Integer Multiplication
Integer Division
Floating Point Numbers

Overview

Multiplying Hardware & Software

Dividing Hardware & Software

Introduction to Floating Point

Doing Floating Point Arithmetic

MIPS Floating Point Instructions

The Dangers of Floating Point

MULTIPLY

- **Paper and pencil example (unsigned):**

```
        1000  Multiplicand U
        1001  Multiplier   M
        1000
       0000
      0000
  × 1000
   01001000  Product       P
```

- **Binary multiplication is easy:**
  - $P_i == 0 \Rightarrow$ place all 0's (0 × multiplicand)
  - $P_i == 1 \Rightarrow$ place a copy of U (1 × multiplicand)
  - Shift the multiplicand left before adding to product
  - *Could we multiply via `add`, `sll`?*

Multiply by Power of 2 via Shift Left

- **Number representation: $B = b_{31}b_{30} \bullet\bullet\bullet\ b_2b_1b_0$**

$$B = b_{31}\times 2^{31} + b_{30}\times 2^{30} + \cdots + b_2\times 2^2 + b_1\times 2^1 + b_0\times 2^0$$

- **What if multiply *B by* 2?**

$$B\times 2 = b_{31}\times 2^{31+1} + b_{30}\times 2^{30+1} + \cdots + b_2\times 2^{2+1} + b_1\times 2^{1+1} + b_0\times 2$$

$$= b_{31}\times 2^{32} + b_{30}\times 2^{31} + \cdots + b_2\times 2^3 + b_1\times 2^2 + b_0\times 2^1$$

- **What if shift *B left* by 1?**

$$B \ll 2 = b_{30}\times 2^{31} + b_{29}\times 2^{30} + \cdots + b_2\times 2^3 + b_1\times 2^2 + b_0\times 2^1$$

- **Multiply by 2^i often replaced by shift left *i***

Multiply in MIPS

M
I
P
S

- **Can multiply variable by any constant using MIPS `sll` and `add` instructions:**

```
i' = i * 10;   /* assume i: $s0 */

sll $t0, $s0, 3          # i * 2^3
add $t1, $zero, $t0
sll $t0, $s0, 1          # i * 2^1
add $s0, $t1, $t0
```

- **MIPS multiply instructions: `mult`, `multu`**

- **`mult $t0, $t1`**
  - puts 64-bit product in pair of new registers `hi`, `lo`; copy to $n by `mfhi`, `mflo`
  - 32-bit integer result in register `lo`

Is Shift Right Arith. ≡ Divide by 2?

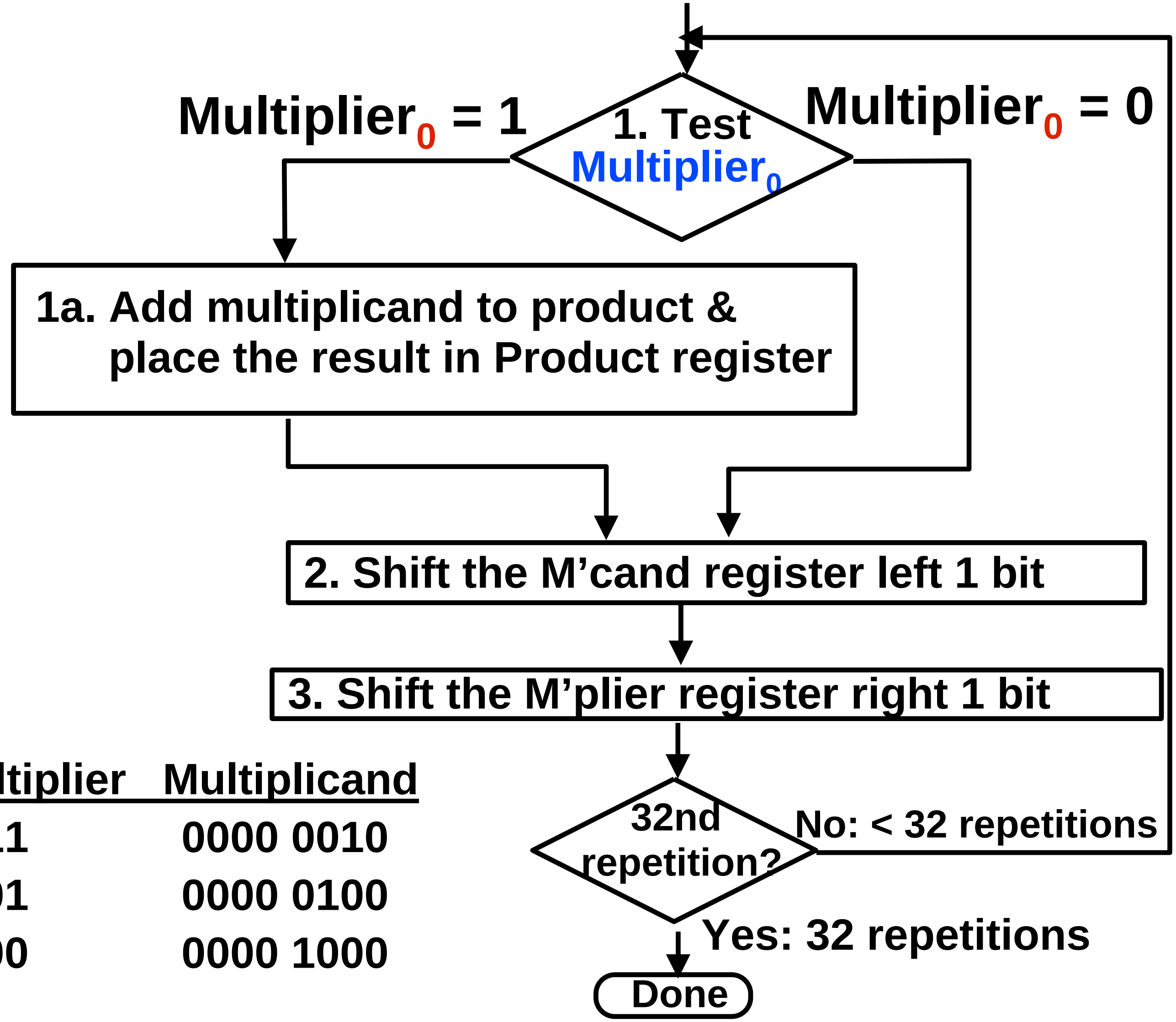
- **Shifting right by *n* bits would seem to be the same as dividing by 2^n**
- **Problem is signed integers**
  - Zero fill (`srl`) is wrong for negative numbers
- **Shift Right Arithmetic (`sra`); sign extends (replicates sign bit); but does it work?**
- **Divide -5 by 4 via `sra 2`; result should be -1**

  1111 1111 1111 1111 1111 1111 1111 1011

  1111 1111 1111 1111 1111 1111 1111 1110
- **= -2, not -1; Off by 1, so doesn't work**
- **Is it always off by 1??**

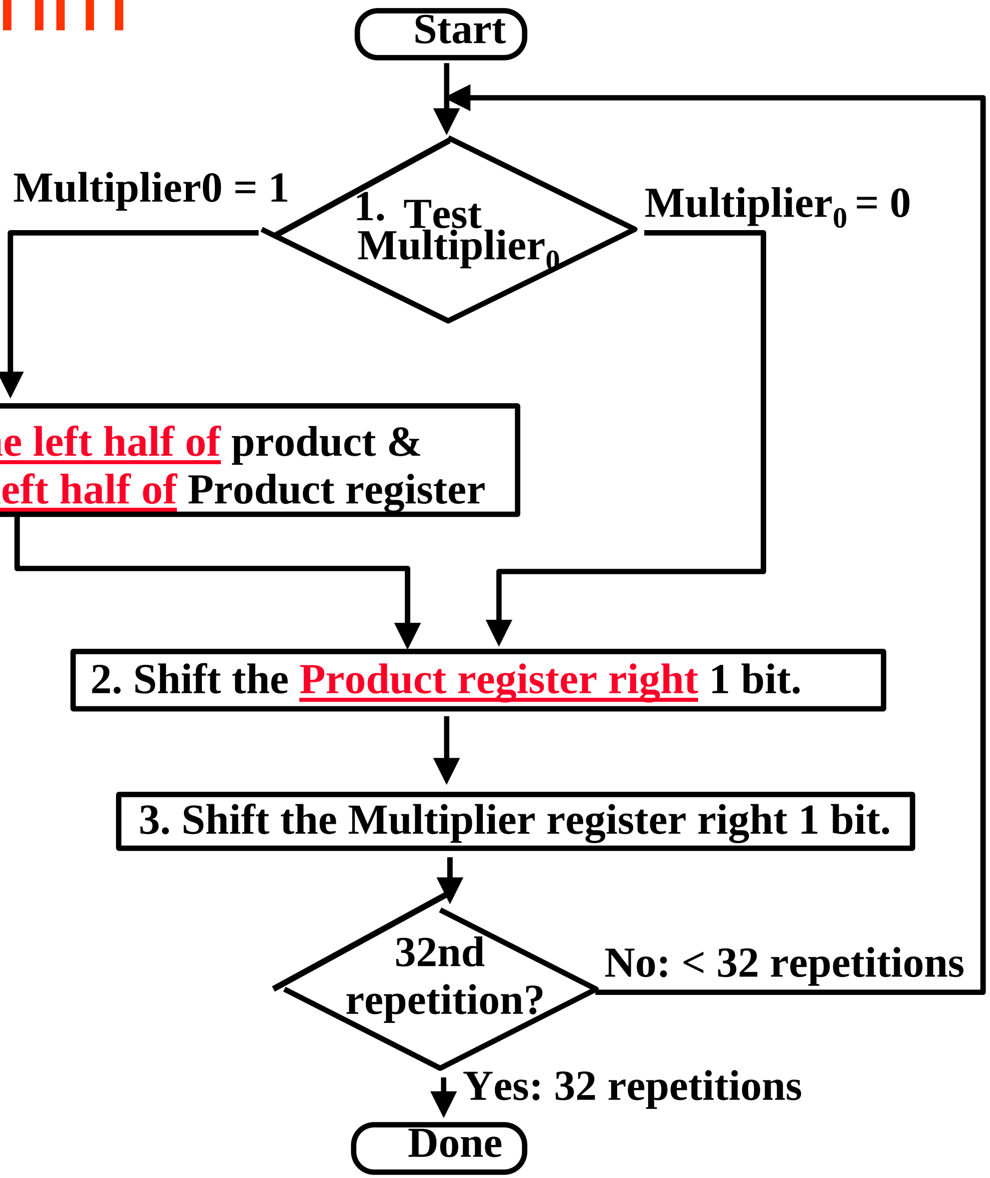
Multiply Algorithm *Version* 1

```
    0010
  x 0011
--------
00000110
```



