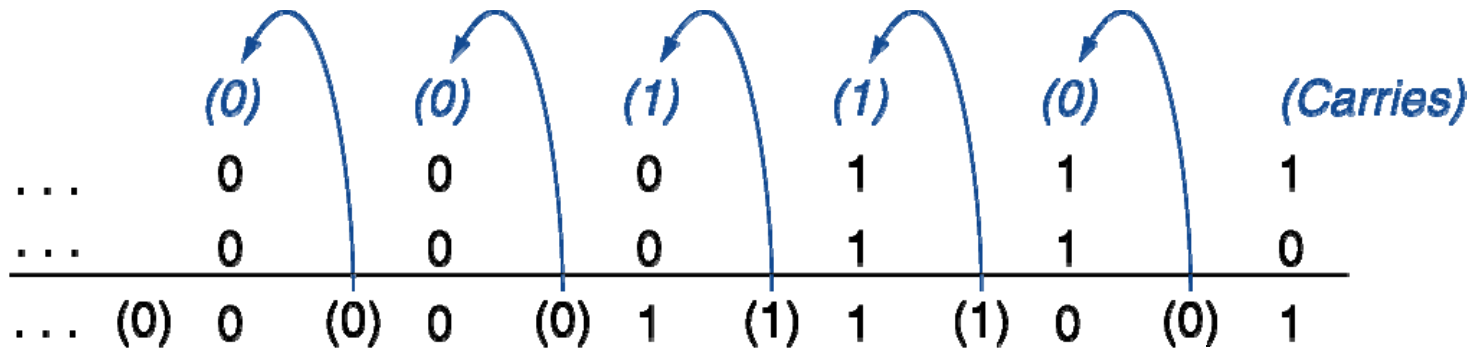


Chapter 3

Arithmetic for Computers

Integer Addition

- Example: $7 + 6$



- **Overflow** if result out of range
 - Adding +ve and -ve operands, no overflow
 - Adding two +ve operands
 - Overflow if result sign is 1
 - Adding two -ve operands
 - Overflow if result sign is 0

Integer Subtraction

- Add negation of second operand

- Example: $7 - 6 = 7 + (-6)$

$$\begin{array}{r} +7: \quad 0000\ 0000\ \dots\ 0000\ 0111 \\ -6: \quad 1111\ 1111\ \dots\ 1111\ 1010 \\ \hline +1: \quad 0000\ 0000\ \dots\ 0000\ 0001 \end{array}$$

- Overflow if result out of range
 - Subtracting two +ve or two -ve operands, no overflow
 - Subtracting +ve from -ve operand
 - Overflow if result sign is 0
 - Subtracting -ve from +ve operand
 - Overflow if result sign is 1

Dealing with Overflow

- Some languages (e.g., C) ignore overflow
 - Use MIPS addu, addui , subu instructions
 - Saturated arithmetic
- 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

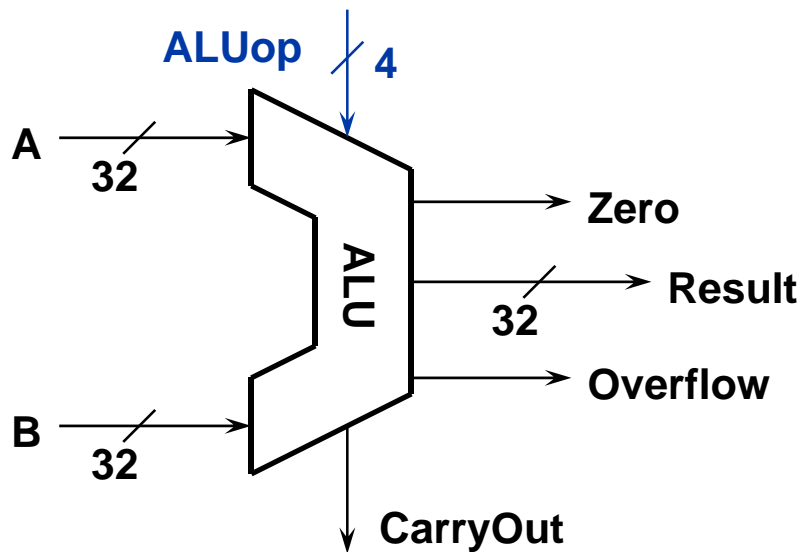
Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
 - overflow when adding two positives yields a negative
 - or, adding two negatives gives a positive
 - or, subtract a negative from a positive and get a negative
 - or, subtract a positive from a negative and get a positive
- Consider the operations $A + B$, and $A - B$
 - Can overflow occur if B is 0 ?
 - Can overflow occur if A is 0 ?
- Overflow detection

Operation	A	B	Result indicating overflow
$A+B$	≥ 0	≥ 0	< 0
$A+B$	< 0	< 0	≥ 0
$A-B$	≥ 0	< 0	< 0
$A-B$	< 0	≥ 0	≥ 0

Designing ALU for MIPS

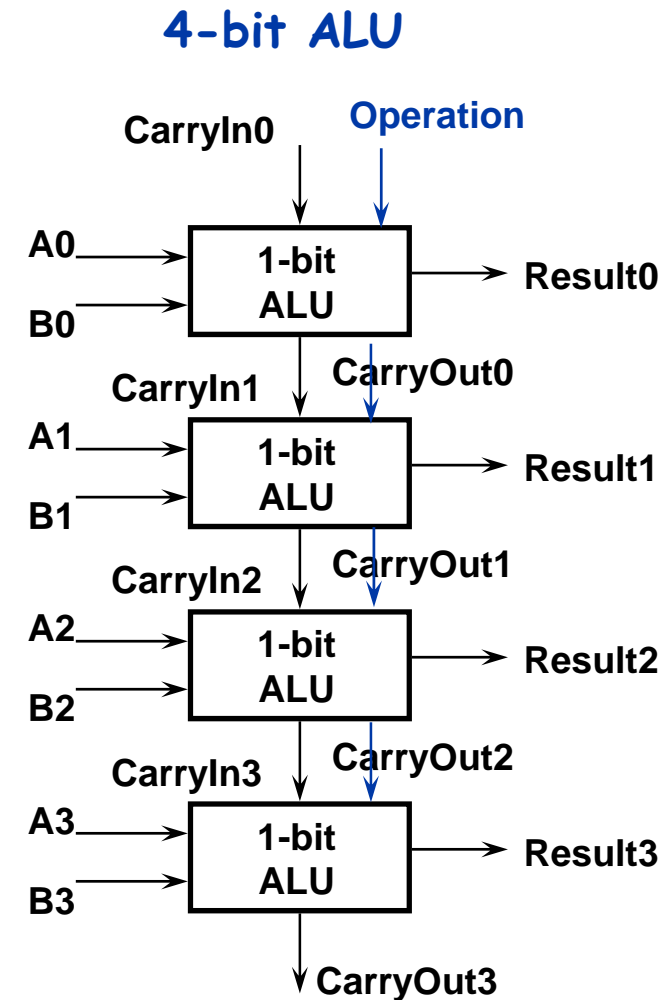
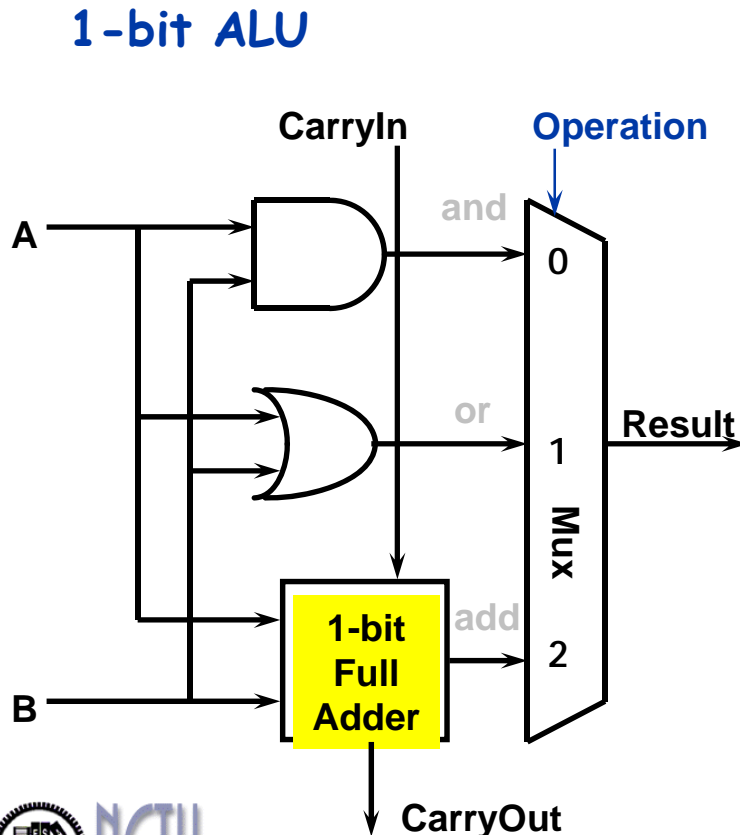
- Arithmetic logic unit (ALU) performs arithmetic and logical operations
 - **add, sub**: two's complement adder/subtractor with overflow detection
 - **and, or, nor** : logical AND, logical OR, logical NOR
 - **slt** (set on less than): two's complement adder with inverter, check sign bit of result



<u>(ALUop)</u>	<u>Function</u>
0000	and
0001	or
0010	add
0110	subtract
0111	set-on-less-than
1100	nor

