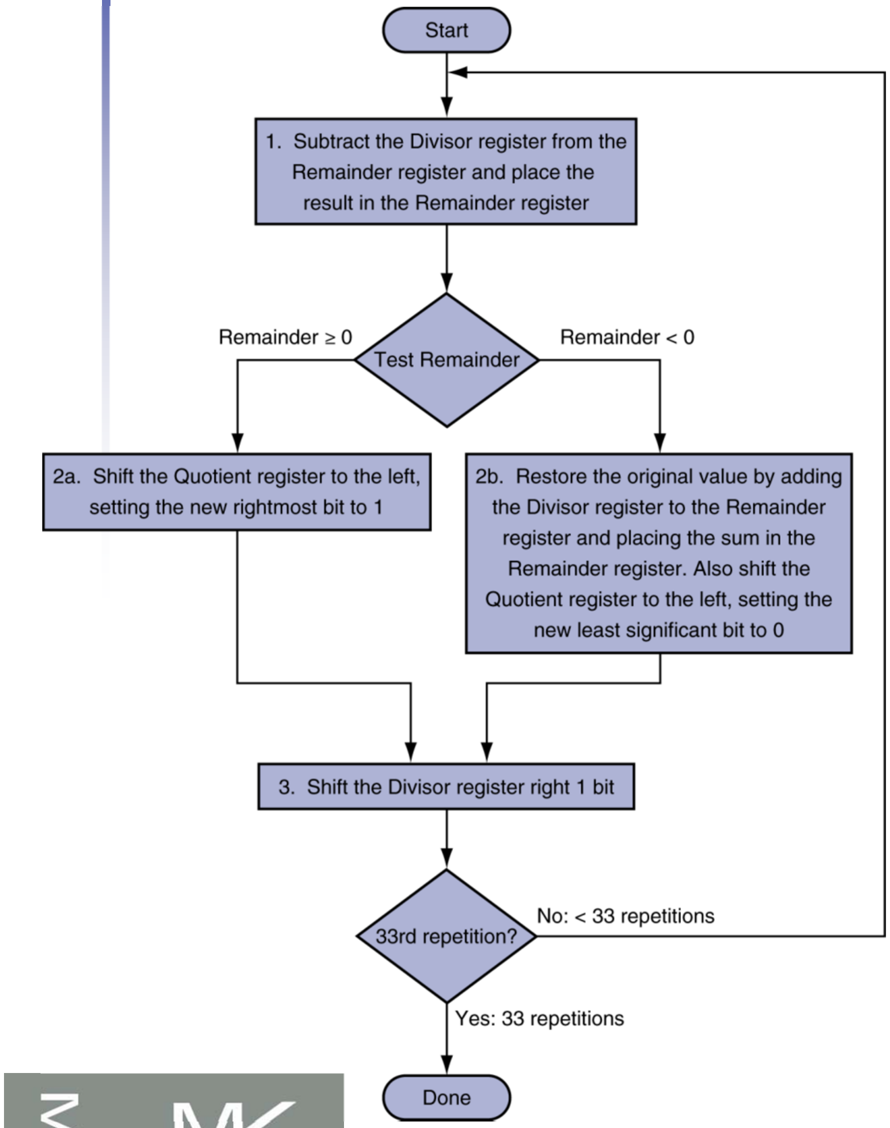
- **div** $s2, $s3

op	rs	rt	rd	shamt	funct
000000	$s2	$s3	00000	00000	26

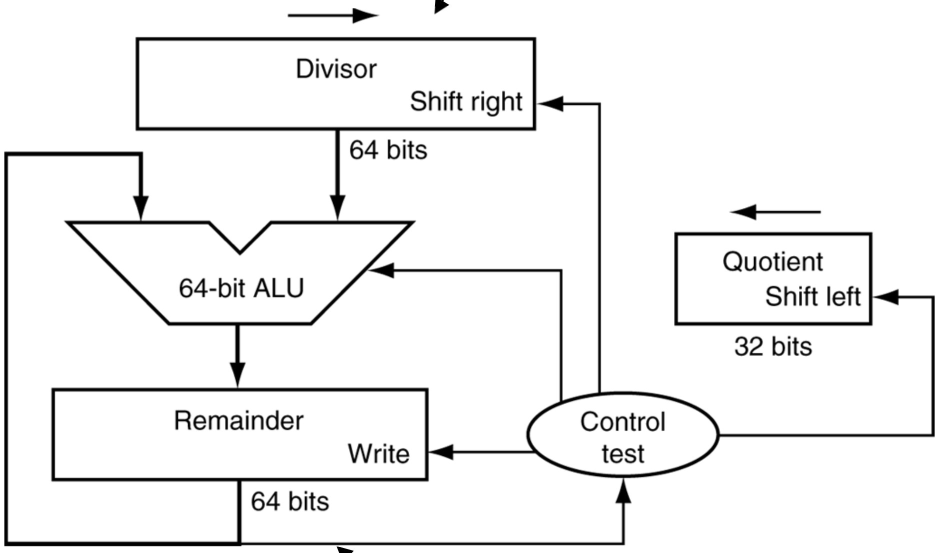
  - The division **quotient** is placed in processor dedicated register **lo** and the **remainder** is placed in processor register **hi**
  - **div** uses signed integers and result is a signed 64-bit number. Overflow and division by 0 are checked in software
  - MIPS uses **divu** for unsigned divisions