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

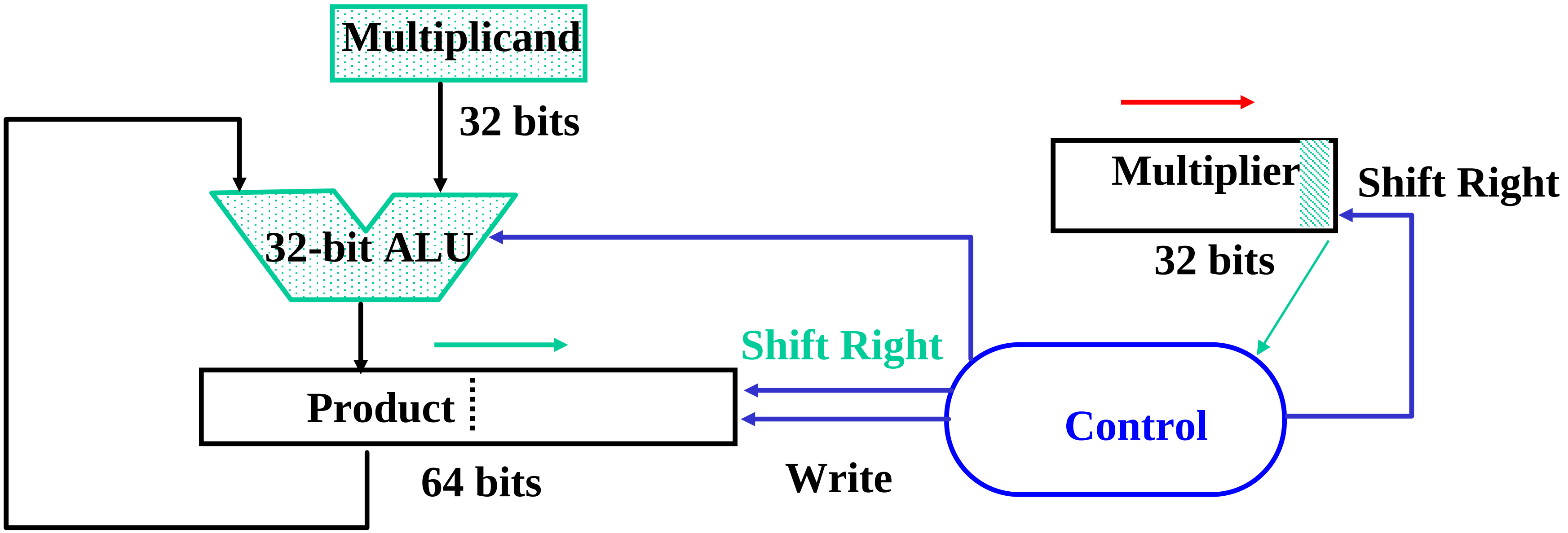
Multiply Algorithm Version 2



	Product	Multiplier	Multiplicand
	0000 0000	0011	0010
1:	0010 0000	0011	0010
2:	0001 0000	0011	0010
3:	0001 0000	0001	0010
1:	0011 0000	0001	0010
2:	0001 1000	0001	0010
3:	0001 1000	0000	0010
1:	0001 1000	0000	0010
2:	0000 1100	0000	0010
3:	0000 1100	0000	0010
1:	0000 1100	0000	0010
2:	0000 0110	0000	0010
3:	0000 0110	0000	0010
	0000 0110	0000	0010

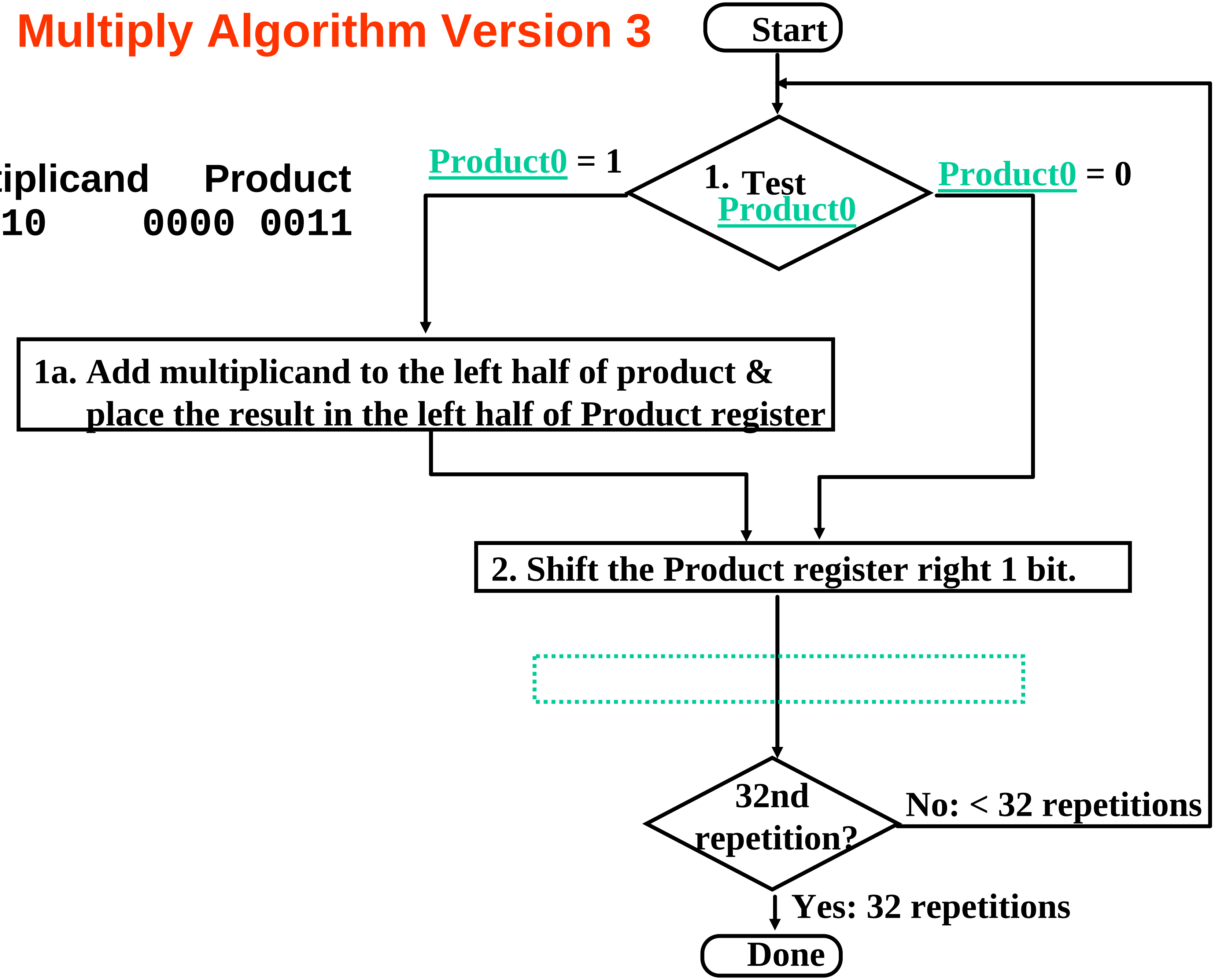
MULTIPLY HARDWARE Version 2

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



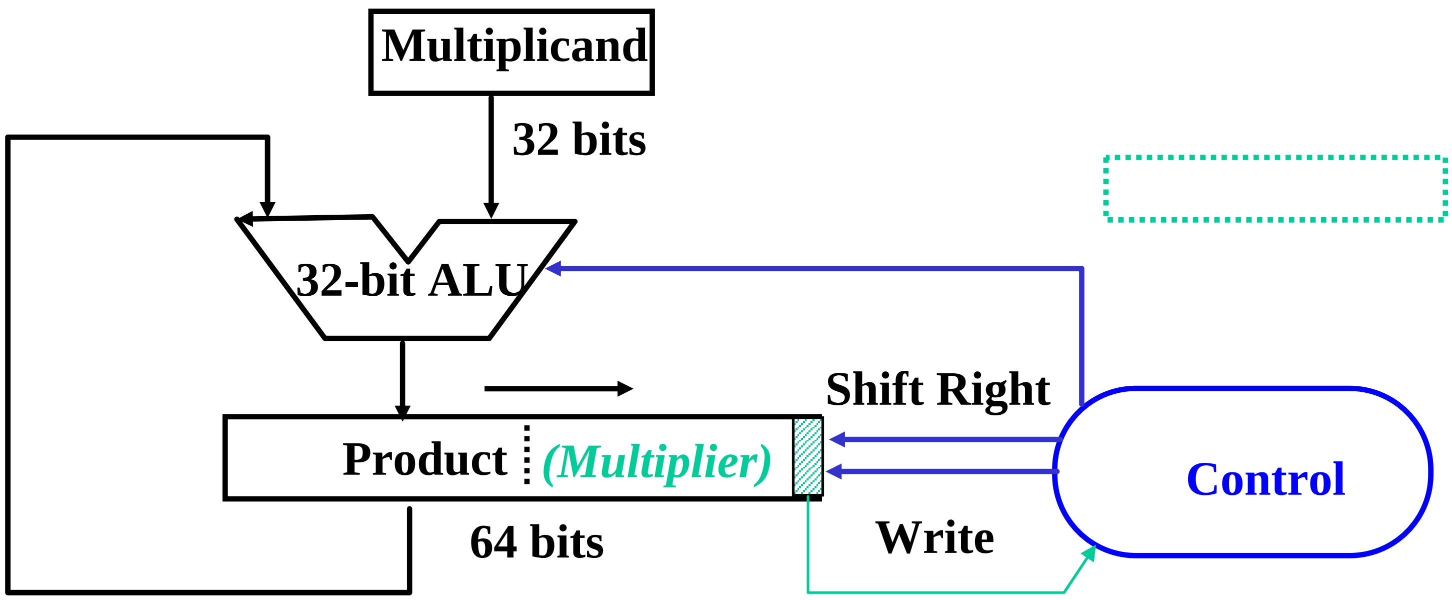
Multiply Algorithm Version 3

Multiplicand	Product
0010	0000 0011



MULTIPLY HARDWARE Version 3

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



Observations on Multiply Version 3

- **2 steps per bit because Multiplier & Product combined**
- **MIPS registers Hi and Lo are left and right half of Product**
- **Gives us MIPS instruction MultU**
- **How can you make it faster?**
- **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
    - can handle multiple bits at a time

Motivation for Booth's Algorithm

- Example 2 x 6 = 0010 x 0110:

```
          0010
x         0110
--------------
+         0000     shift (0 in multiplier)
+        0010      add (1 in multiplier)
+       0100       add (1 in multiplier)
--------------
+      0000        shift (0 in multiplier)
     00001100
```

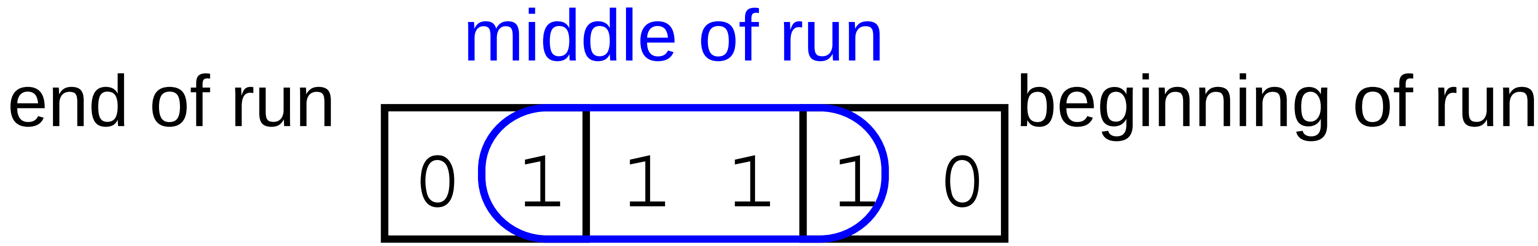
- ALU with add or subtract gets same result in more than one way:

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

- For example
-

```
              0010
x             0110
------------------
              0000  shift (0 in multiplier)
–            0010   sub (first 1 in multpl.)
            0000    shift (mid string of 1s)
+          0010     add (prior step had last 1)
------------------
          00001100
```

Booth's Algorithm



Current Bit	Bit to the Right	Explanation	Example	Op
1	0	Begins run of 1s	0001111000	sub
1	1	Middle of run of 1s	0001111000	none
0	1	End of run of 1s	0001111000	add
0	0	Middle of run of 0s	0001111000	none

Originally for Speed (when shift was faster than add)