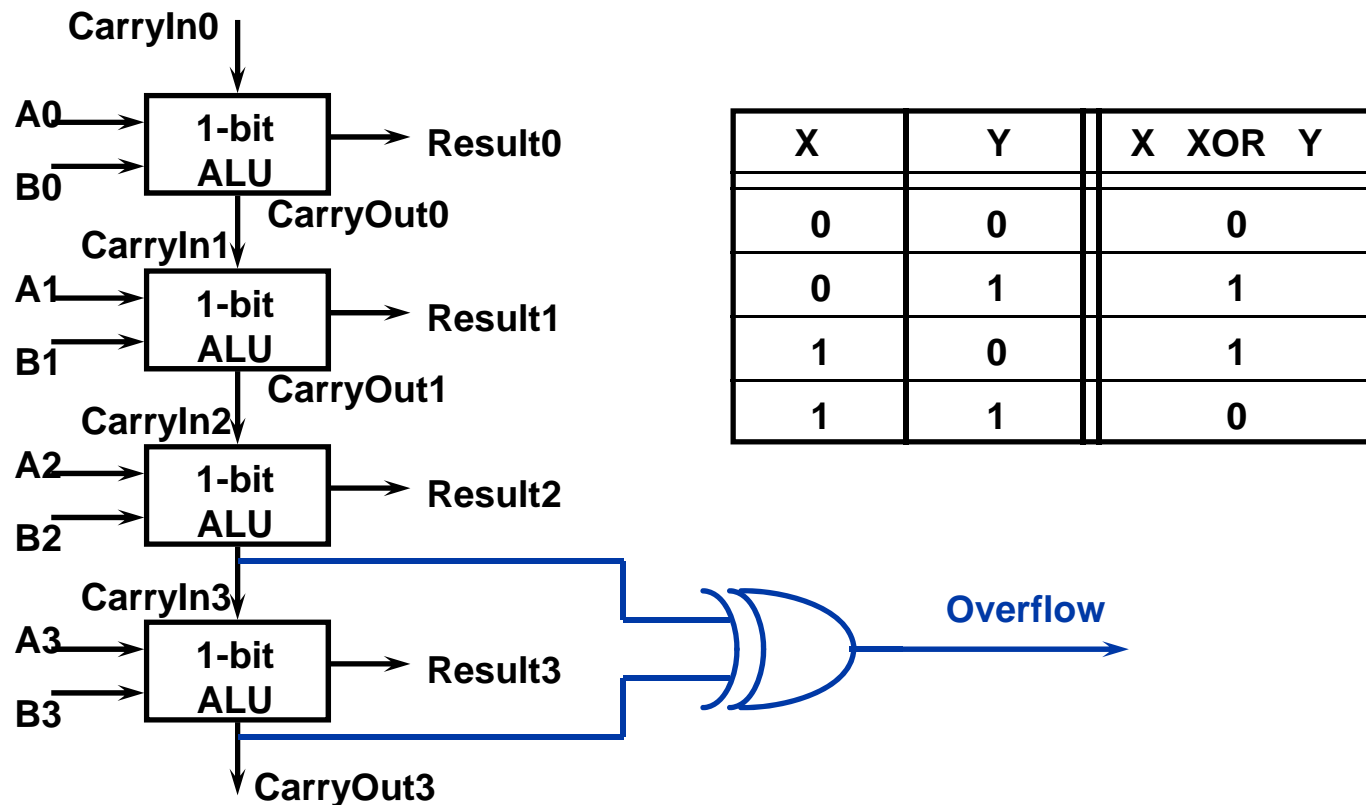
A 4-bit ALU Example

- Design trick 3: take pieces you know (or can imagine) and try to put them together



Overflow Detection Logic

- $\text{Overflow} = \text{CarryIn}[N-1] \text{ XOR } \text{CarryOut}[N-1]$

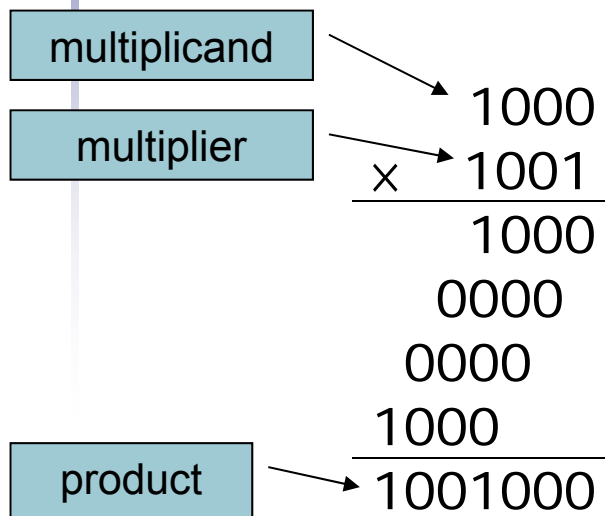


Arithmetic for Multimedia

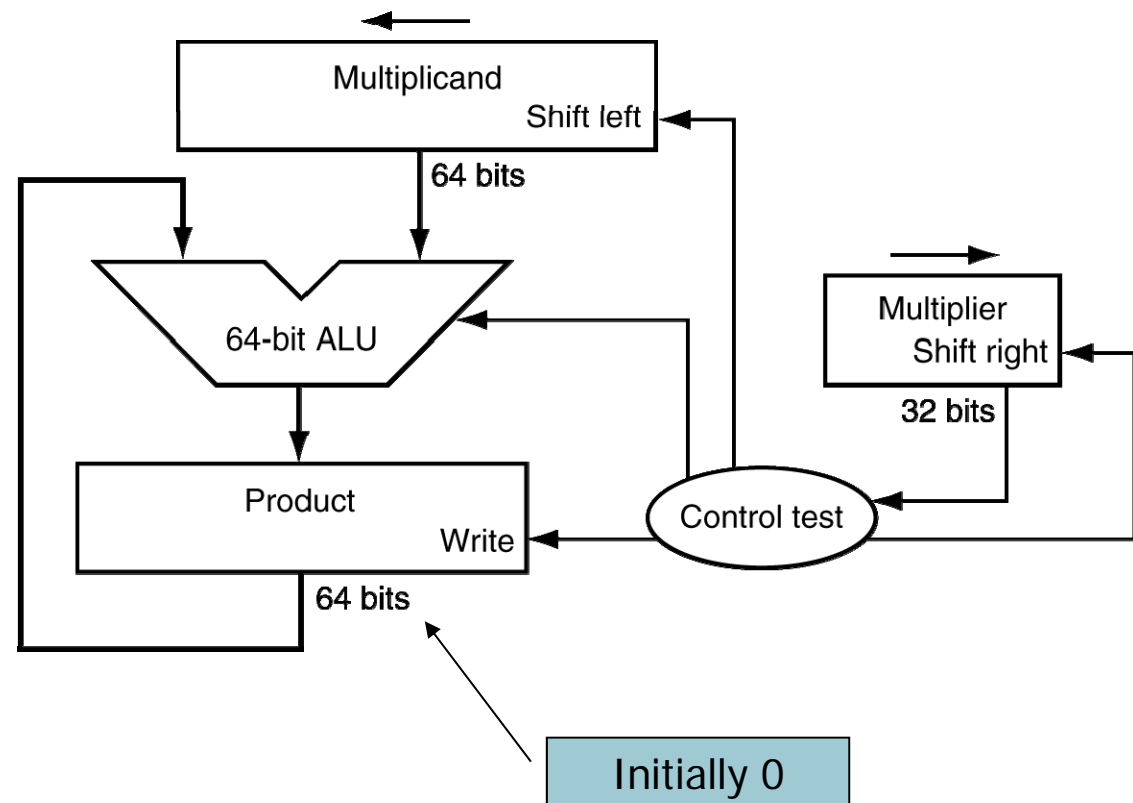
- Graphics and media processing operates on vectors of 8-bit and 16-bit data
 - Use 64-bit adder, with partitioned carry chain
 - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
 - SIMD (single-instruction, multiple-data)
- Saturating operations
 - On overflow, result is largest representable value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

Multiplication

- Start with long-multiplication approach



Length of product is the sum of operand lengths



Multiplication in MIPS

```
mult $t1, $t2 # t1 * t2
```

- No destination register: product could be $\sim 2^{64}$; need two special registers to hold it
- 3-step process:

```
      $t1  01111111111111111111111111111111
X     $t2  01000000000000000000000000000000
```

```
0001111111111111111111111111111111 11000000000000000000000000000000
```