000000	$s2	$s3	00000	00000	27

Division Hardware (version 1)


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
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
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

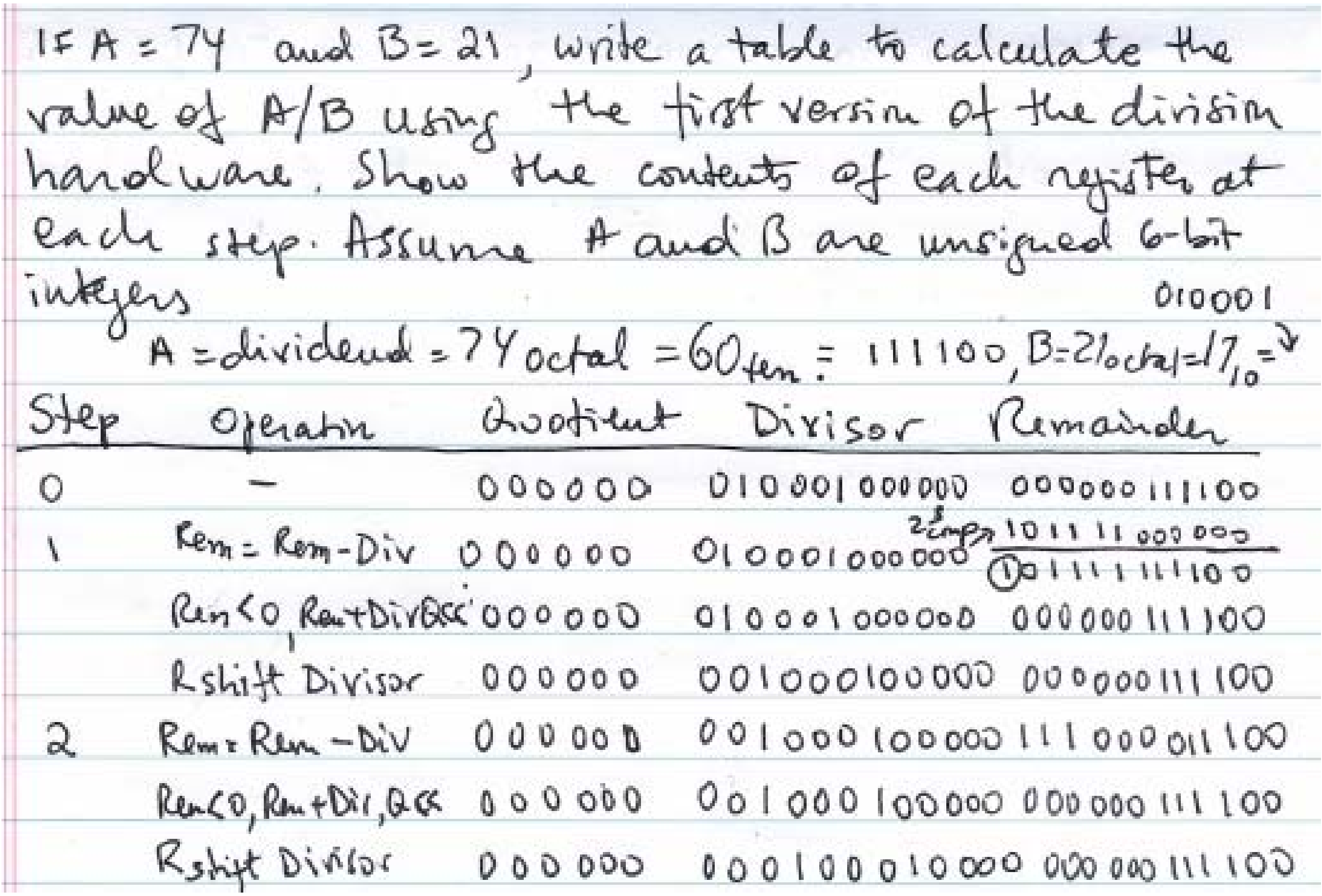
Initially divisor in left half



Initially dividend

If Reminder63 = 1, Quotient0=0
Reminder63 = 0, Quotient0=1

Exercise



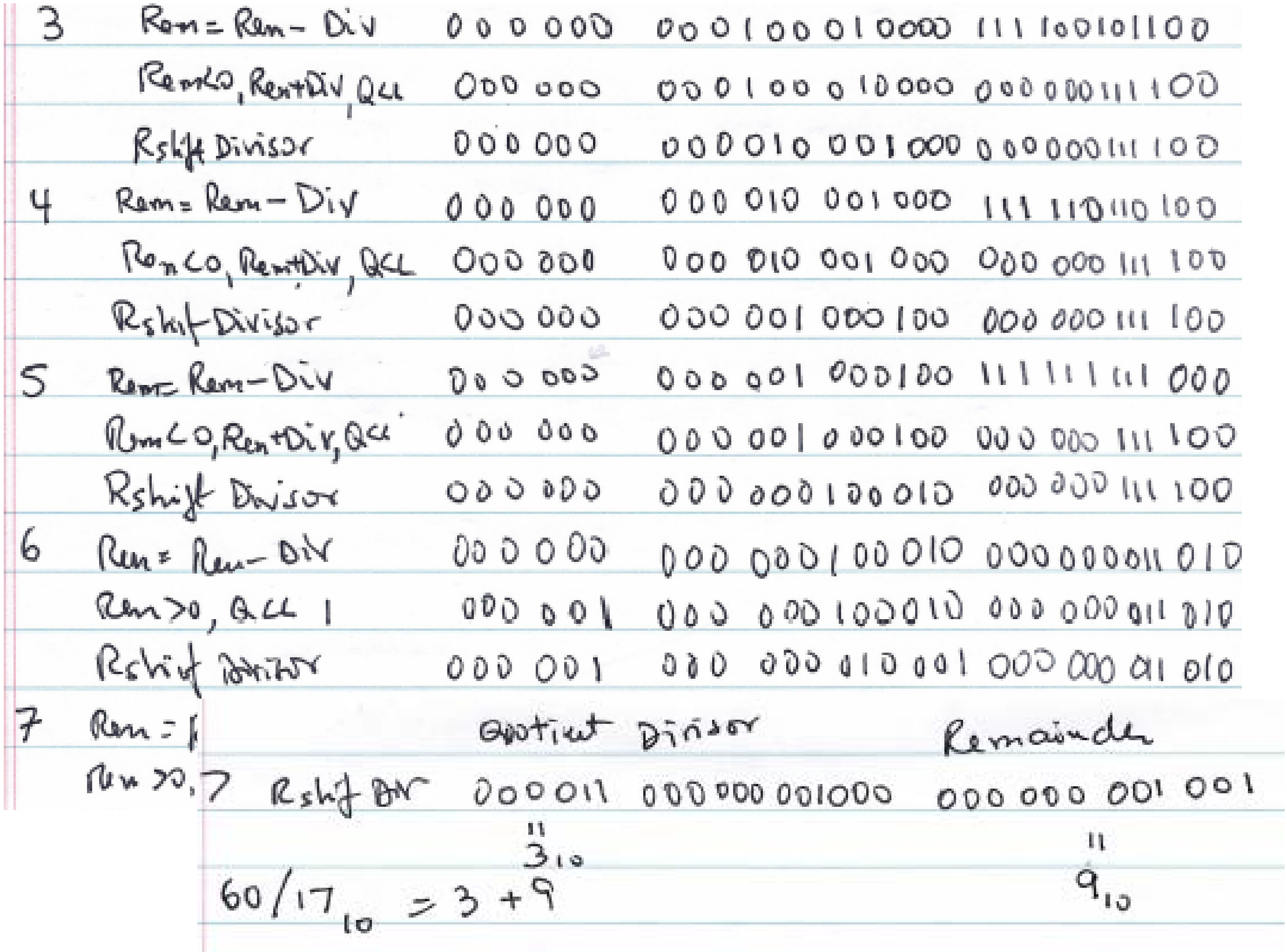
IF A = 74 and B = 21, write a table to calculate the value of A/B using the first version of the division hardware. Show the contents of each register at each step. Assume A and B are unsigned 6-bit integers

010001

A = dividend = 74 octal = 60$_{ten}$ = 111100, B = 21$_{octal}$ = 17$_{10}$ = ↲

Step	Operation	Quotient	Divisor	Remainder
0	–	000000	010001000000	000000111100
1	Rem = Rem-Div	000000	010001000000	2's comp → 101111000000 ①01111111100
	Rem<0, Rem+Div, Q<<	000000	010001000000	000000111100
	Rshift Divisor	000000	001000100000	000000111100
2	Rem = Rem – Div	000000	001000100000	111000011100
	Rem<0, Rem+Div, Q<<	000000	001000100000	000000111100
	Rshift Divisor	000000	000100010000	000000111100

Exercise (Cont’d)