- **Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one**

```
     –1
+ 10000
  01111
```

Booths Example (2 x 7)

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 0111 0	10 -> sub
1a. P = P - m	1110	+ 1110	
		1110 0111 0	shift P (sign ext)
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

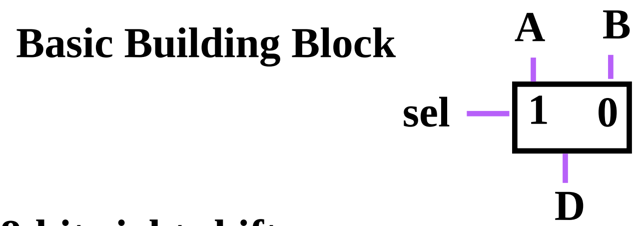
Booths Example (2 x -3)

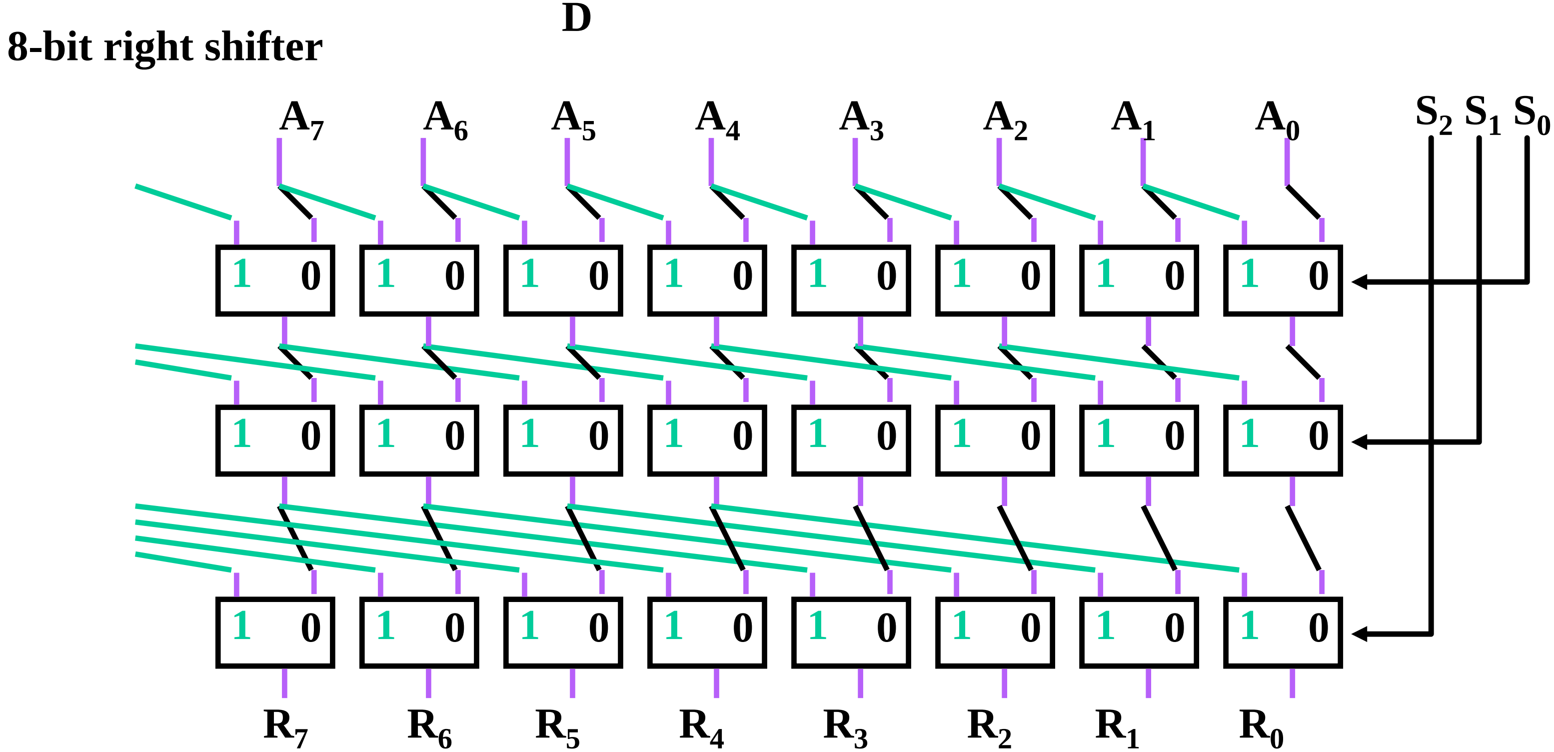
Operation	Multiplicand	Product	next?
0. initial value	0010	0000 1101 0	10 -> sub
1a. P = P - m	1110	+ 1110	
		1110 1101 0	shift P (sign ext)
1b.	0010	1111 0110 1	01 -> add
		+ 0010	
2a.		0001 0110 1	shift P
2b.	0010	0000 1011 0	10 -> sub
		+ 1110	
3a.	0010	1110 1011 0	shift
3b.	0010	1111 0101 1	11 -> nop
4a		1111 0101 1	shift
4b.	0010	1111 1010 1	done

MIPS logical instructions

Instruction	*Example*	*Meaning*	*Comment*
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, constant
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, constant
xor immediate	xori $1, $2,10	$1 = ~$2 &~10	Logical XOR reg, constant
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by variable

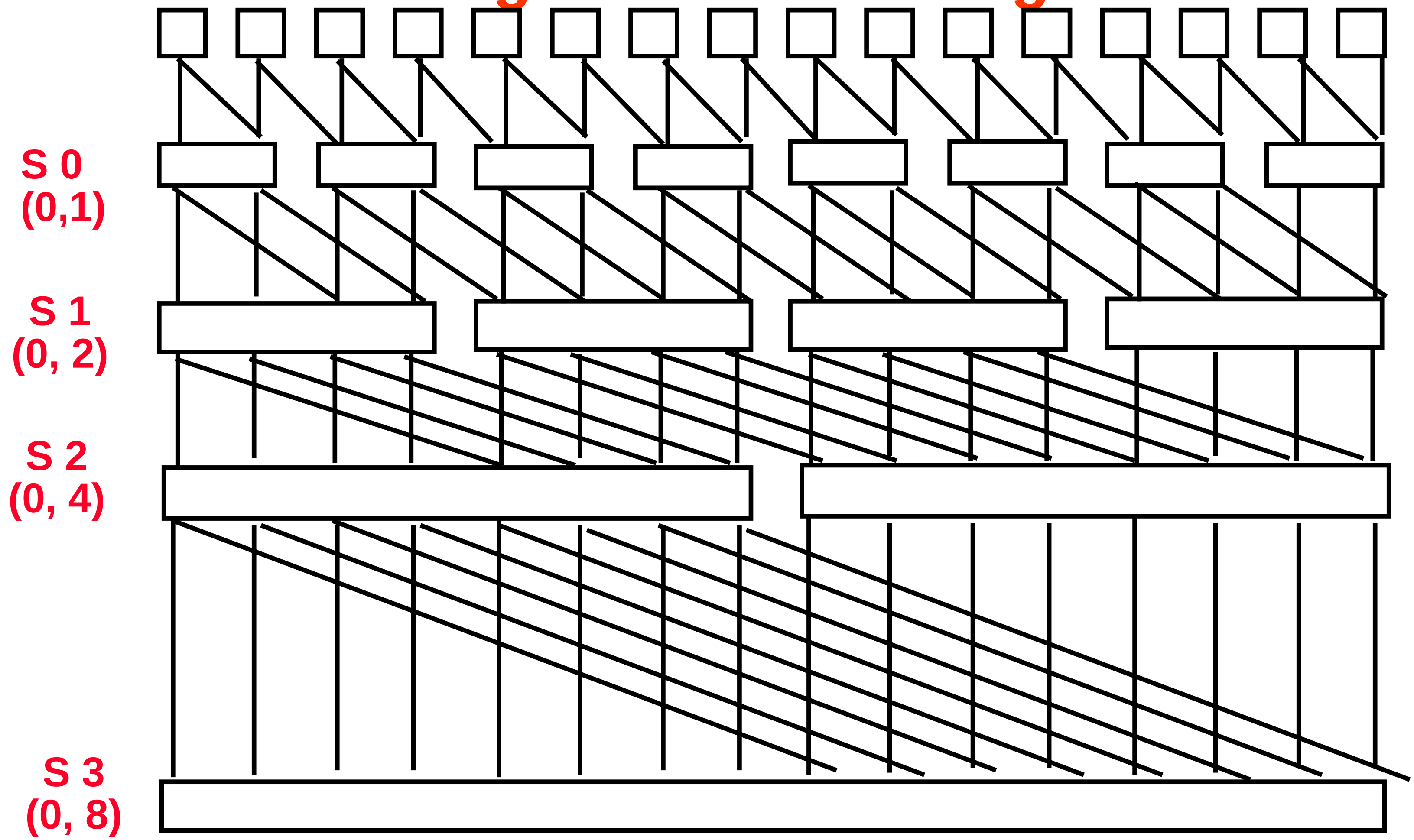
Combinational Shifter from MUXes





- What comes in the MSBs?
- How many levels for 32-bit shifter?
- What if we use 4-1 Muxes ?

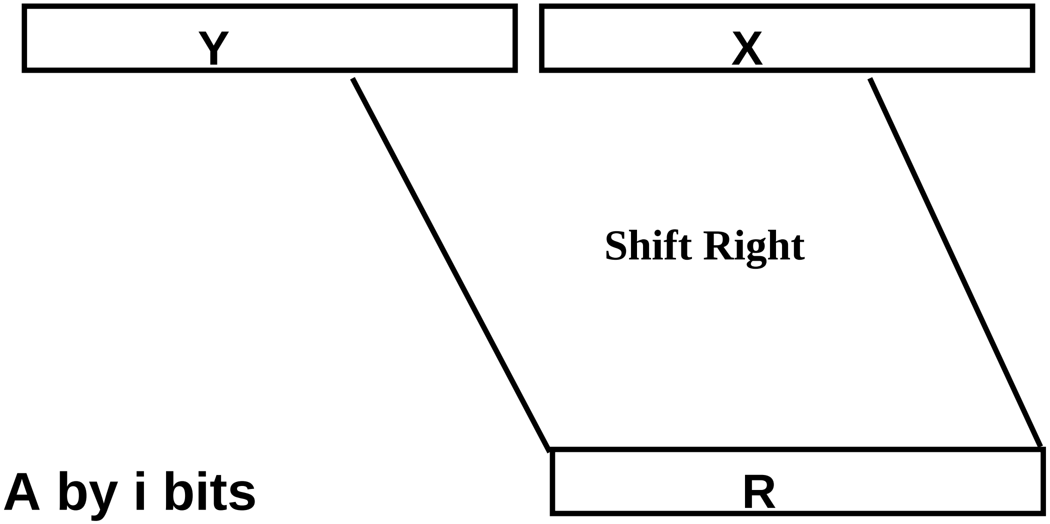
General Shift Right Scheme using 16 bit example



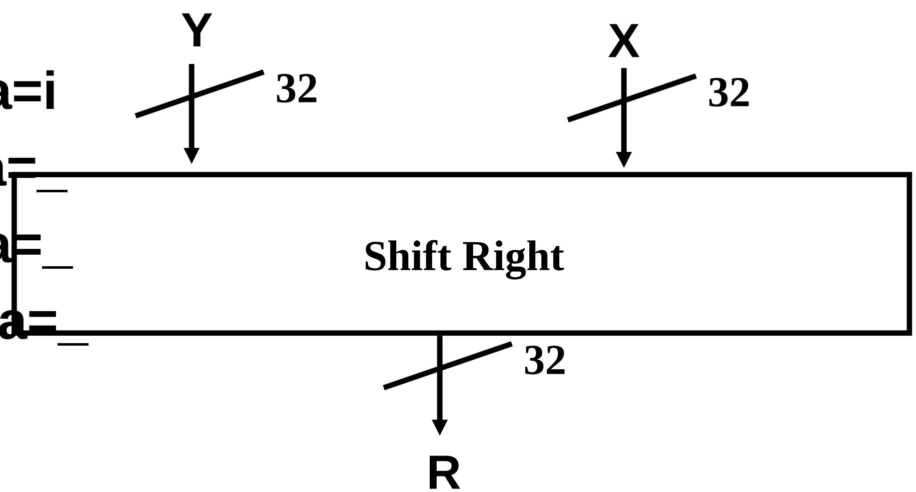
If added Right-to-left connections could support Rotate (not in MIPS but found in ISAs)

Funnel Shifter

Instead Extract 32 bits of 64.