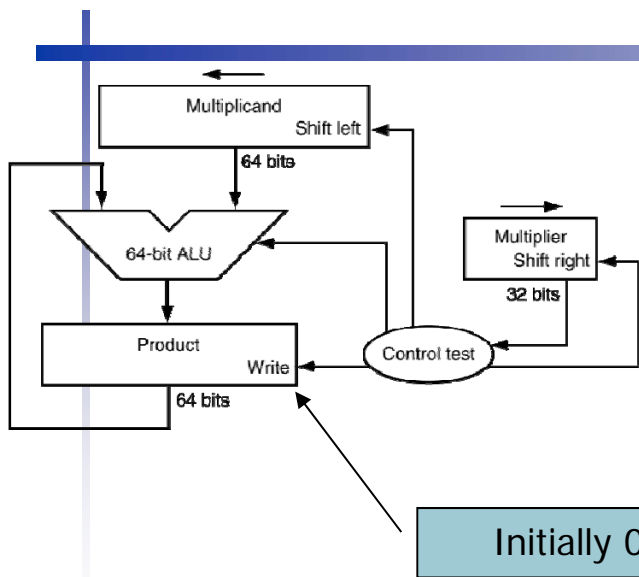
Hi

Lo

```
mfhi $t3  $t3 00011111111111111111111111111111
```

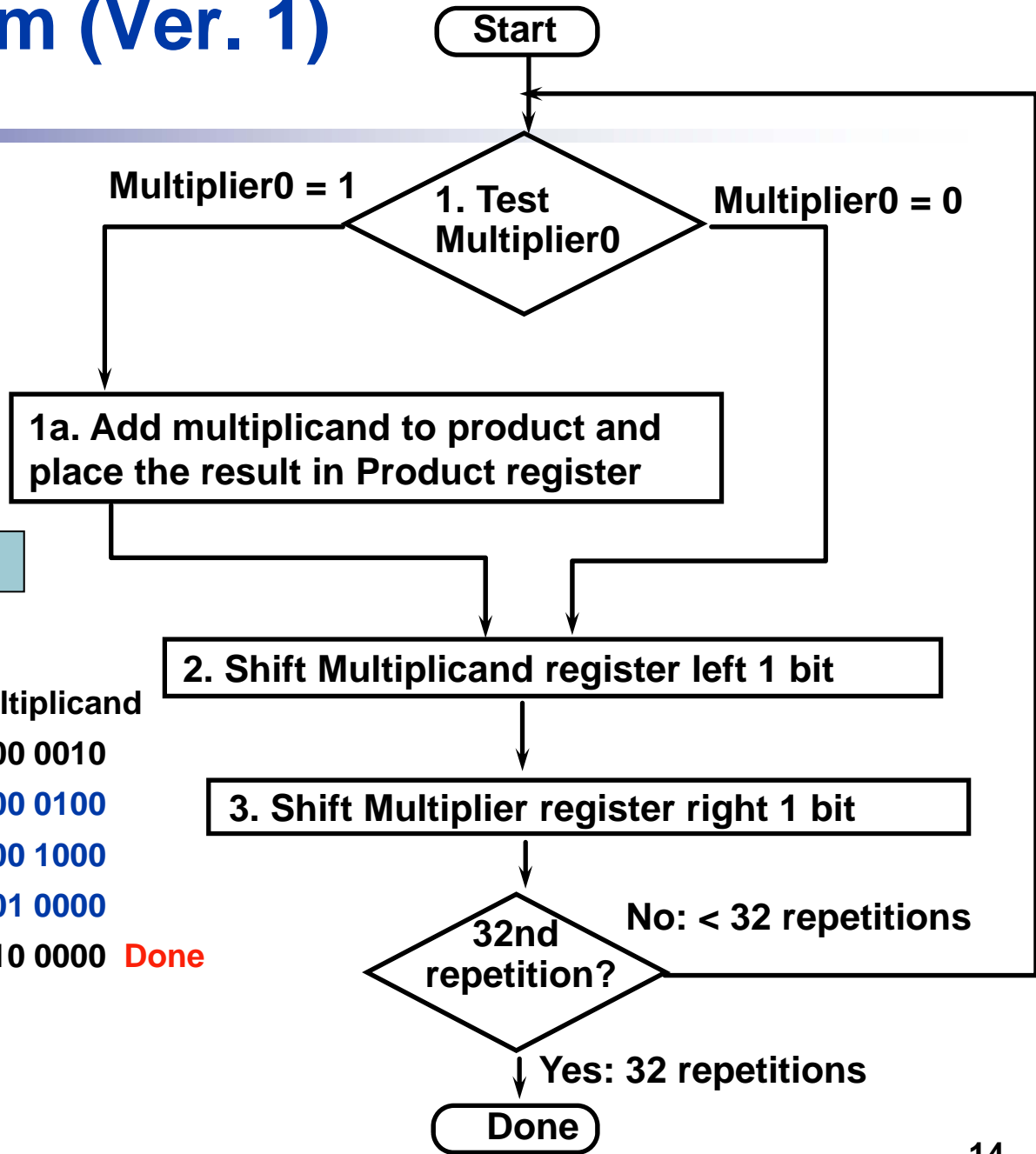
```
mflo $t4  $t4 11000000000000000000000000000000
```

Multiply Algorithm (Ver. 1)



0010 x 0011

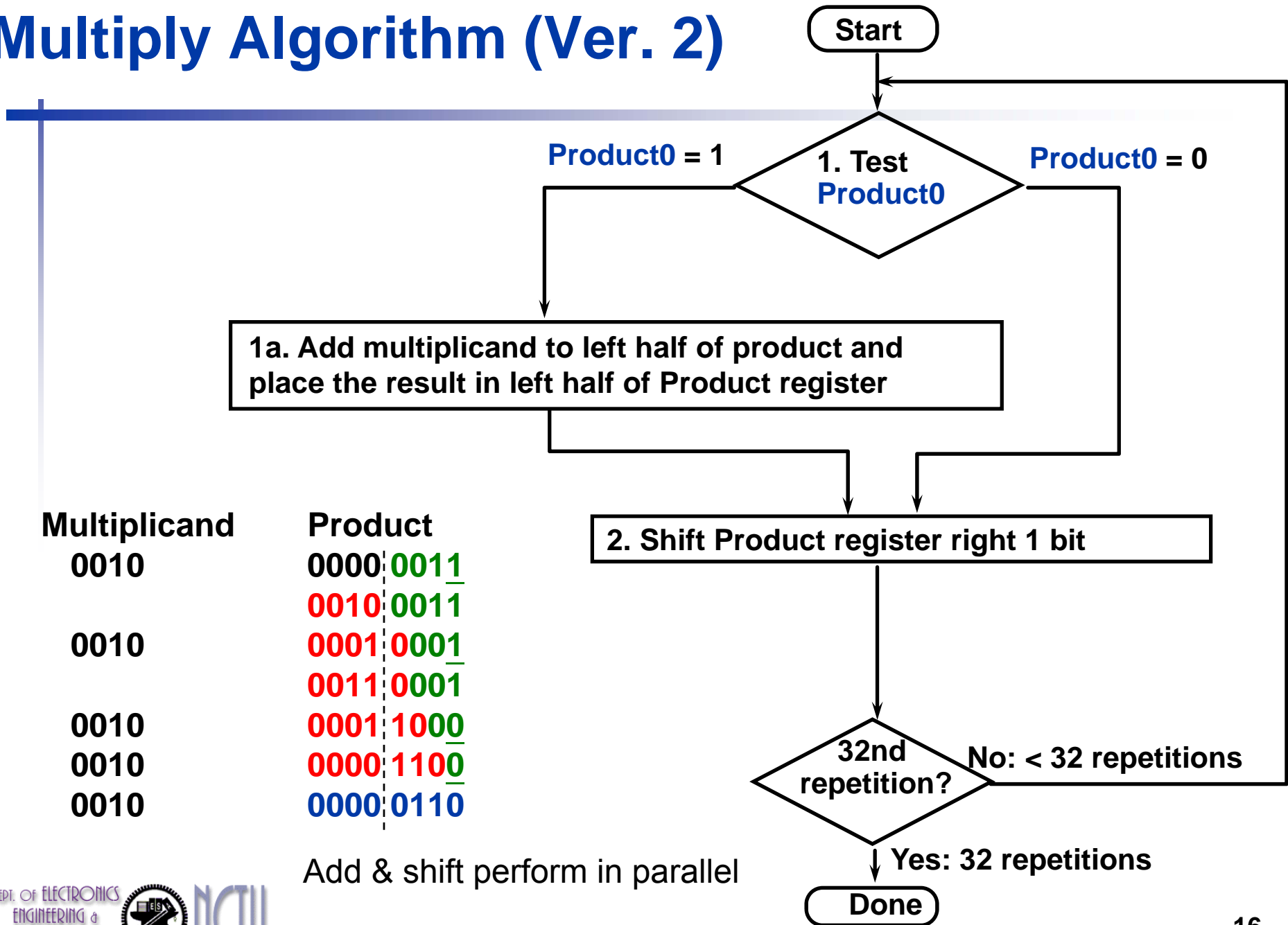
Product	Multiplier	Multiplicand
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110	0000	0001 0000
0000 0110	0000	0010 0000 Done



Observations

- 1 clock per cycle => too slow
 - Ratio of multiply to add 5:1 to 100:1
- Half of the bits in multiplicand always 0
 - => 64-bit adder is wasted
- 0's inserted in right of multiplicand as shifted
 - => least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?
- Product register wastes space => combine Multiplier and Product register

Multiply Algorithm (Ver. 2)

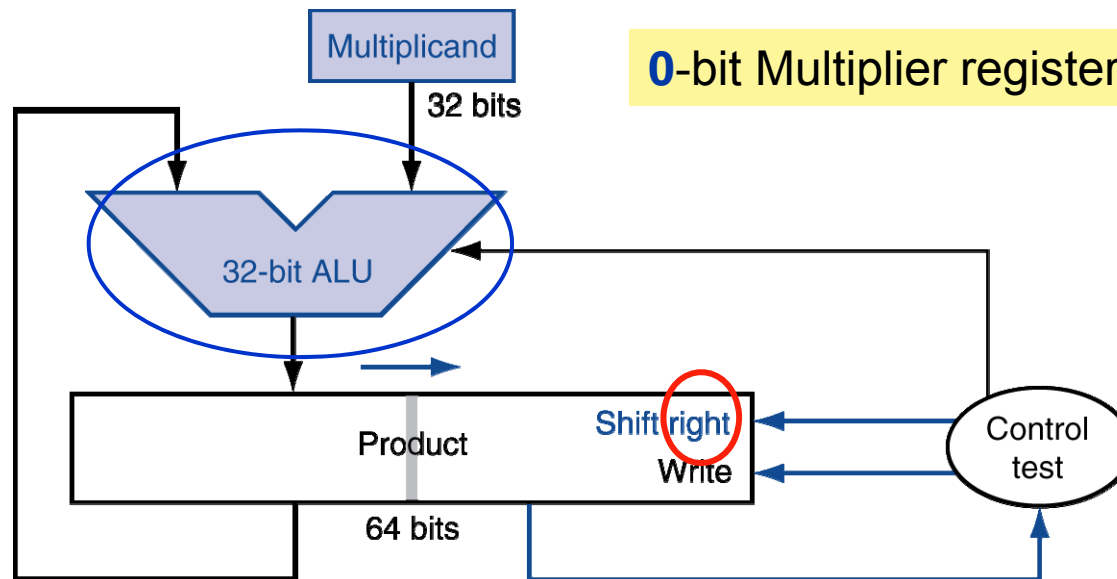


Multiplicand	Product
0010	0000 0011
0010	0010 0011
0010	0001 0001
0010	0011 0001
0010	0001 1000
0010	0000 1100
0010	0000 0110