		Quotient	Divisor	Remainder
3	Rem = Rem − Div	000 000	000100 010000	111 10010 1100
	Rem<0, Rem+Div, Q<<	000 000	000100 010000	000 000111 100
	Rshift Divisor	000 000	000010 001000	000000111 100
4	Rem = Rem − Div	000 000	000 010 001 000	111 110110 100
	Rem<0, Rem+Div, Q<<	000 000	000 010 001 000	000 000 111 100
	Rshift Divisor	000 000	000 001 000 100	000 000 111 100
5	Rem = Rem − Div	000 000	000 001 000100	111 111 111 000
	Rem<0, Rem+Div, Q<<	000 000	000 001 000100	000 000 111 100
	Rshift Divisor	000 000	000 000100 010	000 000 111 100
6	Rem = Rem − Div	000 000	000 000100 010	000000011 010
	Rem>0, Q<< 1	000 001	000 000100010	000 000 011 010
	Rshift Divisor	000 001	000 000 010 001	000 000 011 010
7	Rem = [			
	Rem>0, 7 Rshft Div	000011	000 000 001000	000 000 001 001

$= 3_{10}$ $\qquad$ $= 9_{10}$

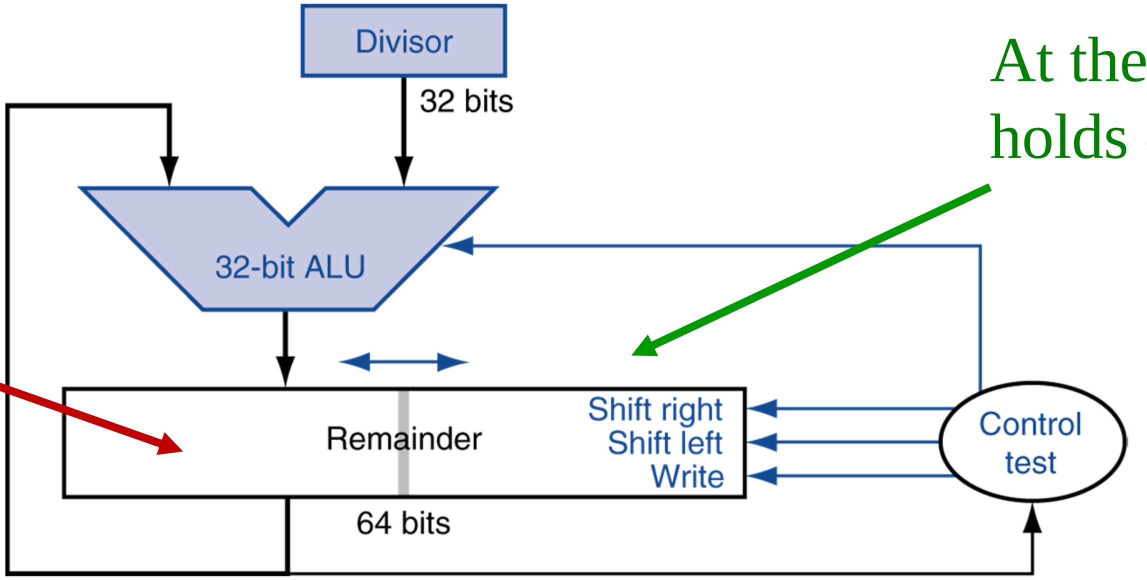
$60/17_{10} = 3 + 9$

Observations on Divide Version 1

- 1/2 bits in divisor always 0
  => 1/2 of 64-bit adder is wasted
  => 1/2 of divisor is wasted
- Instead of shifting divisor to right,
  shift the remainder to left?
- 1st step cannot produce a 1 in quotient bit (otherwise too big for the register)
  => switch order to shift first and then subtract, can save 1 iteration