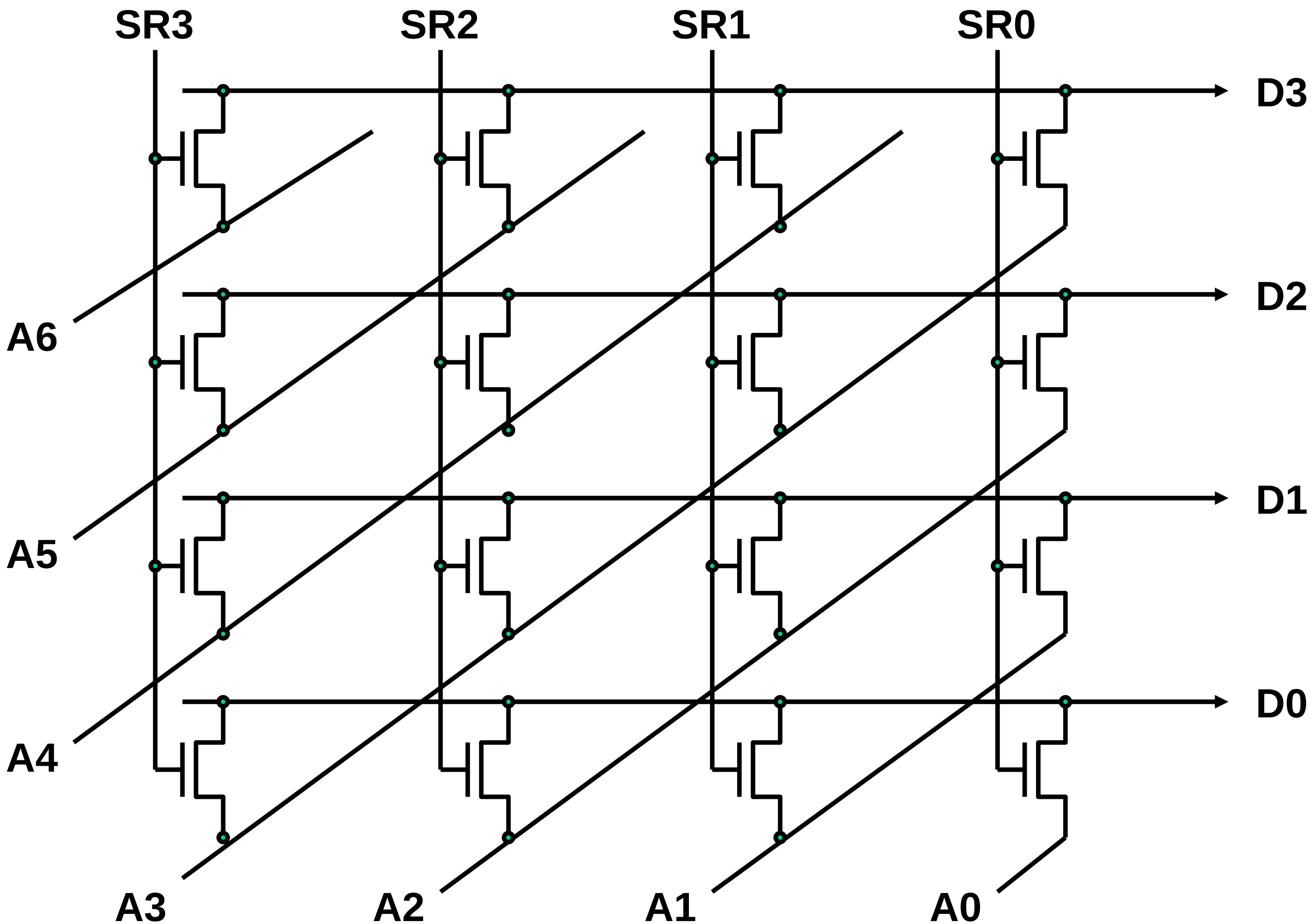


- **Shift A by i bits**
  (sa= shift right amount)
- **Logical: Y = 0, X=A, sa=i**
- **Arithmetic? Y = _, X=_, sa=_**
- **Rotate? Y = _, X=_, sa=_**
- **Left shifts? Y = _, X=_, sa=_**



Barrel Shifter

Technology-dependent solutions: transistor per switch



Divide: Paper & Pencil

```
                    1001        Quotient
Divisor 1000    ⎡ 1001010        Dividend
                   –1000
                      10
                      101
                      1010
                     –1000
                        10      Remainder (or Modulo result)
```

See how big a number can be subtracted, creating quotient bit on each step

Binary => 1 * divisor or 0 * divisor

Dividend = Quotient x Divisor + Remainder

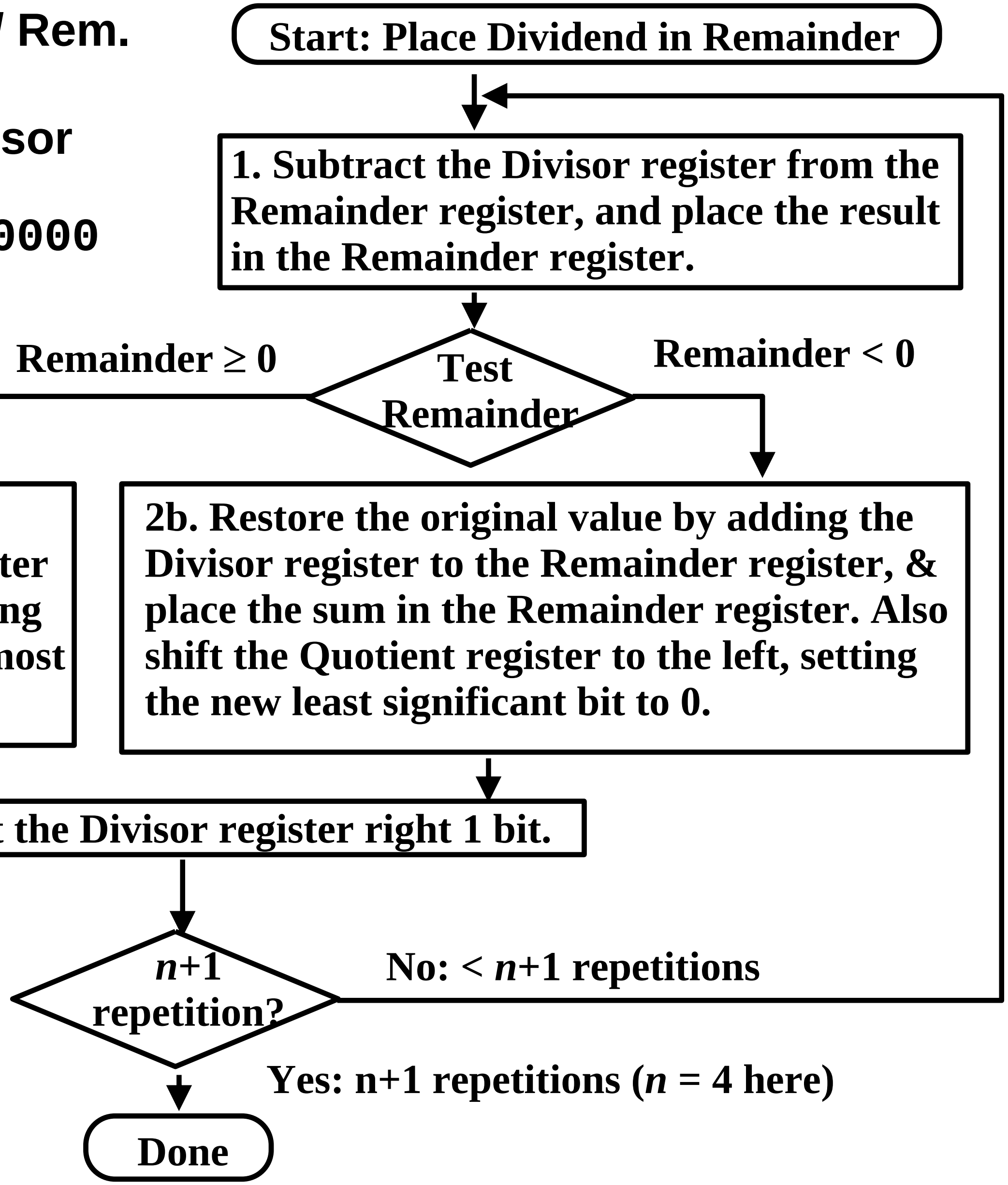
=> | Dividend | = | Quotient | + | Divisor |

3 versions of divide, successive refinement

Divide Algorithm

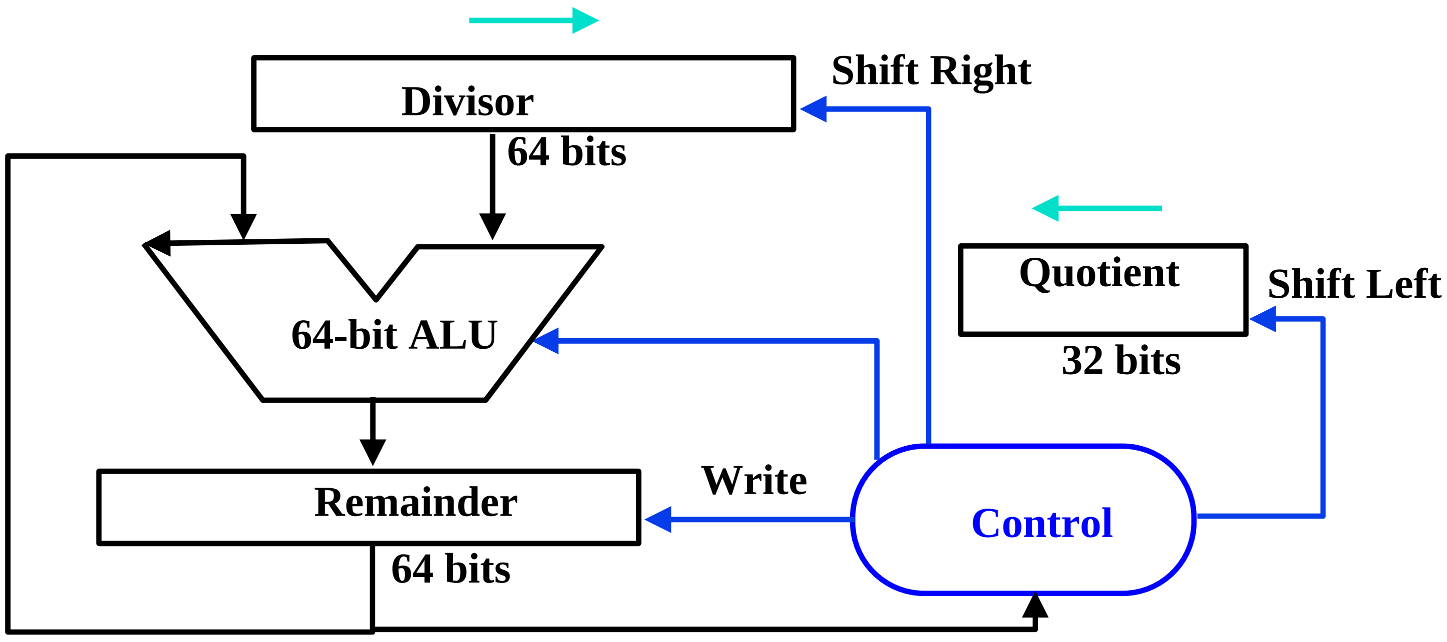
- Takes *n*+1 steps for *n*-bit Quotient / Rem.