Add & shift perform in parallel

Optimized Multiplier

- Perform steps in parallel: add/shift



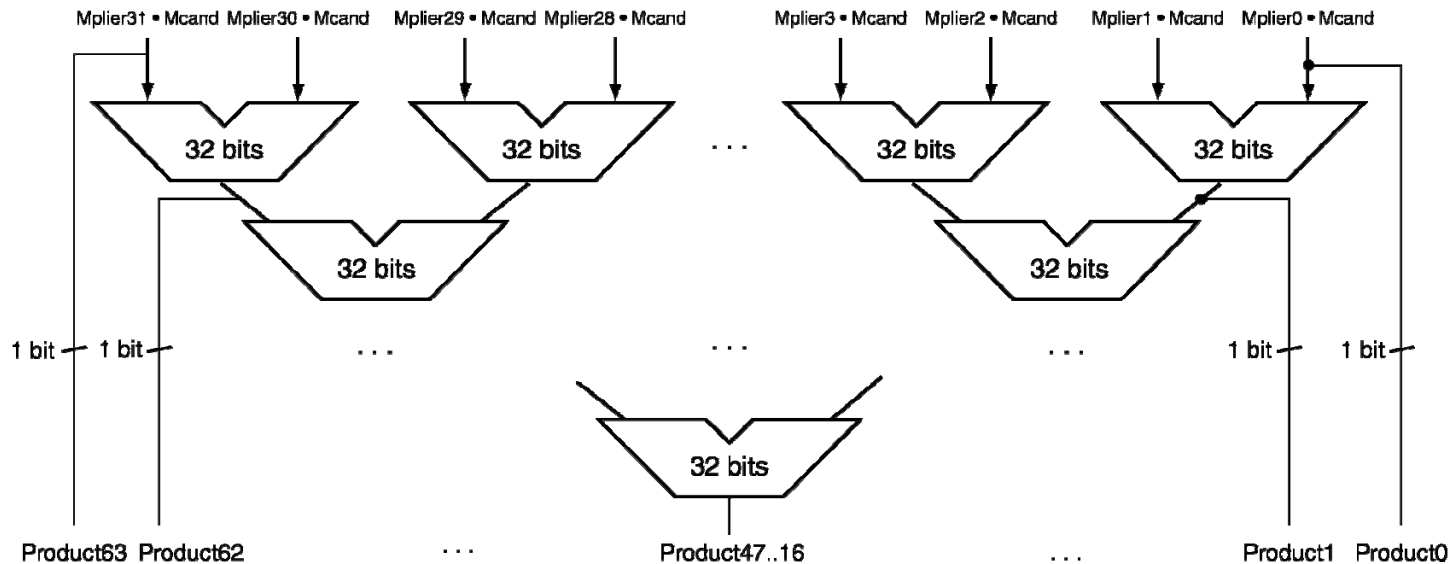
- One cycle per partial-product addition
 - That's ok, if frequency of multiplications is low

Concluding Remarks

- 2 steps per bit because multiplier and product registers combined
- MIPS registers Hi and Lo are left and right half of Product register
=> this gives the MIPS instruction MultU
- **What about signed multiplication?**
 - The easiest solution is to make both positive and remember whether to complement product when done (leave out sign bit, run for 31 steps)
 - Apply definition of 2's complement
 - sign-extend partial products and subtract at end
 - **Booth's Algorithm** is an elegant way to multiply signed numbers using same hardware as before and save cycles

Faster Multiplier

- Uses multiple adders
 - Cost/performance tradeoff

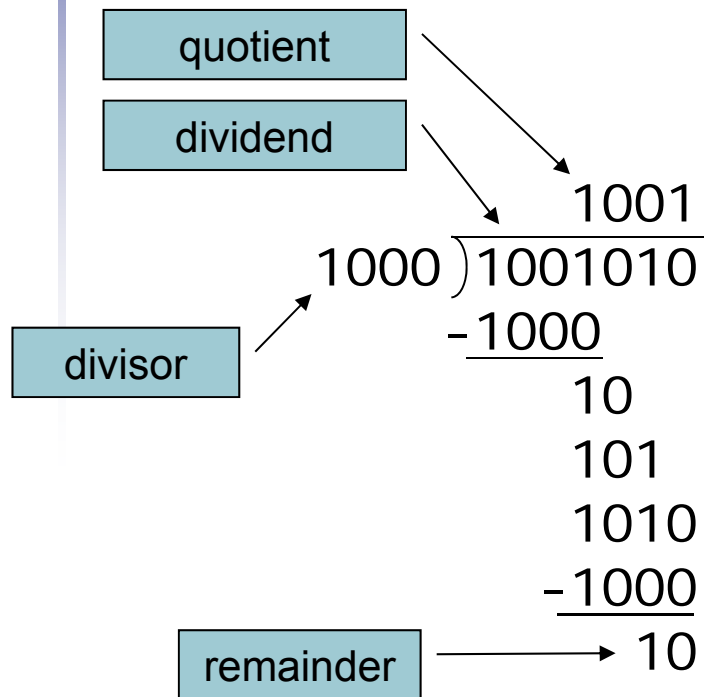


- Can be pipelined
 - Several multiplication performed in parallel

MIPS Multiplication Instructions

- 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

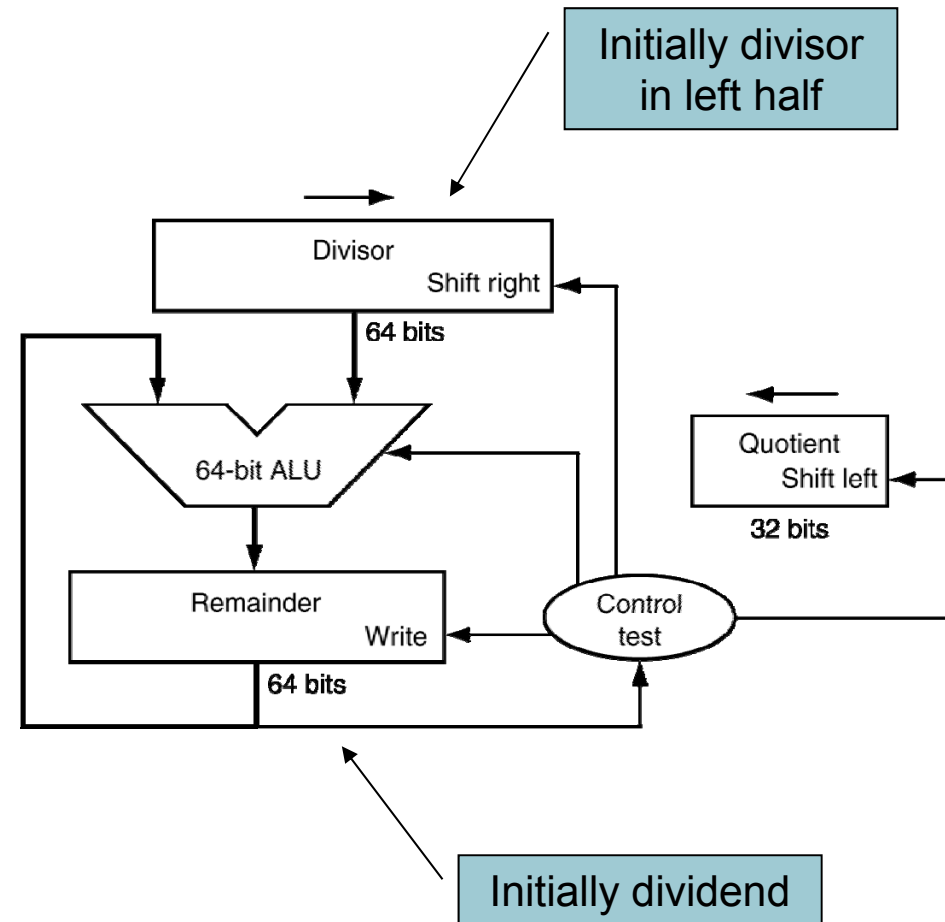
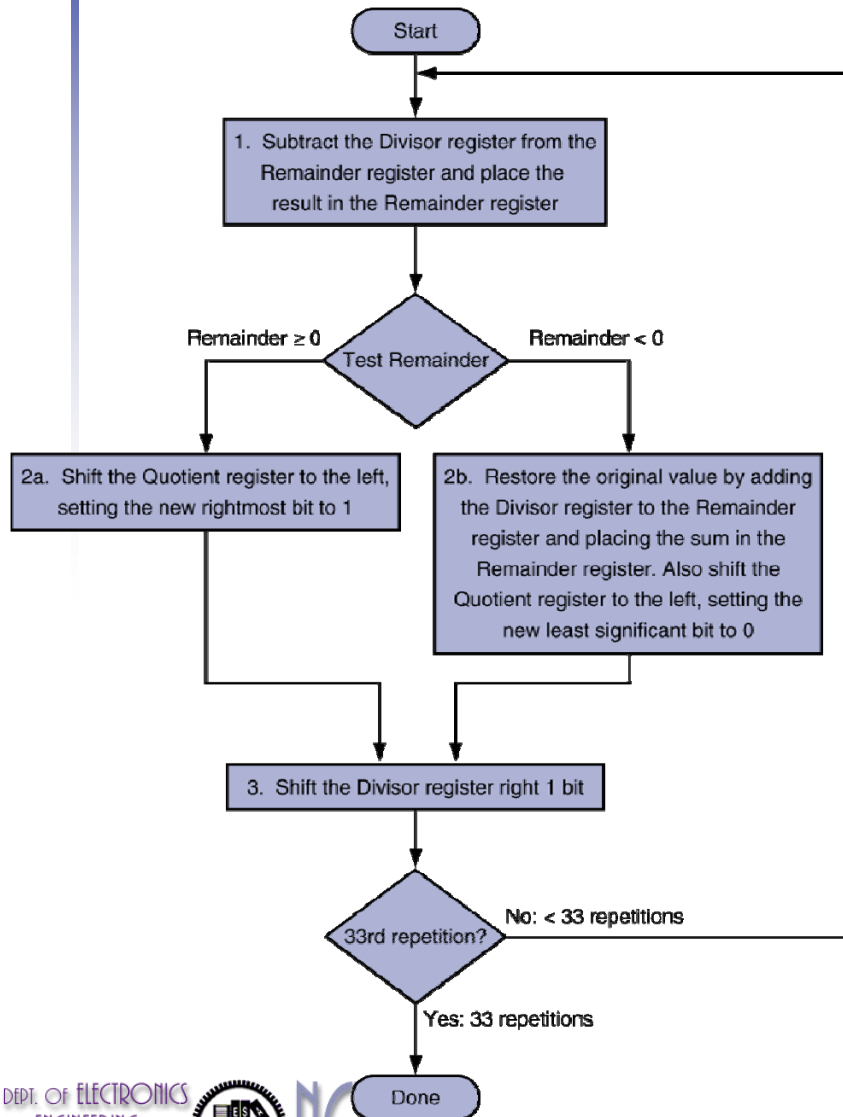
Division