Optimized Divider

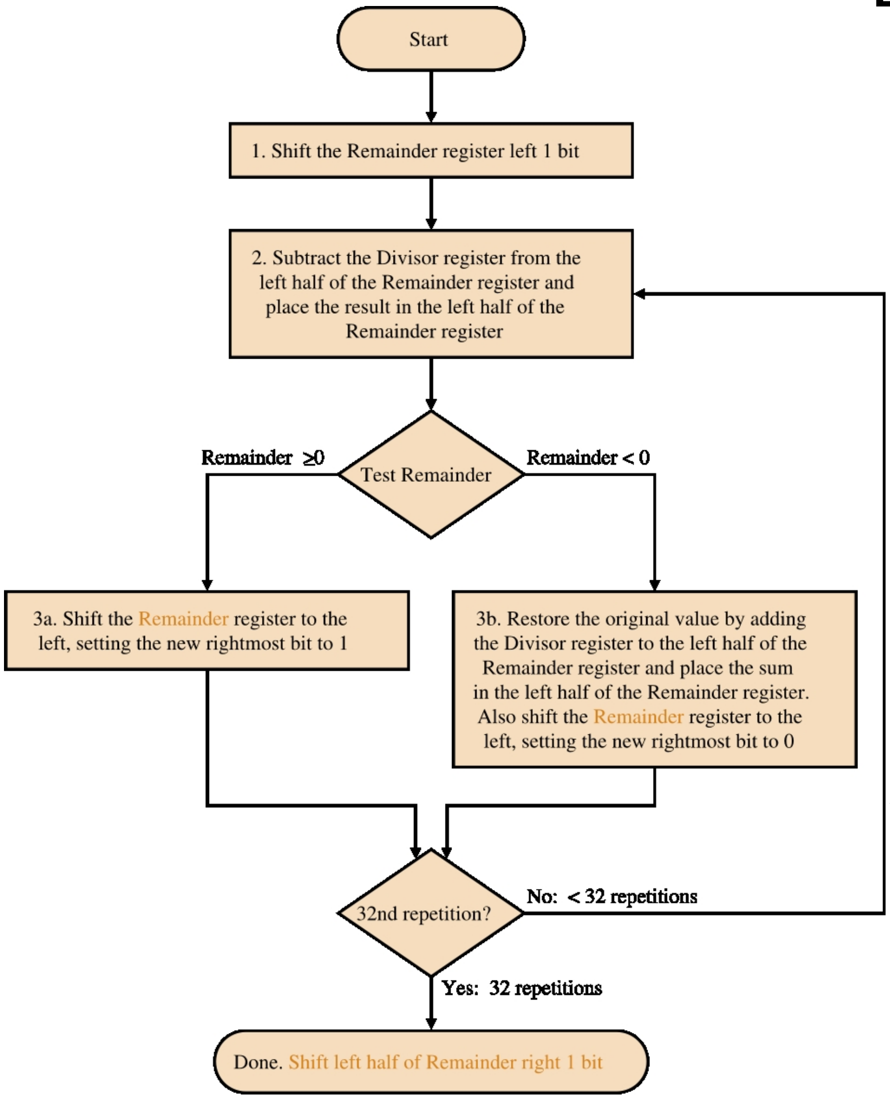
At the end remainder is here



At the end right half holds quotient

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

Divide Algorithm Version 2


Start
1. Shift the Remainder register left 1 bit
2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register
Test Remainder
Remainder ≥0
Remainder < 0
3a. Shift the Remainder register to the left, setting the new rightmost bit to 1
3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done. Shift left half of Remainder right 1 bit

Divide 0000 0111 by 0010 (2s 1110)

Iteration Divisor Remainder reg oper?

Remainder register initially holds dividend

```
0.        0010    0000 0111     in. val
0a        0010    0000 1110 shft lf 1
1a        0010    1110 1110 Rem=Rem-Div
1b        0010    0000 1110 Rem=Rem+Div
                  0001 1100 sll Rem, R0=0
2a        0010    1111 1100 Rem=Rem-Div
2b        0010    0001 1100 Rem=Rem+Div
                  0011 1000 sll Rem, R0=0
3a        0010    0001 1000 Rem=Rem-Div
3b        0010    0011 0001 sll Rem, R0=1

4a        0010    0001 0001 Rem=Rem-Div
4b        0010    0010 0011 sll Rem, R0=1
Shift left half
of Rem right 1
                  0001 0011
```

remainder   quotient

Observations on Divide Version 2

- Same Hardware as Multiply: just need ALU to add or subtract, and 64-bit register to shift left or shift right
- **`Hi`** and **`Lo`** registers in MIPS combine to act as 64-bit register for multiply and divide
- Signed Divides: Simplest is to remember signs, make positive, and complement quotient and remainder if necessary
  - Note: Dividend and Remainder must have same sign
  - Note: Quotient negated if Divisor sign & Dividend sign disagree
    e.g., –7 ÷ 2 = –3, remainder = –1
  - And 7 ÷ (– 2)= – 3, remainder = 1
- Possible for quotient to be too large: if divide 64-bit integer by 1, quotient is 64 bits ("called saturation")

Review: MIPS Instructions, so far

Category	Instruction	Op Code	Example	Meaning
Arithmetic (R & I format)	add	0 and 32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	add unsigned	0 and 33	addu $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 and 34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	subt unsigned	0 and 35	subu $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	8	addi $s1, $s2, 6	$s1 = $s2 + 6
	add immediate unsigned	9	addiu $s1, $s2, 6	$s1 = $s2 + 6
	multiply	**0 and 24**	**mult $s1, $s2**	**hi || lo = $s1 * $s2**
	multiply unsigned	**0 and 25**	**multu $s1, $s2**	**hi || lo = $s1 * $s2**
	divide	**0 and 26**	**div $s1, $s2**	**lo = $s1/$s2, remainder in hi**
	divide unsigned	**0 and 27**	**divu $s1, $s2**	**lo = $s1/$s2, remainder in hi**