Remainder	Quotient	Divisor
0000 0111	00000	0010 0000



Integer Division

- ALU, Divisor, and Remainder registers: 64bit;
- Quotient register: 32 bits;
- 32 bit *divisor* starts in left ½ of Divisor reg. and it is shifted right 1 on each step
- Remainder register initialized with *dividend*



Divide Algorithm Example

	Remainder	Quotient	Divisor
	0000 0111	00000	0010 0000
1:	1110 0111	00000	0010 0000
2:	0000 0111	00000	0010 0000
3:	0000 0111	00000	0001 0000
1:	1111 0111	00000	0001 0000
2:	0000 0111	00000	0001 0000
3:	0000 0111	00000	0000 1000
1:	1111 1111	00000	0000 1000
2:	0000 0111	00000	0000 1000
3:	0000 0111	00000	0000 0100
1:	0000 0011	00000	0000 0100
2:	0000 0011	00001	0000 0100
3:	0000 0011	00001	0000 0010
1:	0000 0001	00001	0000 0010
2:	0000 0001	00011	0000 0010
3:	0000 0001	00011	0000 0010

Answer:
Quotient = 3
Remainder = 1

Divide Algorithm

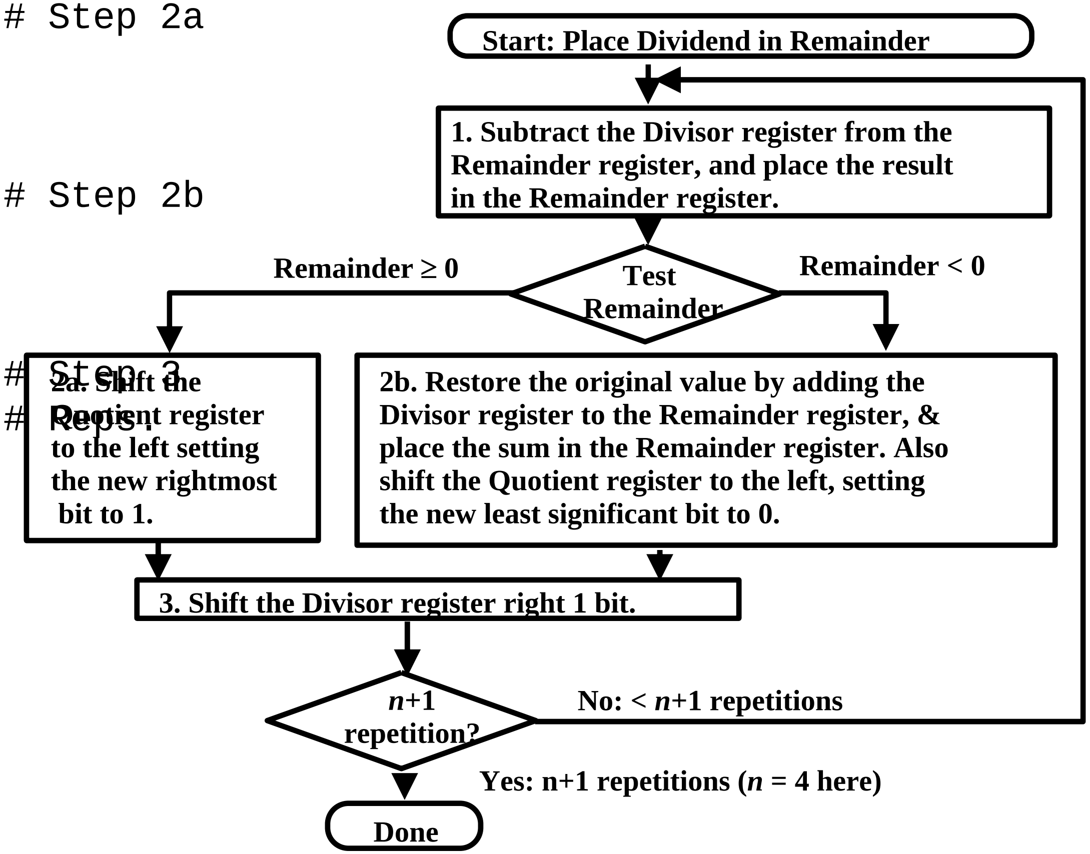
```
Let $s0 = Dividend,
    $s1 = Divisor,
    $s2 = Remainder,
    $s3 = Quotient,
    $s4 = Repetitions
Start:
        move    $s2, $s0
Loop:
        sub     $s2, $s2, $s1   # Step 1
        bltz    $s2, Label2b
        sll     $s3, $s3, 1      # Step 2a
        ori     $s3, $s3, 1
        j       Label3
Label2b:
        add     $s2, $s2, $s1    # Step 2b

        sll     $s3, $s3, 1
Label3:
        slr     $s1, $s1, 1      # Step 3
        addi    $s4, $s4, -1     # Reps.
        Bgtz    $s4, Loop
```

Quotient = 0; 32 bit *divisor* starts in left ½ of Divisor reg. and it is shifted right 1 on each step; Remainder = *dividend*;

If Remainder < 0, we need to add Divisor back to *dividend*; else 1 is generated for Quotient;

Shift Divisor right 1 bit;

Repeat 33 times



What is in a number?

- **What can be represented in N bits?**
- **Unsigned 0 to $2^N - 1$**
- **2s Complement -2^{N-1} to $2^{N-1} - 1$**
- **1s Complement $-2^{N-1}+1$ to $2^{N-1}-1$**
- **Excess M 2^{-M} to 2^{N-M-1}**
  - (E = e + M)
- **BCD 0 to $10^{N/4} - 1$**
- **But, what about?**
  - very large numbers? 9,349,398,989,787,762,244,859,087,678
  - very small number? 0.0000000000000000000000045691
  - rationals 2/3
  - irrationals $\sqrt{2}$
  - transcendentals e,

Recall Scientific Notation

(sign, magnitude) **(sign, magnitude)**

Mantissa ***exponent***

$$\mathbf{6.02 \times 10^{23}}$$

decimal point ***radix (base)***

- **Normal form:**

no leading 0s (digit 1 to left of decimal point)

- **Alternatives to representing 1/1,000,000,000**

Normalized:	1.0×10^{-9}
Not normalized:	0.1×10^{-8}, 10.0×10^{-10}

Scientific Notation for Binary Numbers

(sign, magnitude)
Mantissa

(sign, magnitude)
exponent

$$\mathbf{1.0 \times 2^{-1}}$$

radix (base)

binary point

- **Computer arithmetic that supports it called floating point, because it represents numbers where binary point is not fixed, as it is for integers**

- **Declare such a variable in C as `float (double, long double)`**

- **Normalized form: $1.xxxxxxxxxx_2 \times 2^{yyyy_2}$**

  Simplifies data exchange, increases accuracy

  4_{10} == 1.0×2^2, 8_{10} == 1.0×2^3

Single Precision FP Representation

- **Start with a single word (32-bits)**

31	30 23	22 0
S	**Exponent**	**Significand**
1 bit	**8 bits**	**23 bits**

- **Meaning: $(-1)^S$ x Mantissa x 2^E**

- **Can now represent numbers as small as 2.0×10^{-38} to as large as 2.0×10^{38}**

- **Relationship between Mantissa and Significand bits? Between E and Exponent?**

- **In C type `float`**

Floating Point Number Representation

- **What if result too large? (> 2.0×10^{38})**

  Overflow!

  Overflow ⇔ Exponent larger than can be represented in 8-bit Exponent field

- **What if result too small? (>0, < 2.0×10^{-38})**

  Underflow!

  Underflow ⇔ Negative Exponent too small

- **How to reduce chances of overflow or underflow?**

Double Precision FP Representation

- **Next Multiple of Word Size (64 bits)**

31	30 … 20	19 … 0
S	**Exponent**	**Significand**
1 bit	11 bits	20 bits

Significand (cont'd)
32 bits

- **Double Precision (*vs.* Single Precision)**

1. **C variable declared as `double`**
2. **Represent numbers almost as small as 2.0 x 10^{-308} to almost as large as 2.0 x 10^{308}**
3. **But primary advantage greater accuracy due to larger significand**
4. **There is also `long double` version (16 bytes)**

MIPS follows IEEE 754 F.P. Standard

- **To pack more bits, make leading 1 of mantissa implicit for normalized numbers**

  1 + 23 bits single, 1 + 52 bits double

  0 has no leading 1, so reserve exponent value 0 just for number 0.0

  Meaning: (almost correct)

  $$(\mathbf{-1})^{S} \times (\mathbf{1 + Significand}) \times \mathbf{2}^{\mathbf{Exponent}},$$

  where 0 < Significand < 1

- **If label significand bits left-to-right as s_1, s_2, s_3, ... then value is:**

  $$(-1)^{S} \times (1+(s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_2 \times 2^{-3}) + \cdots) \times 2^{\text{Exponent}}$$

Representing Exponent

- **Want to compare Fl. Pt. numbers as if they were integers, to help in sorting**

  Sign first part of number

  Exponent next, so bigger exponent $\Rightarrow$ bigger number

  $1.1 \times 10^{20} > 1.9 \times 10^{10}$

- **What About Negative Exponents?**

  Use 2's comp? 1.0×2^{-1} *vs*. $1.0 \times 2^{+1}$ (1/2 v. 2)

1/2	**0**	**1111 1111**	**000 0000 0000 0000 0000 0000**
2	**0**	**0000 0001**	**000 0000 0000 0000 0000 0000**

This notation using integer compare of

1/2 *vs*. 2 makes 1/2 > 2!

Doesn't work!

Representing Exponent

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

- **Instead, pick notation 0000 0000 as most negative, and 1111 1111 as most positive**

- **1.0 x × 2^{-1} *vs*. 1.0 x ×2^{+1} (1/2 v. 2)**

  - **Called <u>Biased Notation</u>, where bias is number subtracted to get real number**

    IEEE 754 uses bias of 127 for single precision

    Representation (Finally, the truth!):

$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - 127)}$

1023 is bias for double precision

Example: Converting Decimal to FP

- **Show MIPS representation of -0.75 (show exponent in decimal to simplify)**

$-0.75 = -3/4 = -3/2^2$

$-11_{two}/2^2 = -11_{two} \times 2^{-2} = -0.11_{two} \times 2^0$

Normalized to $-1.1_{two} \times 2^{-1}$

$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

$(-1)^1 \times (1 + .100\ 0000\ ...\ 0000) \times 2^{(126-127)}$

1	**0111 1110**	**100 0000 0000 0000 0000 0000**

***S* = 1; Exponent = 126; Significand = 100 ... 000$_2$**

Example: Converting FP to Decimal

- **Sign S = 0 $\Rightarrow$ positive**
- **Exponent E :**

  $0110\ 1000_{two} = 104_{ten}$

  Bias adjustment: 104 - 127 = -13
- **Mantissa:**

  $1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$

  $= 1+ (5{,}587{,}778 / 2^{23})$

  $= 1+ (5{,}587{,}778 / 8{,}388{,}608) = 1.0 + 0.666115$
- **Represents: $1.666115_{ten} \times 2^{-13} \sim 2.034 \times 10^{-4}$**

0	**0110 1000**	**101 0101 0100 0011 0100 0010**

Continuing Example: Binary to ???