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

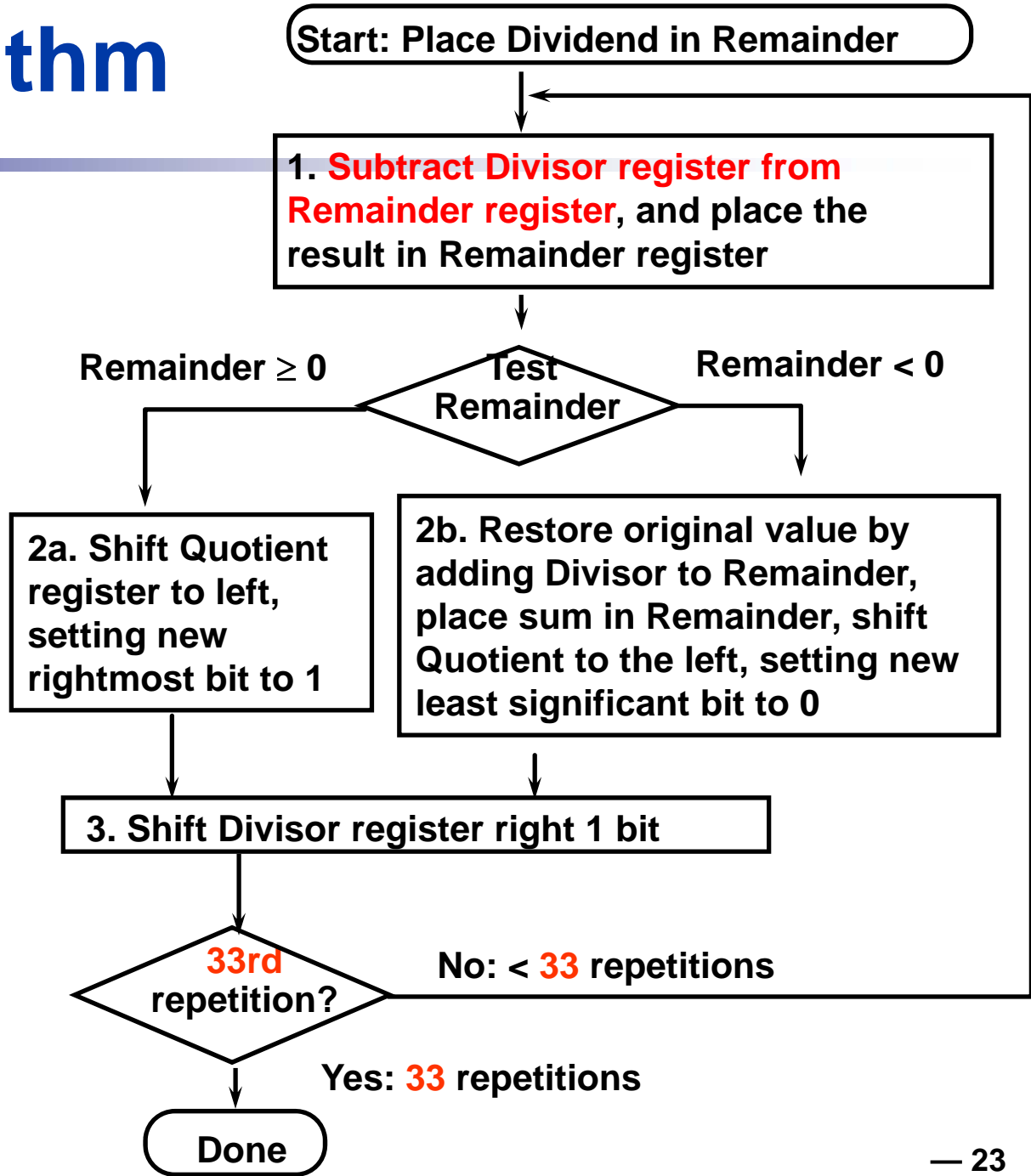
- Check for 0 divisor
- Long division approach
 - If divisor \leq dividend bits
 - 1 bit in quotient, subtract
 - Otherwise
 - 0 bit in quotient, bring down next dividend bit
- Restoring division
 - Do the subtract, and if remainder goes < 0 , add divisor back
- Signed division
 - Divide using absolute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

Division Hardware



Divide Algorithm

Quot.	Divisor	Rem.
0000	<u>0010</u> 0000	0000 <u>0111</u>
		11100111
		00000111
0000	<u>0001</u> 0000	00000111
		11110111
		00000111
0000	<u>0000</u> 1000	00000111
		11111111
		00000111
0000	<u>0000</u> 0100	00000111
		00000011
0001		00000011
0001	<u>0000</u> 0010	00000011
		00000001
0011		00000001
0011	<u>0000</u> 0001	00000001

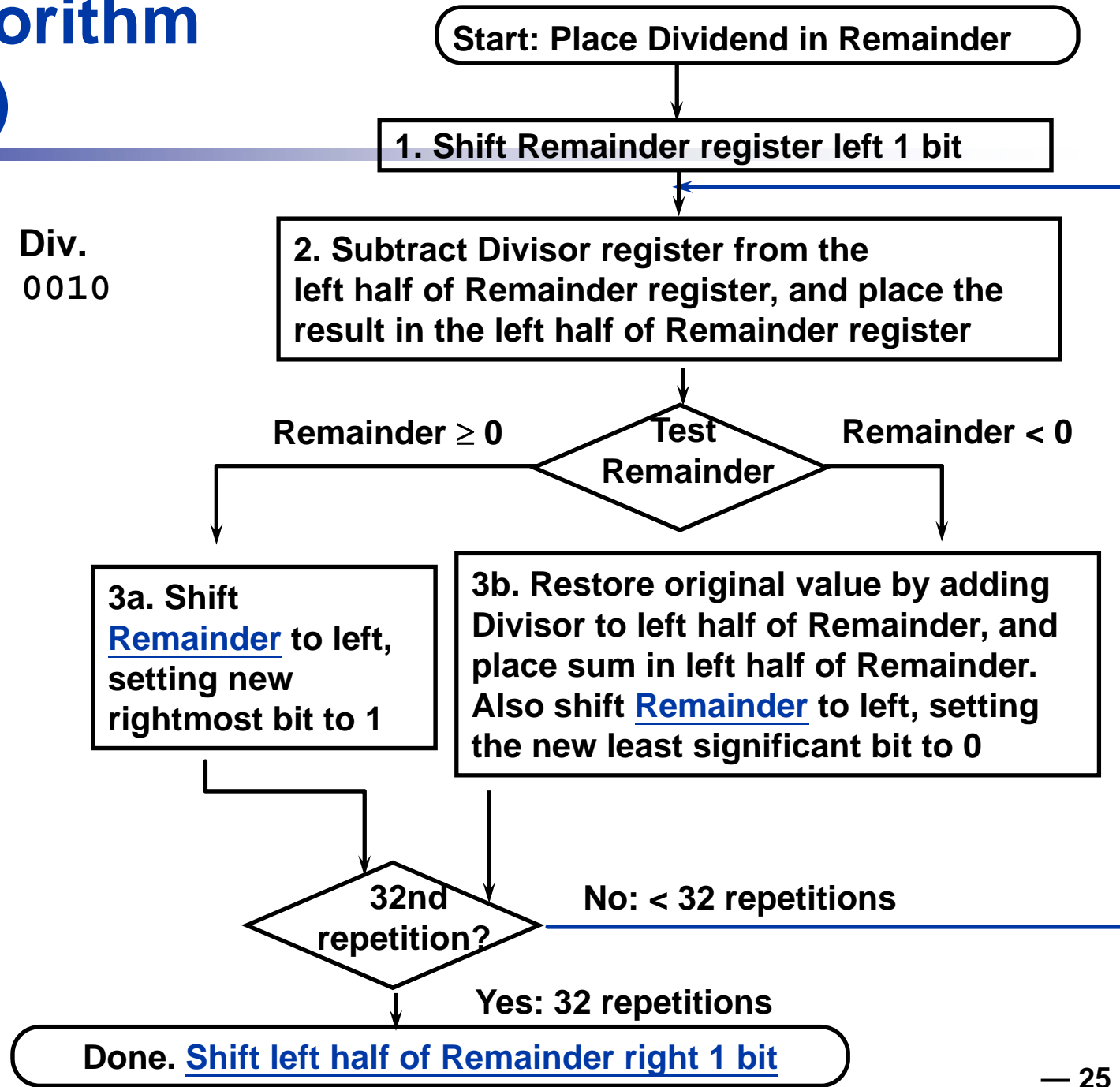


Observations

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

Divide Algorithm (Version 2)