Review: MIPS ISA, continued

Category	Instr	Op Code	Example	Meaning
Shift (R format)	sll	0 and 0	sll $s1, $s2, 4	$s1 = $s2 << 4
	srl	0 and 2	srl $s1, $s2, 4	$s1 = $s2 >> 4
	sra	0 and 3	sra $s1, $s2, 4	$s1 = $s2 >> 4
Data Transfer (I format)	load word	35	lw $s1, 24($s2)	$s1 = Memory($s2+24)
	store word	43	sw $s1, 24($s2)	Memory($s2+24) = $s1
	load byte	32	lb $s1, 25($s2)	$s1 = Memory($s2+25)
	load byte unsigned	36	lbu $s1, 25($s2)	$s1 = Memory($s2+25)
	store byte	40	sb $s1, 25($s2)	Memory($s2+25) = $s1
	load upper imm	15	lui $s1, 6	$s1 = 6 * 2^{16}
	move from hi	0 and 16	mfhi $s1	$s1 = hi
	move to hi	0 and 17	mthi $s1	hi = $s1
	move from lo	0 and 18	mflo $s1	$s1 = lo
	move to lo	0 and 19	mtlo $s1	lo = $s1

Review: MIPS ISA- continued

Category	Instr	Op Code	Example	Meaning
Cond. Branch (I & R format)	br on equal	4	beq $s1, $s2, L	if ($s1==$s2) go to L
	br on not equal	5	bne $s1, $s2, L	if ($s1 !=$s2) go to L
	set on less than	0 and 42	slt $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set on less than unsigned	0 and 43	sltu $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set on less than immediate	10	slti $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0
	set on less than imm. unsigned	11	sltiu $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0
Uncond. Jump (J & R format)	jump	2	j 2500	go to 10000
	jump and link	3	jal 2500	go to 10000; $ra=PC+4
	jump register	0 and 8	jr $s1	go to $s1
	jump and link reg	0 and 9	jalr $s1, $s2	go to $s1, $s2=PC+4

Representing Big (and Small) Numbers

- What can be represented in N bits?
- Unsigned 0 to 2^N -1
- 2s Complement -2^{N-1} to 2^{N-1} - 1

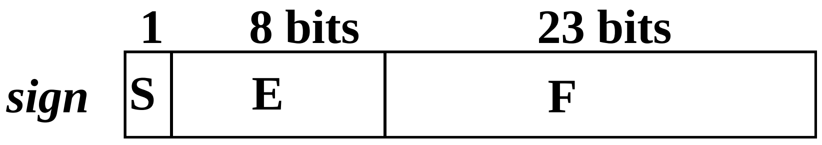
- What if we want to encode the approx. age of the earth? 4,600,000,000 or 4.6 x 109
- or the weight in kg of one a.m.u. (atomic mass unit) 0.0000000000000000000000000166 or 1.6 x 10-27
- There is no way we can encode either of the above in a 32-bit integer.

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

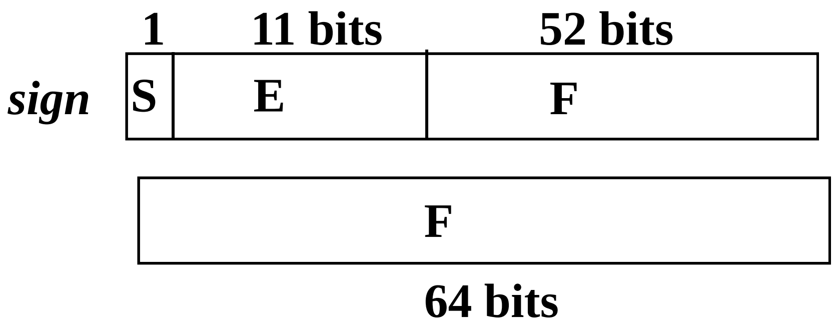
MIPS Register bit allocation

- Representation of floating point means that the binary point "floats" to get a non-0 bit before it. The binary point is not fixed.
- Since number of bits in register is **fixed** - we need to **`compromise`**



	1	8 bits	23 bits
sign	S	E	F

- When exponent is too large – or too small – an exception **`Overflow`**, or **`underflow`**