- **Convert 2's Complement to Integer:**

$2^{29} + 2^{28} + 2^{26} + 2^{22} + 2^{20} + 2^{18} + 2^{16} + 2^{14} + 2^{9} + 2^{8} + 2^{6} + 2^{1}$

$= 878{,}003{,}010_{ten}$

0011 0100 0101 0101 0100 0011 0100 0010

- **Convert Binary to Instruction:**

0011 01	00 01	01 0101	0100 0011 0100 0010
13	2	21	17218

```
ori $s5, $v0, 17218
```

- **Convert Binary to ASCII:**

0011 0100	0101 0101	0100 0011	0100 0010
4	U	C	B

Principle: Type not associated with Data

- **What does this bit pattern mean:**

0011 0100 0101 0101 0100 0011 0100 0010

$2.034*10^{-4}$? 878,003,010? "4UCB"?

`ori $s5, $v0, 17218?`

- **Data can be anything; operation of instruction that accesses operand determines its type!**

  Side-effect of stored program concept: instructions stored as numbers

- **Power/danger of unrestricted addresses/ pointers: use ASCII as Fl. Pt., instructions as data, integers as instructions, ...**

How to Order Floating Point Fields?

- **"Natural": Sign, Significand, Exponent?**

  Problem: If want to sort using integer operations, won't work:

  1.0 x 2^{20} vs. 1.1 x 2^{10} ; storing significant first makes FP comparisons impossible to carry out correctly via integer comparisons. Second FP looks bigger!

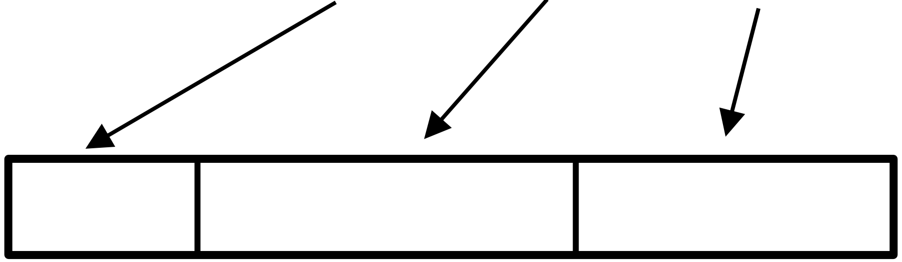
0	**10000**	**10100**

0	**11000**	**01010**

- **Exponent, Sign, Significand?**

  Need to get sign first, since negative < positive

- **Therefore order is Sign Exponent Significant**



How To Convert Decimal to Binary

- **How convert 10.4_{ten} to binary?**
- **Deal with fraction & whole parts separately:**

10 ¸ 2 = 5 remainder 0
5 ¸ 2 = 2 remainder 1
2 ¸ 2 = 1 remainder 0
1 ¸ 2 = 0 remainder 1

.4 x 2 = 0.8
.8 x 2 = 1.6
.6 x 2 = 1.2
.2 x 2 = 0.4
.4 x 2 = 0.8

$$10.4_{ten} = 1010.\overline{0110}_{two}$$

Do It Yourself

- **Convert 10.4_{ten} to single precision floating point**
- **Recall that:**

10.4_{ten} is 1010.0110_{two}

Do It Yourself

(1) Normalize

$1010.0110_{two} \times 2^0 = 1.0100110 \times 2^3$

(2) Determine Sign Bit

positive, so S = 0

(3) Determine Exponent:

2^3 so 3 + bias (= 127) = 130 = 10000010_{two}

(4) Determine Significand

drop leading 1 of mantissa, expand to 23 bits = 01001100110011001100110

0	10000010	01001100110011001100110
S	Exponent	Significand

Example: Converting FP to Decimal

1. **Sign: 0 $\Rightarrow$ positive**

2. **Exponent:**

   $0110\ 1000_2 = 104_{10}$

   Bias adjustment: 104 - 127 = -13

3. **Mantissa:**

   $1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$

   $= 1 + (5{,}587{,}778/2^{23})$

   $= 1 + (5{,}587{,}778/8{,}388{,}608) = 1.0 + 0.666115$

4. **Represents: $1.666115_{ten} * 2^{-13} \sim 2.034*10^{-4}$**

0	**0110 1000**	**101 0101 0100 0011 0100 0010**

Example: Decimal F. P. Addition

- **Assume 4 digit significand, 2 digit exponent**
- **Let's add $9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}$**
- **Exponents must match, so adjust smaller number to match larger exponent**

$$1.610 \times 10^{-1} = 0.1610 \times 10^{0} = 0.01610 \times 10^{1}$$

- **Can represent only 4 digits, so must discard last two:**

$$0.016 \times 10^{1}$$

Example: Decimal F. P. Addition

- **Now, add significands:**

$$\begin{array}{r} 9.999 \\ +\ 0.016 \\ \hline 10.015 \end{array}$$

- **Thus, sum is 10.015×10^1**
- **Sum is not normalized, so correct it, checking for underflow/overflow:**

$$10.015 \times 10^1 \Rightarrow 1.0015 \times 10^2$$

- **Cannot store all digits, must round. Final result is:**

$$\boxed{1.002 \times 10^2}$$

Basic Binary FP Addition Algorithm

For addition (or subtraction) of X to Y ($X < Y$):

1. **Compute $D = \mathrm{Exp}_Y - \mathrm{Exp}_X$ (align binary points)**
2. **Right shift $(1+\mathrm{Sig}_X)$ D bits $\Rightarrow (1+\mathrm{Sig}_X)*2^{-D}$**
3. **Compute $(1+\mathrm{Sig}_X)*2^{-D} + (1+\mathrm{Sig}_Y)$; Normalize if necessary; continue until MS bit is 1**

4. **Too small (*e.g.*, 0.001xx...) left shift result, decrement result exponent; check for underflow**

4'. **Too big (*e.g.*, 10.1xx...)**
right shift result, increment result exponent; check for overflow

5. **If result significand is 0, set exponent to 0**

FP Subtraction

- **Similar to addition**
- **How do we do it?**

  De-normalize to match exponents

  Subtract significands

  Keep the same exponent

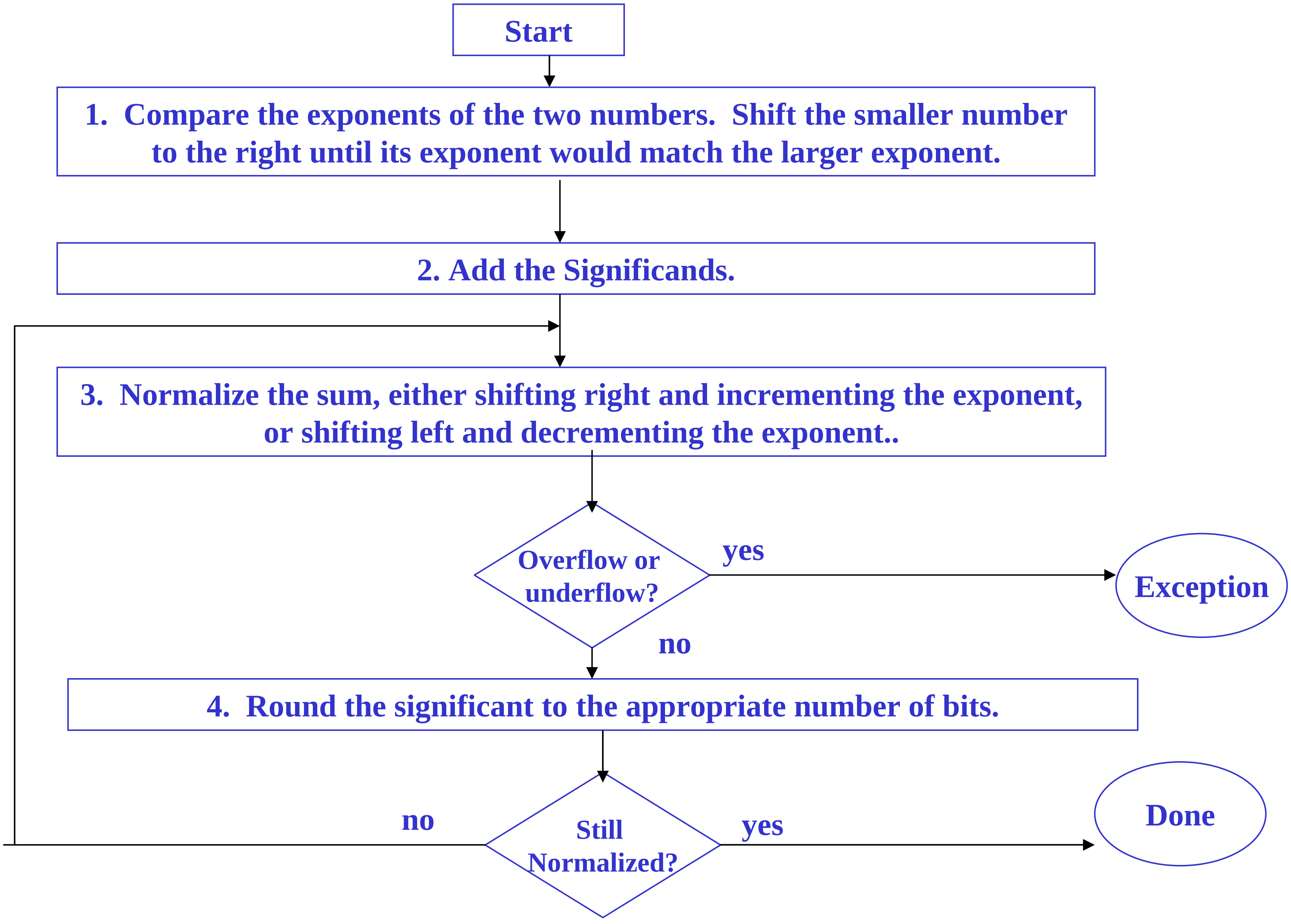
  Normalize (possibly changing exponent)
- **Problems in implementing FP add/sub:**

  Managing the signs,

  determining to add or sub,

  swapping the operands.
- **Question: How do we integrate this into the integer arithmetic unit?**

Floating Point Addition



Example: Decimal F. P. Multiply

- **Let's multiply:**

  $\mathbf{1.110_{ten} \times 10^{10} \times 9.200_{ten} \times 10^{-5}}$

  (Assume 4-digit significand, 2-digit exponent)

- **First, add exponents:**

  $$\begin{array}{r} 10 \\ +\ -5 \\ \hline 5 \end{array}$$

- **Next, multiply significands:**

  1.110 x 9.200 = 10.212000

Example: Decimal F. P. Multiply

- **Product is not normalized, so correct it, checking for underflow / overflow:**

  $10.212000 \times 10^5 \Rightarrow 1.0212 \times 10^6$

- **Significand exceeds 4 digits, so round:**

  1.021×10^6

- **Check signs of original operands**
  - **same $\Rightarrow$ positive**
  - **different $\Rightarrow$ negative**

  Final result is: $\boxed{+1.021 \times 10^6}$

Basic Binary FP Multiplication Algorithm

For multiplication of $P = X \times Y$:

1. **Compute Exponent: $Exp_P = (Exp_Y + Exp_X) - Bias$**

2. **Compute Product: $(1 + Sig_X) \times (1 + Sig_Y)$**
 Normalize if necessary; continue until most significant bit is 1