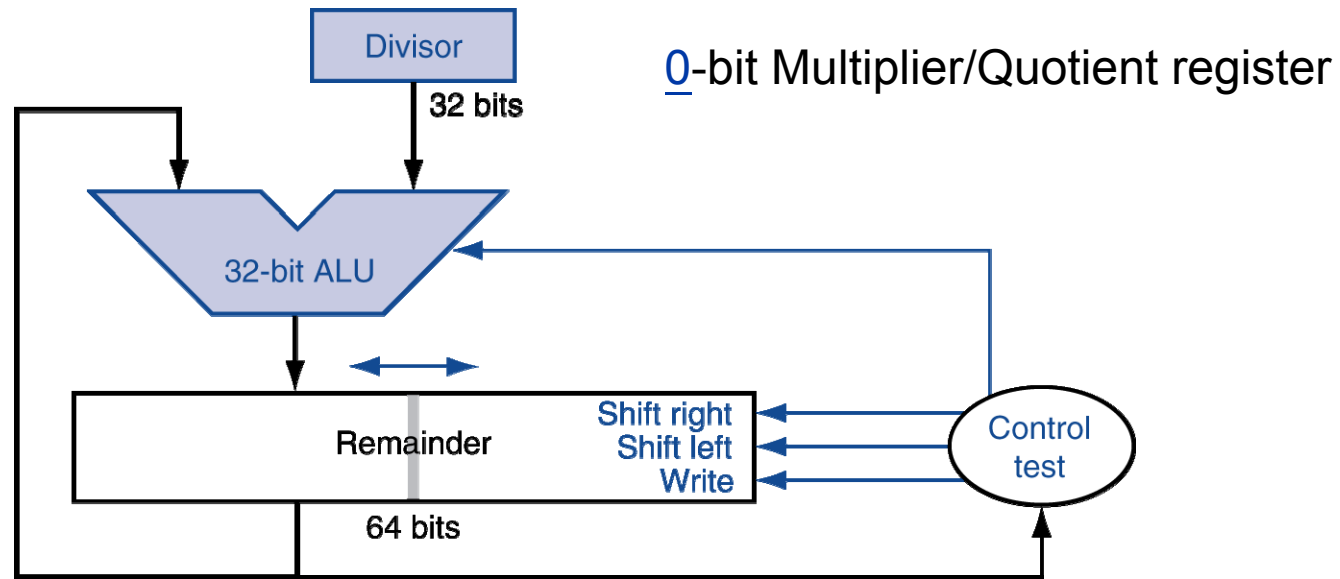
Step	Remainder	Div.
0	0000 0111	0010
1.1	0000 1110	
1.2	1110 1110	
1.3b	0001 1100	
2.2	1111 1100	
2.3b	0011 1000	
3.2	0001 1000	
3.3a	0011 0001	
4.2	0001 0001	
4.3a	0010 0011	
	0001 0011	



Concluding Remarks

- Observations: Divide vs. Multiply
 - 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

Optimized Divider



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

Faster Division

- Can't use parallel hardware as in multiplier
 - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step
 - Still require multiple steps

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

Floating Point

- Representation for non-integral numbers
 - Including very small and very large numbers
- Like scientific notation
 - -2.34×10^{56} ← normalized
 - $+0.002 \times 10^{-4}$ ← not normalized
 - $+987.02 \times 10^9$ ← not normalized
- In binary
 - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types float and double in C

Floating Point Standard

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

IEEE 754 Standard (1/2)

- Regarding single precision (SP), DP similar
- Sign bit:
 - 1 means negative
 - 0 means positive
- Significand:
 - To pack more bits, **leading 1** implicit for normalized numbers
 - 1 + 23 bits single, 1 + 52 bits double
 - always true: **$0 \leq \text{Significand} < 1$**
(for normalized numbers)
- Note: 0 has no leading 1, so reserve exponent value 0 just for number 0

IEEE 754 Standard (2/2)

- Exponent:
 - **Need to represent positive and negative exponents**
 - Also want to compare FP numbers as if they were integers, to help in value comparisons
 - If use **2's complement** to represent?
e.g., 1.0×2^{-1} versus $1.0 \times 2^{+1}$ ($1/2$ versus 2)

$1/2$

0	1111 1111	000 0000 0000 0000 0000 0000
---	-----------	------------------------------

2

0	0000 0001	000 0000 0000 0000 0000 0000
---	-----------	------------------------------

If we use integer comparison for these two words, we will conclude that $1/2 > 2$!!!

Biased (Excess) Notation

- let notation 0000 be most negative, and 1111 be most positive
- Example: Biased 7

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



IEEE 754 Standard

- Using biased notation

- the bias is the number subtracted to get the real number
- IEEE 754 uses bias of 127 for single precision:
Subtract 127 from Exponent field to get actual value for exponent
- 1023 is bias for double precision
- The example becomes

1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 0000 0000 0000 0000 0000

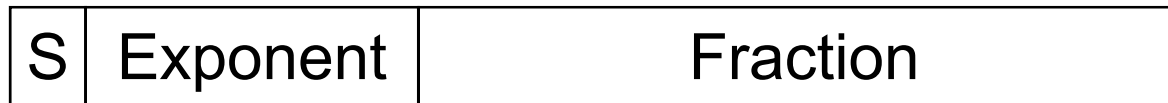
IEEE Floating-Point Format

single: 8 bits

single: 23 bits

double: 11 bits

double: 52 bits



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

- S: sign bit (0 \Rightarrow non-negative, 1 \Rightarrow negative)
- Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
 - 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

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: 000000000001
⇒ 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: 111111111110
⇒ 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 Precision

- Relative precision
 - all fraction bits are significant
 - Single: approx 2^{-23}
 - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
 - Double: approx 2^{-52}
 - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Floating-Point Example

- Represent -0.75
 - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
 - $S = 1$
 - Fraction = $1000\dots00_2$
 - Exponent = $-1 + \text{Bias}$
 - Single: $-1 + 127 = 126 = 01111110_2$
 - Double: $-1 + 1023 = 1022 = 011111111110_2$
- Single: $10111111101000\dots00$
- Double: $101111111111101000\dots00$

Floating-Point Example

- What number is represented by the single-precision float

11000000101000...00

- $S = 1$

- Fraction = $01000...00_2$

- Exponent = $10000001_2 = 129$

- $$\begin{aligned}x &= (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)} \\ &= (-1) \times 1.25 \times 2^2 \\ &= -5.0\end{aligned}$$

Concluding Remarks

- What have we defined so far? (single precision)

<u>Exponent</u>	<u>Significand</u>	<u>Object</u>
0	0	<u>???</u>
0	nonzero	<u>???</u>
1-254	anything	+/- floating-point
255	0	<u>???</u>
255	nonzero	<u>???</u>

Zero and Special Numbers

- Represent 0?

- exponent all zeroes
- significand all zeroes too
- What about sign?
- +0: 0 00000000 000000000000000000000000
- -0: 1 00000000 000000000000000000000000

- Why two zeroes?

- Helps in some limit comparisons

- Special numbers

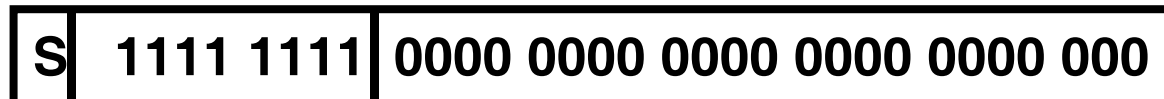
- Range: $1.0 \times 2^{-126} \approx 1.8 \times 10^{-38}$
 - What if result too small? ($>0, < 1.8 \times 10^{-38} \Rightarrow$ Underflow!)
 - What if result too large? ($> 3.4 \times 10^{38} \Rightarrow$ Overflow!)

Gradual Underflow

- Represent denormalized numbers (denorms)
 - Exponent : all zeroes
 - Significand : non-zeroes
 - Allow a number to degrade in significance until it become 0 (gradual underflow)
- The smallest normalized number
 - $1.0000\ 0000\ 0000\ 0000\ 0000\ 0000 \times 2^{-126}$

Representation for +/- Infinity

- In FP, divide by zero should produce +/- infinity, not overflow
- Why?
 - OK to do further computations with infinity, e.g., $X/0 > Y$ may be a valid comparison
- IEEE 754 represents +/- infinity
 - Most positive exponent reserved for infinity
 - Significands all zeroes



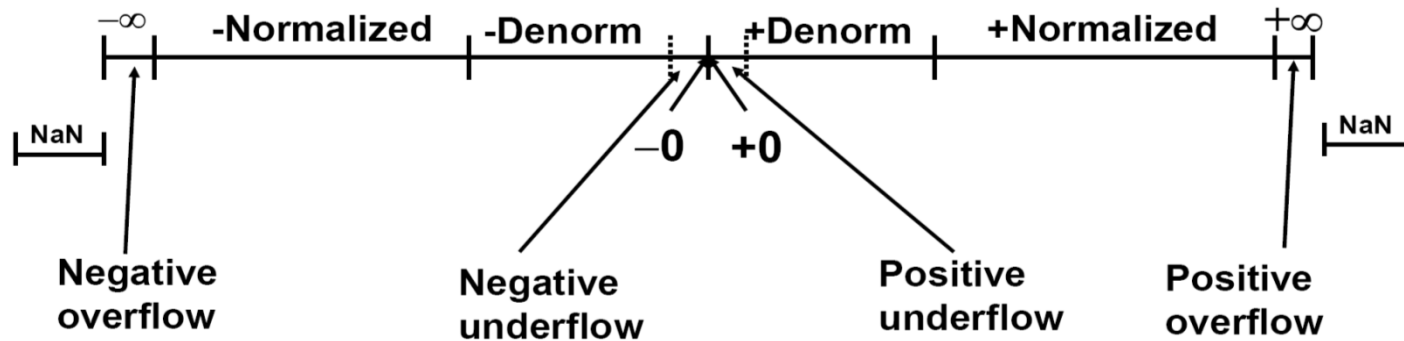
Representation for Not a Number

- What do I get if I calculate $\text{sqrt}(-4.0)$ or $0/0$?
 - If infinity is not an error, these should not be either
 - They are called *Not a Number* (NaN)
 - Exponent = 255, Significand nonzero
- Why is this useful?
 - Hope NaNs help with debugging?
 - They contaminate: $\text{op}(\text{NaN}, X) = \text{NaN}$
 - OK if calculate but don't use it