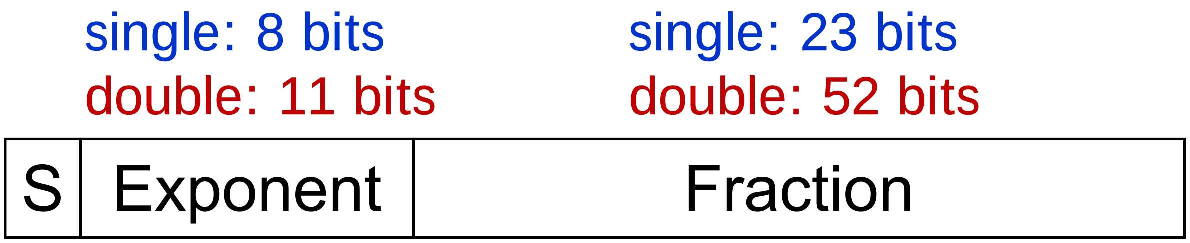


	1	11 bits	52 bits
sign	S	E	F

F

64 bits

IEEE Floating-Point Format



single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- Significand: 1+Fraction
- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

IEEE 754 - continued

- If the 23 significand bits are numbered from *left-to-right* then the floating point number represented by these bits is

$$N = (-1)^{S}\times(1+s_1\bullet 2^{-1}+s_2\bullet 2^{-2}+\ ..\ +s_{23}\bullet 2^{-23})\times 2^{(E-bias)}$$

- So the register containing the bits

1	1000 0011	111000…..	0

represents

$$N = (-1)\ \times(1+2^{-1}+2^{-2}+2^{-3})\times 2^{(2^7+2^1+2^0-127)}$$

$$N= -1x(1+0.5+0.25+0.125)\times 2^{(128+2+1-127)}$$

$$= -1.875\ \times 2^{\ (131-127)}$$

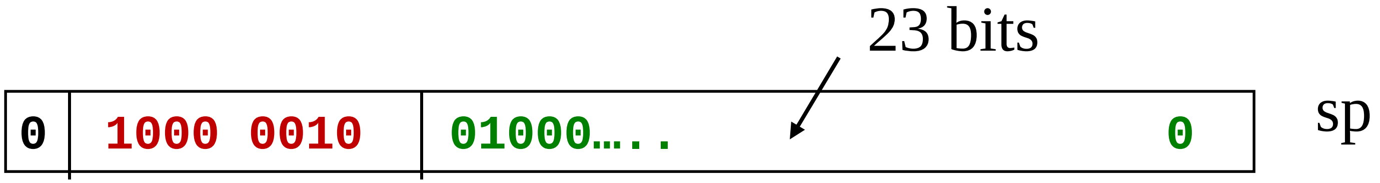
$$N = -1.875\times 2^4 = -1.875\times 16= -30$$

Exercise

- Show the IEEE 754 representation of 10_{ten} in single and double precision

$$10_{ten} = 1010_{two} = 1.01 \times 2^3$$ in normalized notation

- The sign bit is 0, the exponent is 3+127 = 130= $1000\ 0010_{two}$

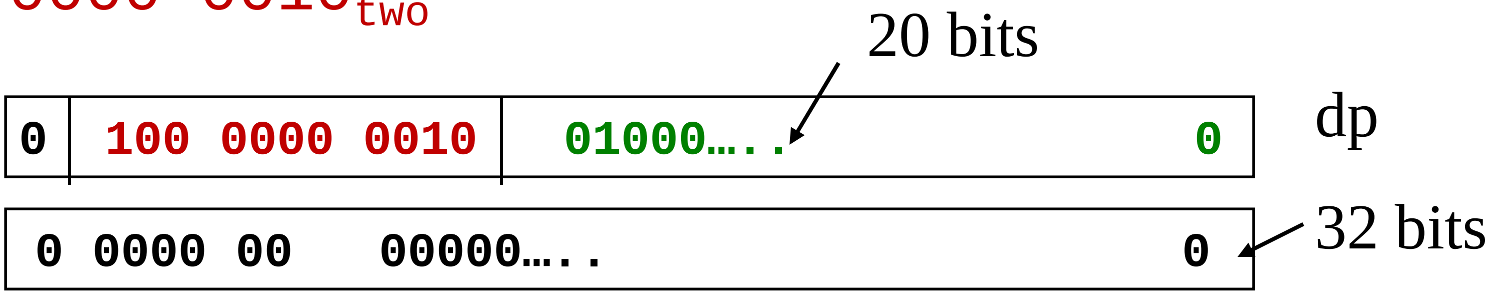


23 bits

0	1000 0010	01000…. . 0

sp

- In double precision the exponent is 3+1023= 1026 = $100\ 0000\ 0010_{two}$



20 bits

0	100 0000 0010	01000…. . 0

dp

0 0000 00 00000…. . 0

32 bits

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    ⇒ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    ⇒ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  11000000101000…00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Fxponent = 10000001_2 = 129
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