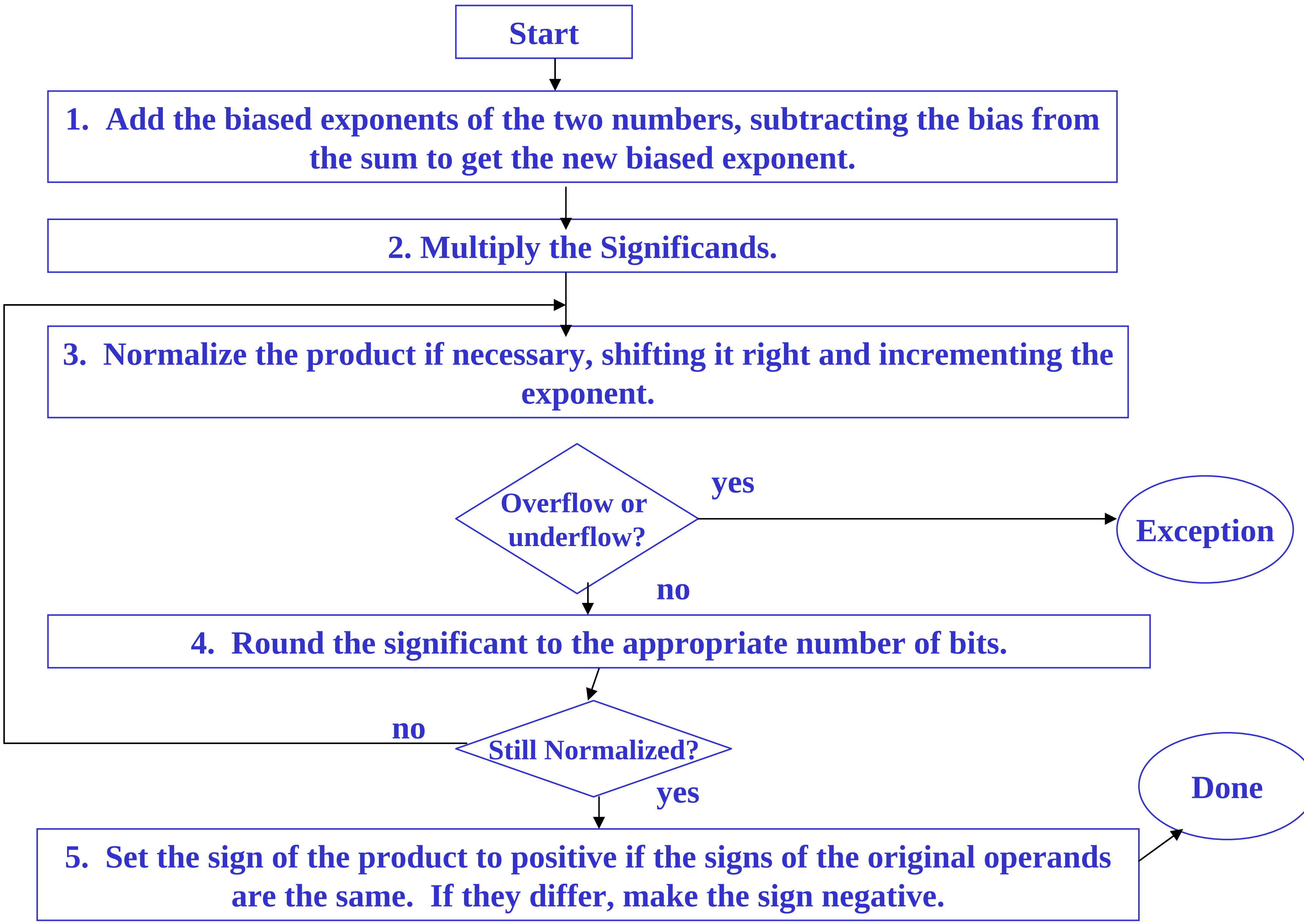
4. **Too small (*e.g.*, 0.001xx...) $\rightarrow$**
 left shift result, decrement result exponent

4'. **Too big (*e.g.*, 10.1xx...) $\rightarrow$**
 right shift result, increment result exponent

5. **If (result significand is 0) then set exponent to 0**

6. **if (Sgn_X == Sgn_Y) then**
 Sgn_P = positive (0)
 else
 Sgn_P = negative (1)

FP Multiplication Algorithm



Representation for Not a Number

- **What do I get if I calculate**

  `sqrt(-4.0)`or

  `0/0?`

- **If infinity is not an error, these shouldn't be either.**

  Called Not a Number (NaN)

  Exponent = 255, Significand nonzero

- **Why is this useful?**

  Hope NaNs help with debugging?

  They contaminate: op(NaN, *X*) = NaN

What else can I put in?

- **What defined so far? (Single Precision)**

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- fl. pt. number
255	0	+/- infinity
255	nonzero	???

- **Representing "Not a Number"; *e.g.,* sqrt(-4); called NaN**

  Exp == 255, Significand nonzero

  They contaminate FP ops: (NaN θ *X*) = NaN

  Hope NaNs help with debugging?

  Only valid operations are ==, !=

What else can I put in?

- **What defined so far? (Single Precision)**

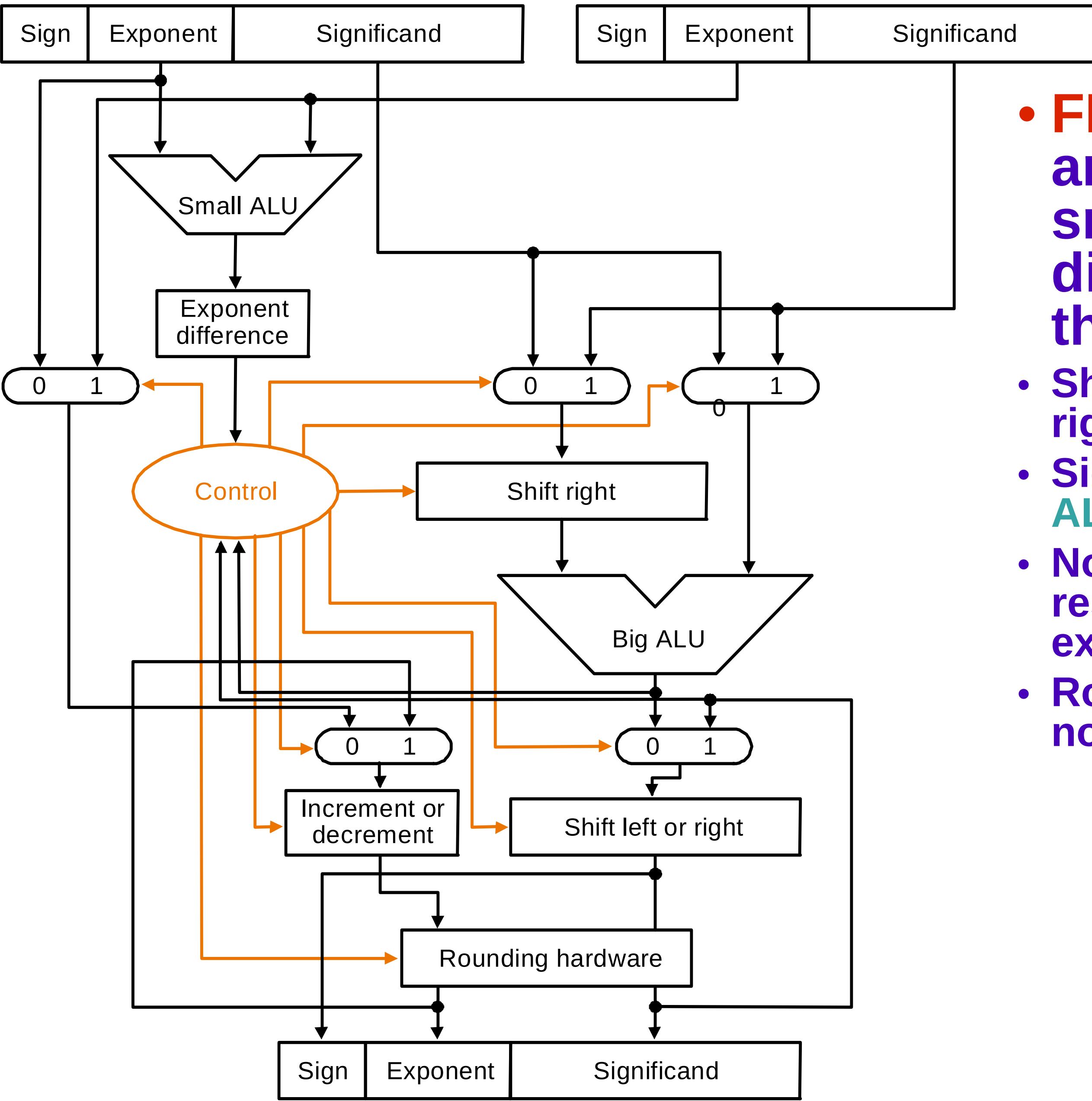
Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- fl. pt. number
255	0	+/- infinity
255	nonzero	NaN

- **Exp. = 0, Significand nonzero?**

  Can we get greater precision?

- **Represent very, very small magnitude numbers**
- **0 < *x* < smallest normalized number);**
- **Denormalized Numbers (text p. 300, and discussion later).**

Floating Point ALU



- **FP ADD: Exponents are subtracted by small ALU; the difference controls the 3 MUXes;**
  - **Shift smaller exp. to the right until exponents match;**
  - **Significants are added in Big ALU;**
  - **Normalization step shifts result left or right, adjusts exponents;**
  - **Rounding and possible nornalization**

MIPS Floating Point Architecture (1/4)

- **Separate floating point instructions:**
  - Single Precision:

    `add.s, sub.s, mul.s, div.s`
  - Double Precision:

    `add.d, sub.d, mul.d, div.d`
- **These instructions are far more complicated than their integer counterparts, so they can take much longer to execute.**

MIPS Floating Point Architecture (2/4)

- **Problems:**

It’s inefficient to have different instructions take vastly differing amounts of time.

Generally, a particular piece of data will not change from FP to int, or vice versa, within a program. So only one type of instruction will be used on it.

Some programs do no floating point calculations

It takes lots of hardware relative to integers to do Floating Point fast

MIPS Floating Point Architecture (3/4)

- **1990 Solution: Make a completely separate chip that handles only FP.**

- **Coprocessor 1: FP chip**

1. contains 32 32-bit registers: `$f0`, `$f1`, …
2. most of the registers specified in `.s` and `.d` instruction refer to this set
3. separate load and store: `lwc1` and `swc1` ("load word coprocessor 1", "store …")
4. Double Precision: by convention, even/odd pair contain one DP FP number: `$f0`/`$f1`, `$f2`/`$f3`, … , `$f30`/`$f31`

MIPS Floating Point Architecture (4/4)

- **1990 Computer actually contains multiple separate chips:**

  Processor: handles all the normal stuff

  Coprocessor 1: handles FP and only FP;

  more coprocessors?… Yes, later

  Today, FP coprocessor integrated with CPU, or cheap chips may leave out FP HW

- **Instructions to move data between main processor and coprocessors:**

`mfc1 rt, rd`	Move floating point register rd to CPU register rt.
`mtc1 rd, rt`	Move CPU register rt to floating point register rd.
`mfc1.d rdest, frsrc1`	Move floating point registers frsrc1 & frsrc1 + 1 to CPU registers rdest & rdest + 1.