IEEE 754 Encoding of FP Numbers

- What have we defined so far? (single-precision)

<u>Exponent</u>	<u>Significand</u>	<u>Object</u>
0	0	0
0	nonzero	denom
1-254	anything	+/- fl. pt. #
255	0	+/- infinity
255	nonzero	NaN



Floating-Point Addition

- Now 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

Floating-Point Addition Algorithm

Basic addition algorithm:

compute $Y_e - X_e$ (to align binary point)

(1) right shift the smaller number, say X_m , that many positions to form $X_m \times 2^{X_e - Y_e}$

(2) compute $X_m \times 2^{X_e - Y_e} + Y_m$

if demands normalization, then normalize:

(3) left shift result, decrement result exponent

right shift result, increment result exponent

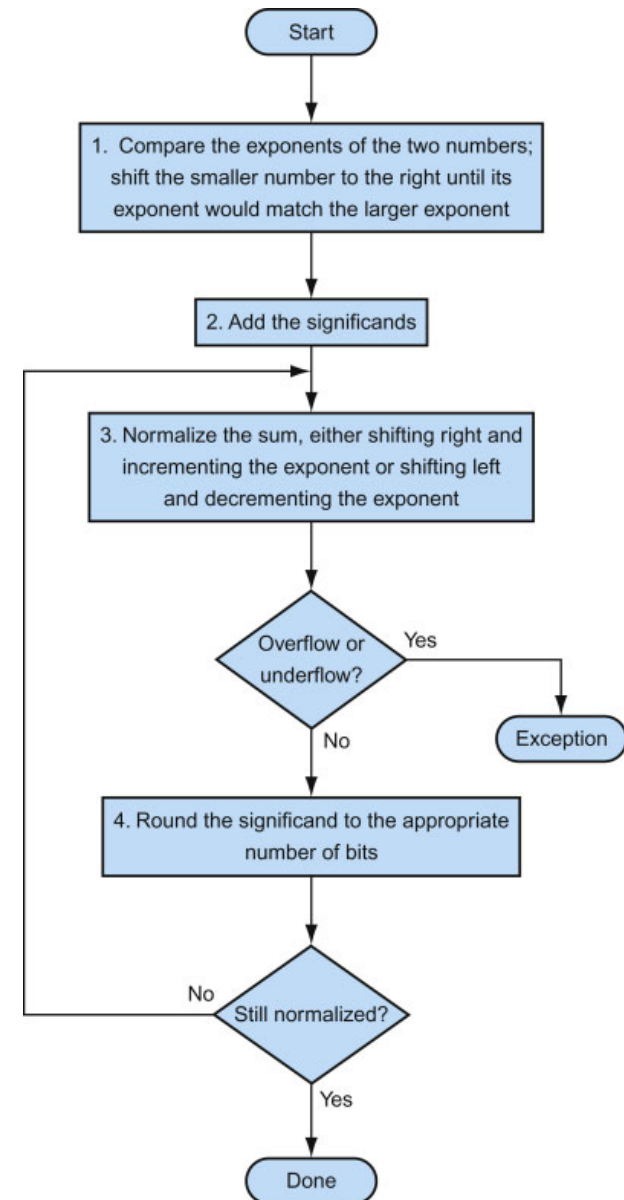
(3.1) check overflow or underflow during the shift

(4) round the mantissa

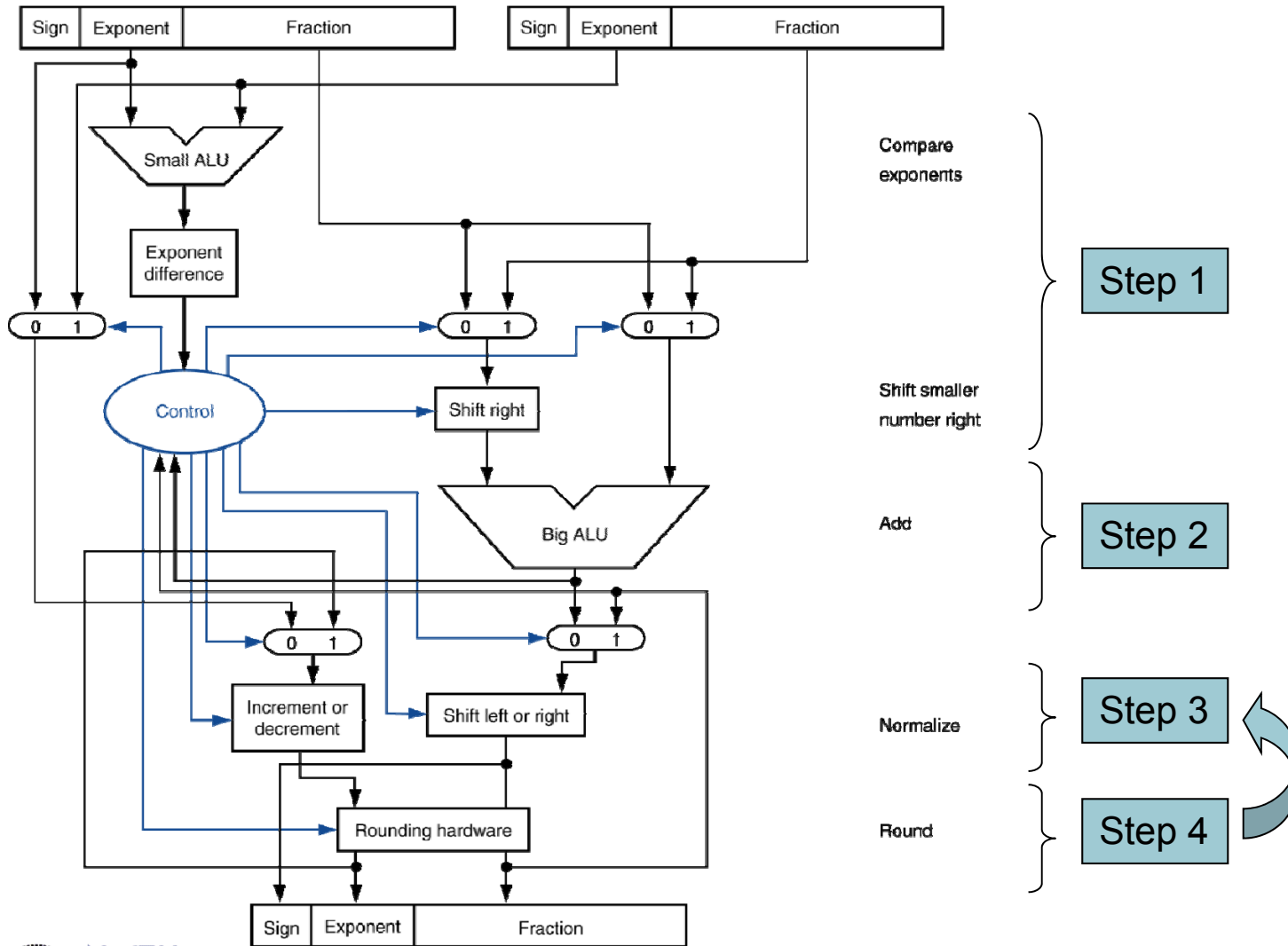
continue until MSB of data is 1

(NOTE: Hidden bit in IEEE Standard)

(5) if result is 0 mantissa, set the exponent



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
 - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
- 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: $+ve \times -ve \Rightarrow -ve$
 - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- Much more complex than integer arithmetic
- Doing it in one clock cycle would take too long
- FP multiplier is of similar complexity to FP adder
 - But uses a multiplier for significand instead of an adder
- FP arithmetic hardware usually does
 - Addition, subtraction, multiplication, division, reciprocal, square-root
 - FP \leftrightarrow 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`, `sdcl`
 - 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`
- more examples,
please refer to Fig. 3.17, p. 200**

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {  
    return ((5.0/9.0)*(fahr - 32.0));  
}
```

- fahr in \$f12, result in \$f0, literals in global memory space

- Compiled MIPS code:

```
f2c: lwc1    $f16, const5($gp)    # $f16=5.0(in Mem.)  
     lwc1    $f18, const9($gp)    # $f18=9.0(in Mem.)  
     div.s   $f16, $f16, $f18     # $f16=5.0/9.0  
     lwc1    $f18, const32($gp)   # $f18=32.0(in Mem)  
     sub.s   $f18, $f12, $f18     # f18=fahr-32.0  
     mul.s   $f0, $f16, $f18     # $f0=(5/9)*(fahr-32)  
     jr     $ra
```

FP Example: Array Multiplication

- $X = X + Y \times Z$
 - All 32×32 matrices, 64-bit double-precision elements

- C code:

```
void mm (double x[][],
         double y[][], double z[][]) {
    int i, j, k;
    for (i = 0; i != 32; i = i + 1)
        for (j = 0; j != 32; j = j + 1)
            for (k = 0; k != 32; k = k + 1)
                x[i][j] = x[i][j]
                    + y[i][k] * z[k][j];
}
```

- Addresses of x, y, z in \$a0, \$a1, \$a2, and
i, j, k in \$s0, \$s1, \$s2

FP Example: Array Multiplication

■ MIPS code:

```
li    $t1, 32      # $t1 = 32 (row size/loop end)
li    $s0, 0       # i = 0; initialize 1st for loop
L1:   li    $s1, 0  # j = 0; restart 2nd for loop
L2:   li    $s2, 0  # k = 0; restart 3rd for loop
-----
sll   $t2, $s0, 5   # $t2 = i * 32 (size of row of x)
addu  $t2, $t2, $s1 # $t2 = i * size(row) + j
sll   $t2, $t2, 3   # $t2 = byte offset of [i][j]
addu  $t2, $a0, $t2 # $t2 = byte address of x[i][j]
l.d   $f4, 0($t2)  # $f4 = 8 bytes of x[i][j]
-----
L3:   sll   $t0, $s2, 5 # $t0 = k * 32 (size of row of z)
addu  $t0, $t0, $s1 # $t0 = k * size(row) + j
sll   $t0, $t0, 3   # $t0 = byte offset of [k][j]
addu  $t0, $a2, $t0 # $t0 = byte address of z[k][j]
l.d   $f16, 0($t0) # $f16 = 8 bytes of z[k][j]
```

...

FP Example: Array Multiplication

...

sl l	\$t0, \$s0, 5	# \$t0 = i * 32 (size of row of y)
addu	\$t0, \$t0, \$s2	# \$t0 = i * size(row) + k
sl l	\$t0, \$t0, 3	# \$t0 = byte offset of [i][k]
addu	\$t0, \$a1, \$t0	# \$t0 = byte address of y[i][k]
l . d	\$f18, 0(\$t0)	# \$f18 = 8 bytes of y[i][k]
mul . d	\$f16, \$f18, \$f16	# \$f16 = y[i][k] * z[k][j]
add . d	\$f4, \$f4, \$f16	# f4 = x[i][j] + y[i][k] * z[k][j]
addi u	\$s2, \$s2, 1	# \$k = k + 1
bne	\$s2, \$t1, L3	# if (k != 32) go to L3
s . d	\$f4, 0(\$t2)	# x[i][j] = \$f4
addi u	\$s1, \$s1, 1	# \$j = j + 1
bne	\$s1, \$t1, L2	# if (j != 32) go to L2
addi u	\$s0, \$s0, 1	# \$i = i + 1
bne	\$s0, \$t1, L1	# if (i != 32) go to L1

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
 - Extra bits of precision (guard, round, sticky)
 - Choice of rounding modes
 - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
 - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

Extra Bits for Rounding

- Why rounding after addition?
 - Because **not every intermediate results is truncated**
 - To keep more precision
- **Guard and round bits:** extra bits to guard against loss of bits during intermediate additions
 - to the right of significand
 - can later be shifted left into significand during normalization
- Sticky bit
 - Additional bit to the right of the round digit
 - Better fine tune rounding

$$\begin{array}{r} b_0 . b_1 b_2 b_3 \dots b_{p-1} 0 0 0 \\ + 0 . 0 0 X \dots X \quad X X S \end{array}$$

← Sticky bit: set to 1 if any 1 bit falls off the end of the round bit

- Get the same results as if the intermediate results were calculated to **infinite precision** and then rounded.

Example

- Try to add 2.98×10^0 and 2.34×10^2
 - only 3 decimal digits are allowed

$$\begin{array}{r} 2.34 \\ + 0.02 \\ \hline 2.36 \end{array} \quad \text{without guard bits}$$

- with 2 more guard bits during computation
- perform rounding at last

$$\begin{array}{r} 2.3400 \\ + 0.0298 \\ \hline 2.3698 \end{array} \quad \rightarrow \text{rounding} \rightarrow 2.37$$

- With guard bits and rounding \rightarrow more accurate results

Interpretation of Data

The BIG Picture

- Bits have no inherent meaning
 - Interpretation depends on the instructions applied
- Computer representations of numbers
 - Finite range and precision
 - Need to account for this in programs

Associativity

- Parallel programs may interleave operations in unexpected orders
 - Assumptions of associativity may fail

		$(x+y)+z$	$x+(y+z)$
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Need to validate parallel programs under varying degrees of parallelism

Subword Parallelism

- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
 - Example: 128-bit adder:
 - 16x8-bit adds; 8x16-bit adds; 4x32-bit adds
- Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD) (sec. 6.6)
 - ARM NEON multimedia instruction extension
 - Intel SSE, SSE2 FP instructions

ARM NEON Instructions

- NEON supports all the subword data type you can imagine except 64-bit FP numbers
 - 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers
 - 32-bit FP numbers

Data transfer	Arithmetic	Logical/Compare
VLDR.F32	VADD.F32, VADD{L,W}{S8,U8,S16,U16,S32,U32}	VAND.64, VAND.128
VSTR.F32	VSUB.F32, VSUB{L,W}{S8,U8,S16,U16,S32,U32}	VORR.64, VORR.128
VLD{1,2,3,4}.{I8,I16,I32}	VMUL.F32, VMULL{S8,U8,S16,U16,S32,U32}	VEOR.64, VEOR.128
VST{1,2,3,4}.{I8,I16,I32}	VMLA.F32, VMLAL{S8,U8,S16,U16,S32,U32}	VBIC.64, VBIC.128
VMOV.{I8,I16,I32,F32}, #imm	VMLS.F32, VMLSL{S8,U8,S16,U16,S32,U32}	VORN.64, VORN.128
VMVN.{I8,I16,I32,F32}, #imm	VMAX.{S8,U8,S16,U16,S32,U32,F32}	VCEQ.{I8,I16,I32,F32}
VMOV.{I64,I128}	VMIN.{S8,U8,S16,U16,S32,U32,F32}	VCGE.{S8,U8,S16,U16,S32,U32,F32}
VMVN.{I64,I128}	VABS.{S8,S16,S32,F32}	VCGT.{S8,U8,S16,U16,S32,U32,F32}
	VNEG.{S8,S16,S32,F32}	VCLE.{S8,U8,S16,U16,S32,U32,F32}
	VSHL.{S8,U8,S16,U16,S32,S64,U64}	VCLT.{S8,U8,S16,U16,S32,U32,F32}
	VSHR.{S8,U8,S16,U16,S32,S64,U64}	VTST.{I8,I16,I32}

x86 FP Architecture

- Originally based on 8087 FP coprocessor
 - 8×80 -bit extended-precision registers
 - Used as a push-down stack
 - Registers indexed from TOS: ST(0), ST(1), ...
- FP values are 32-bit or 64 in memory
 - Converted on load/store of memory operand
 - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
 - Result: poor FP performance

x86 FP Instructions

Data transfer	Arithmetic	Compare	Transcendental
FI LD mem/ST(i)	FI ADDP mem/ST(i)	FI COMP	FPATAN
FI STP mem/ST(i)	FI SUBRP mem/ST(i)	FI UCOMP	F2XMI
FLDPI	FI MULP mem/ST(i)	FSTSW AX/mem	FCOS
FLD1	FI DIVRP mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FPSI N
	FRNDI NT		FYL2X

- Optional variations
 - I: integer operand
 - P: pop operand from stack
 - R: reverse operand order
 - But not all combinations allowed

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

Matrix Multiply

■ Unoptimized code:

```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.   for (int i = 0; i < n; ++i)
4.     for (int j = 0; j < n; ++j)
5.       {
6.         double cij = C[i+j*n]; /* cij = C[i][j] */
7.         for(int k = 0; k < n; k++ )
8.           cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.         C[i+j*n] = cij; /* C[i][j] = cij */
10.      }
11. }
```

Matrix Multiply

■ x86 assembly code:

```
1. vmovsd (%r10),%xmm0 # Load 1 element of C into %xmm0
2. mov %rsi,%rcx # register %rcx = %rsi
3. xor %eax,%eax # register %eax = 0
4. vmovsd (%rcx),%xmm1 # Load 1 element of B into %xmm1
5. add %r9,%rcx # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1,
   element of A
7. add $0x1,%rax # register %rax = %rax + 1
8. cmp %eax,%edi # compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
10. jg 30 <dgemm+0x30> # jump if %eax > %edi
11. add $0x1,%r11d # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10) # Store %xmm0 into C element
```

Matrix Multiply

■ Optimized C code:

```

1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.   for ( int i = 0; i < n; i+=4 )
5.     for ( int j = 0; j < n; j++ ) {
6.       __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j]
          */
7.       for( int k = 0; k < n; k++ )
8.         c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.           _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.            _mm256_broadcast_sd(B+k+j*n)));
11.       _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.     }
13. }

```

Matrix Multiply

■ Optimized x86 assembly code:

```

1. vmovapd (%r11),%ymm0      # Load 4 elements of C into %ymm0
2. mov %rbx,%rcx             # register %rcx = %rbx
3. xor %eax,%eax             # register %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
5. add $0x8,%rax             # register %rax = %rax + 8
6. vmulpd (%rcx),%ymm1,%ymm1 # Parallel mul %ymm1,4 A elements
7. add %r9,%rcx              # register %rcx = %rcx + %r9
8. cmp %r10,%rax             # compare %r10 to %rax
9. vaddpd %ymm1,%ymm0,%ymm0  # Parallel add %ymm1, %ymm0
10. jne 50 <dgemm+0x50>      # jump if not %r10 != %rax
11. add $0x1,%esi            # register % esi = % esi + 1
12. vmovapd %ymm0,(%r11)    # Store %ymm0 into 4 C elements

```


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., $-5 / 4$
 - $11111011_2 \gg 2 = 11111110_2 = -2$
 - Rounds toward $-\infty$
 - c.f. $11111011_2 \ggg 2 = 00111110_2 = +62$

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

Rounding Methods

- Round to zero or Truncation
 - The result closest to zero is returned.
 - Nothing is added to the least significant bit.
- Round up
 - The more positive result closest to the infinitely precise result is returned.
 - If the result is positive and either the guard or the sticky bit is 1, the result is rounded.
 - If the result is negative, the result is not rounded because the unrounded result is the most positive result that is closest to the infinitely precise result.
- Round down
 - The more negative result is returned.
- Round to nearest