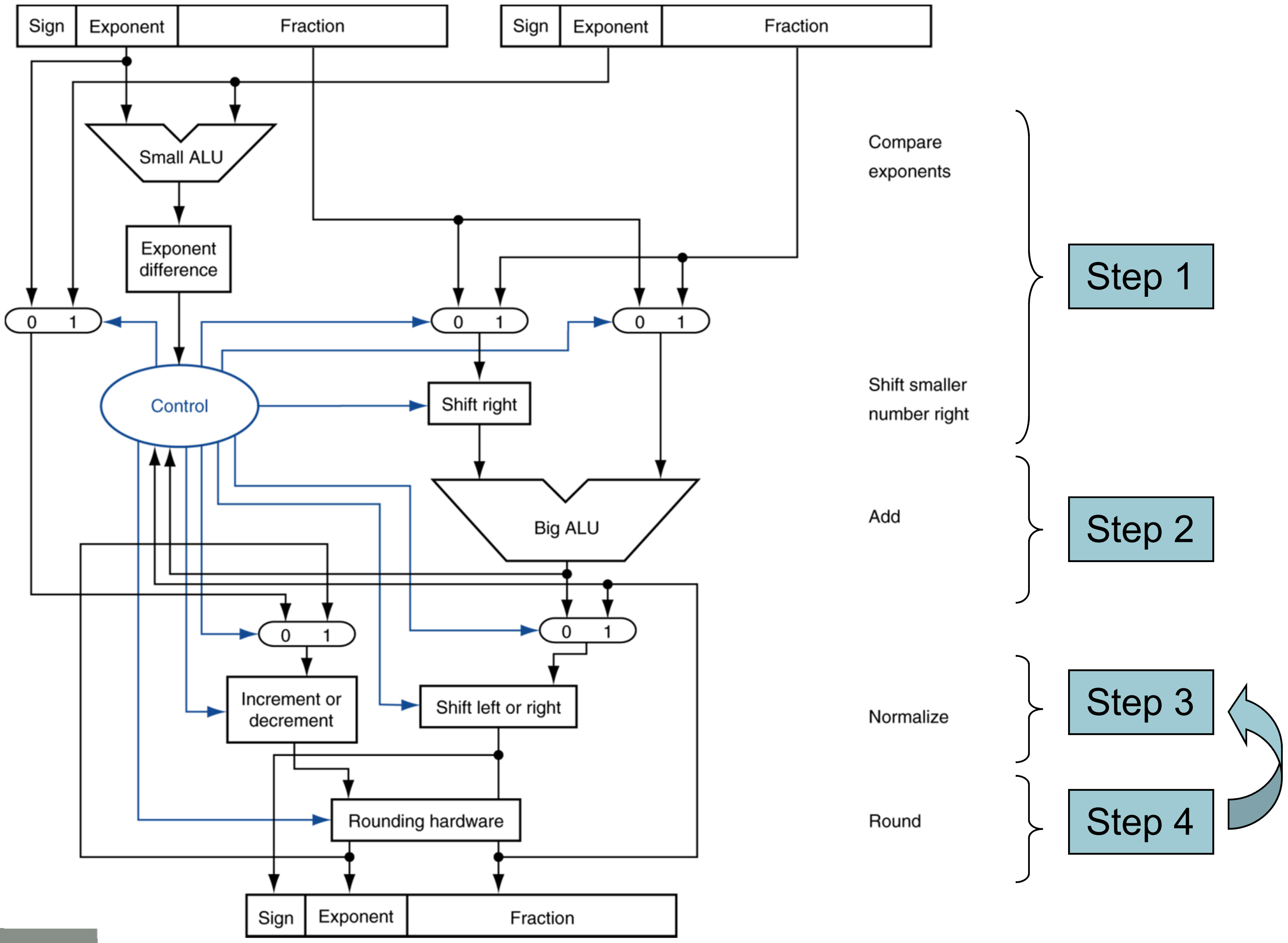
Floating-Point Addition

- Consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2–1 = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

FP Adder Hardware



Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)
- 1. Add exponents
  - Unbiased: –1 + –2 = –3
  - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in MIPS

- FP hardware is coprocessor 1
    - Adjunct processor that extends the ISA
- Separate FP registers
    - 32 single-precision: $f0, $f1, … $f31
    - Paired for double-precision: $f0/$f1, $f2/$f3, …
        - Release 2 of MIPs ISA supports 32 × 64-bit FP reg's
- FP instructions operate only on FP registers
    - Programs generally don't do integer ops on FP data, or vice versa
    - More registers with minimal code-size impact
- FP load and store instructions
    - `lwc1`, `ldc1`, `swc1`, `sdc1`
        - e.g., `ldc1 $f8, 32($sp)`

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*`xx`*`.s`, `c.`*`xx`*`.d` (*xx* is eq, `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

Concluding Remarks

- ISAs support arithmetic
    - Signed and unsigned integers
    - Floating-point approximation to reals
- Bounded range and precision
    - Operations can overflow and underflow