- **Appendix pages A-70 to A-74 contain many, many more FP operations.**

Summary: MIPS F.P. Architecture

- **Single Precision, Double Precision versions of add, subtract, multiply, divide, compare**

Single	`add.s,`	`sub.s,`	`mul.s,`	`div.s,`	`c.lt.s`
Double	`add.d,`	`sub.d,`	`mul.d,`	`div.d,`	`c.lt.d`

See pages A-70 - A74

- **Registers?**
  - Normally integer and Floating Point operations on different data, for performance should have separate registers.
  - MIPS adds 32 32-bit FP regs: `$f0`, `$f1`, `$f2` ...,
  - Thus need FP data transfers:

l.d	fdest, address	load the floating point double at address into register fdest.
mov.s	fd, fs	Move the floating point single from register fs to register fd.

  - Double Precision? Even-odd pair of registers:

`$f0`-`$f1`, `$f2-$3`, *etc.*, act as 64-bit register: `$f0`, `$f2`, `$f4`,

Example with F.P.: Matrix Multiply

```
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];

}
```

- **Starting addresses are parameters in $a0, $a1, and $a2. Integer variables are in $t3, $t4, $t5. Arrays 32 by 32**
- **Use pseudoinstructions: `li` (load immediate), `l.d` / `s.d` (load / store 64 bits)**

MIPS code 1st piece: initialize `x[][]`

- **Initialize Loop Variables**

```
mm:   ...
      li    $t1, 32      # $t1 = 32
      li    $t3, 0       # i = 0; 1st loop
L1:   li    $t4, 0       # j = 0; reset 2nd
L2:   li    $t5, 0       # k = 0; reset 3rd
```

- **To fetch x[i][j], skip i rows (i*32), add j**

```
 sll        $t2,$t3,5    # $t2 = i * 2^5
 addu       $t2,$t2,$t4  # $t2 = i*2^5 + j
```

- **Get byte address (8 bytes), load x[i][j]**

```
 sll        $t2, $t2,3   # i,j byte addr.
 addu       $t2, $a0,$t2# @ x[i][j]
 l.d        $f4, 0($t2)  # $f4 = x[i][j]
```

MIPS code 2nd piece: `z[][], y[][]`

- **Like before, but load z[k][j] into $f16**

```
L3:  sll    $t0, $t5, 5        # $t0 = k * 2^5
     addu   $t0, $t0, $t4      # $t0 = k*2^5 +j
     sll    $t0, $t0, 3        # k,j byte addr.
     addu   $t0, $a2, $t0      # @ z[k][j]
     l.d    $f16, 0($t0)       # $f16 = z[k][j]
```

- **Like before, but load y[i][k] into $f18**

```
     sll    $t0, $t3, 5        # $t0 = i * 2^5
     addu   $t0, $t0, $t5      # $t0 = i*2^5 +k
     sll    $t0, $t0, 3        # i,k byte addr.
     addu   $t0, $a1, $t0      # @ y[i][k]
     l.d    $f18, 0 ($t0)      # $f18 = y[i][k]
```

- **Summary:** `$f4: x[i][j], $f16: z[k][j], $f18: y[i][k]`

MIPS code for last piece: add/mul, loops

- **Add y*z to x**

```
        mul.d $f16,$f18,$f16        # y[][]*z[][]
        add.d $f4, $f4, $f16        # x[][]+ y*z
```

- **Increment k; if end of inner loop, store x**

```
        addiu $t5, $t5,1            # k = k + 1
        bne       $t5, $t1,L3       # if(k!=32) goto L3
        s.d       $f4, 0($t2)       # x[i][j] = $f4
```

- **Increment j; middle loop if not end of j**

```
        addiu $t4, $t4,1            # j = j + 1
        bne       $t4, $t1,L2       # if(j!=32) goto L2
```

- **Increment i; if end of outer loop, return**

```
        addiu $t3,$t3,1             # i = i + 1
        bne       $t3,$t1,L2               # if(i!=32) goto L1
        jr        $ra
```

Floating Point gottchas: Add Associativity?

- **$x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, and $z = 1.0$**
- **$x + (y + z) = -1.5\text{x}10^{38} + (1.5\text{x}10^{38} + 1.0)$**

  $= -1.5\text{x}10^{38} + (1.5\text{x}10^{38}) =$ <u>0.0</u>
- **$(x + y) + z = (-1.5\text{x}10^{38} + 1.5\text{x}10^{38}) + 1.0$**
- **$= (0.0) + 1.0 = 1.0$**
- **Therefore, Floating Point addition <u>not associative!</u>**

  1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ is still 1.5×10^{38}

  FP result approximation of real result!
- **What are the conditions that make smaller arguments "disappear" (rounded down to 0.0)?**

Basic Addition Algorithm/Multiply issues

Addition (or subtraction) includes the following steps:

Good Summary

(1) compute *Ye* - *Xe* (*getting ready to align binary point*)

(2) right shift Xm that many positions to form $Xm \times 2^{Xe - Ye}$

(3) compute ($Xm \times 2^{Xe - Ye}$) + Ym

if representation demands normalization, then normalization step follows:

(4) left shift result, decrement result exponent (e.g., 0.001xx…)
right shift result, increment result exponent (e.g., 101.1xx…)
continue until MSB of data is 1 (NOTE: Hidden bit in IEEE Standard)

(5) for Multiply, doubly biased exponent must be corrected:

Xe = 7
Ye = -3
Excess 8 extra subtraction step of the bias amount

(6) if result is 0 mantissa, may need to zero exponent by special step

Xe = 1111	= 15	= 7 + 8
Ye = 0101	= 5	= -3 + 8
10100	20	4 + 8 + 8

Rounding and IEEE Rounding Modes

- **When we perform math on “real” numbers, we have to worry about rounding to fit the result in the significant field.**

- **The FP hardware carries two extra bits of precision, and then round to get the proper value**

- **Rounding also occurs when converting a double to a single precision value, or converting a floating point number to an integer**

  Round towards $+\infty$
    - ALWAYS round “up”: 2.001 $\rightarrow$ 3
    - -2.001 $\rightarrow$ -2

  Round towards **-**∞
    - ALWAYS round “down”: 1.999 $\rightarrow$ 1,
    - -1.999 $\rightarrow$ -2

  Truncate
    - Just drop the last bits (round towards 0)

  Round to (nearest) even
    - Normal rounding, almost

Round to Even

- **Round like you learned in grade school**
- **Except if the value is right on the borderline, in which case we round to the nearest EVEN number**

  2.5 -> 2

  3.5 -> 4

- **Insures fairness on calculation**

  This way, half the time we round up on tie, the other half time we round down

  Ask statistics majors

- **This is the default rounding mode**

Summary: Extra Bits for Rounding

"Floating Point numbers are like piles of sand; every time you move one you lose a little sand, but you pick up a little dirt."

How many extra bits?

IEEE: As if computed the result exactly and rounded.

Addition:

```
   1.xxxxx        1.xxxxx            1.xxxxx
 + 1.xxxxx        0.001xxxxx         0.01xxxxx
  1x.xxxxy        1.xxxxxyyy        1x.xxxxyyy
```

post-normalization **pre-normalization** **pre and post**

- ***Guard Digits*: digits to the right of the first *p* digits of significand to guard against loss of digits – can later be shifted left into first *P* places during normalization.**
- **Addition: carry-out shifted in**
- **Subtraction: borrow digit and guard**
- **Multiplication: carry and guard, Division requires guard**

Summary: Rounding Digits

Normalized result, but some non-zero digits to the right of the significand --> the number should be rounded

E.g., B = 10, p = 3:

	0	2	1.69	$= 1.6900 * 10^{2\text{-bias}}$
-	0	0	7.85	$= - .0785 * 10^{2\text{-bias}}$
	0	2	1.61	$= 1.6115 * 10^{2\text{-bias}}$

one round digit must be carried to the right of the guard digit so that after a normalizing left shift, the result can be rounded, according to the value of the round digit

***IEEE Standard*: four rounding modes:**

- **round to nearest even (default)**
- **round towards plus infinity**
- **round towards minus infinity**
- **round towards 0**

round to nearest:

round digit < B/2 then truncate
> B/2 then round up (add 1 to ULP: unit in last place)
= B/2 then round to nearest even digit

it can be shown that this strategy minimizes the mean error introduced by rounding

Elaboration: Sticky Bit

Additional bit to the right of the round digit to better fine tune rounding

```
  d0 . d1 d2 d3 . . . dp-1  0  0  0
+  0 .  0   0  X . . .  X    X X  S
                             X X  S
```

Sticky bit: set to 1 if any 1 bits fall off the end of the round digit

```
  d0 . d1 d2 d3 . . . dp-1  0  0  0
-  0 .  0   0  X . . .  X    X X  0
                                 X X 0
```

```
  d0 . d1 d2 d3 . . . dp-1  0  0  0
-  0 .  0   0  X . . .  X    X X  1
```

generates a borrow

Rounding Summary

Radix 2 minimizes wobble in precision

Normal operations in +,-,*,/ require one carry/borrow bit + one guard digit

One round digit needed for correct rounding

Sticky bit needed when round digit is B/2 for max accuracy

Rounding to nearest has mean error = 0, if *uniform distribution* of digits are assumed

C: Casting floats to ints and vice versa

- **`(int)`** ***`floating point exp`***

  Coerces and converts it to the nearest integer (C uses truncation)

  ```
  i = (int) (3.14159 * f);
  ```

- **`(float)`** ***`exp`***

  converts integer to nearest floating point

  ```
  f = f + (float) i;
  ```

C: float -> int -> float

```
if (f == (float)((int) f)) {
   printf("true");
}
```

- **Will not always print "true"**
- **Large values of integers don't have exact floating point representations**
- **What about double?**
- **Small floating point numbers (<1) don't have integer representations**
- **For other numbers, rounding errors**

Summary: Scientific Notation

decimal point

exponent

Sign, magnitude

$$6.02 \times 10^{23} \qquad\qquad 1.673 \times 10^{-24}$$

Mantissa

radix (base)

Sign, magnitude

IEEE F.P. $\pm 1.M \times 2^{e - 127}$

- **Issues:**
  - Arithmetic (+, -, *, /)
  - Representation, Normal form
  - Range and Precision
  - Rounding
  - Exceptions (*e.g.*, divide by zero, overflow, underflow)
  - Errors
  - Properties (negation, inversion, if $A \neq B$ then $A - B \neq 0$)

Summary : Floating-Point Arithmetic

Representation of floating point numbers in IEEE 754 standard:

single precision

	1	8	23
sign	S	E	M

exponent:
excess 127 binary integer

mantissa:
sign + magnitude, normalized binary significand w/ hidden integer bit: 1.M

actual exponent is
$e = E - 127$

$0 < E < 255$

$$N = (-1)^{S}\, 2^{E-127}\, (1.M)$$

0 = 0 00000000 0 . . . 0　　　　**-1.5 = 1 01111111 10 . . . 0**

Magnitude of numbers that can be represented is in the range:

$$2^{-126}\,(1.0) \quad \text{to} \quad 2^{127}\,(2 - 2^{-23})$$

which is approximately:

$$1.8 \times 10^{-38} \quad \text{to} \quad 3.40 \times 10^{38}$$

(integer comparison valid on IEEE Fl.Pt. numbers of same sign!)

Things to Remember

- **Floating Point numbers *approximate* values that we want to use.**
- **IEEE 754 Floating Point Standard is most widely accepted attempt to standardize interpretation of such numbers**
- **New MIPS registers(`$f0`-`$f31`), instruct.ions:**

  Single Precision (32 bits, $2x10^{-38}$… $2x10^{38}$): `add.s, sub.s, mul.s, div.s`

  Double Precision (64 bits , $2x10^{-308}$…$2x10^{308}$): `add.d, sub.d, mul.d, div.d`
- **Type is not associated with data, bits have no meaning unless given in context**