Chapter Three (2/2)

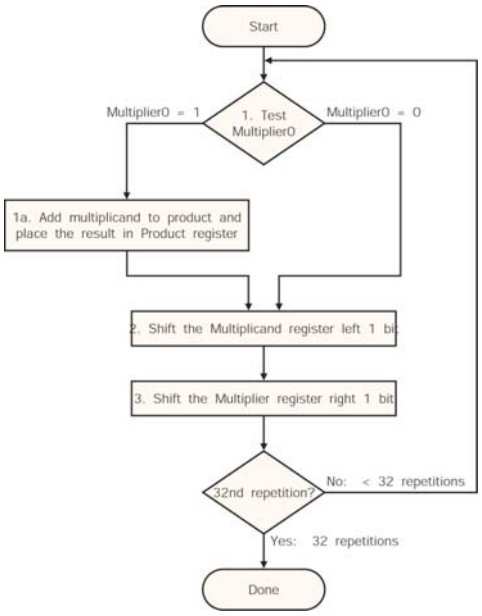
Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- n bits x n bits = 2n bit product
- Binary makes it easy:
  - 0 => place 0 (0 x multiplicand)
  - 1 => place a copy (1 x multiplicand)

```
    0010   (multiplicand)
  x 1011   (multiplier)
    0010
   0010
  0000
 0010
 0010110
```

- Negative numbers: convert and multiply
  - there are better techniques, we won't look at them

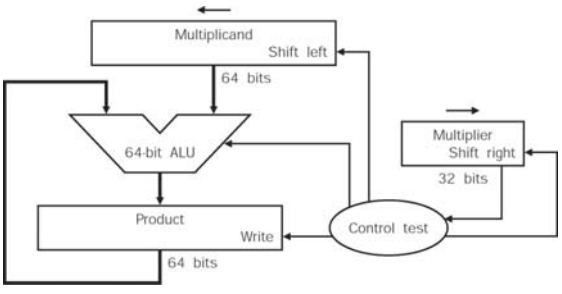
Multiplication: Version 1



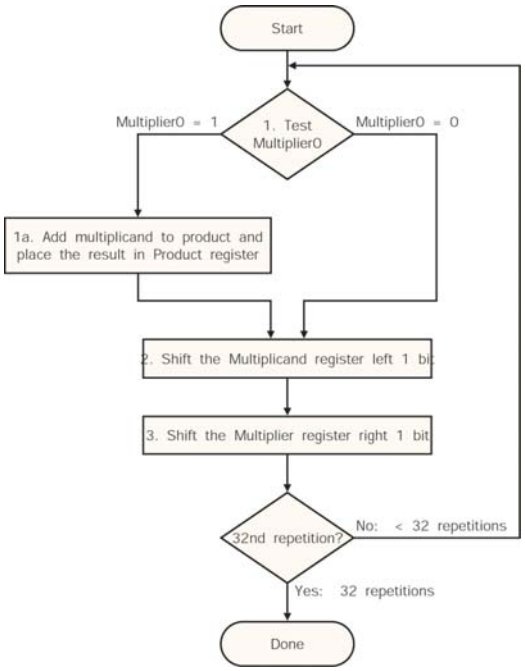
Example: 2 (0010) * 3 (0011)

Iteration	Step	Multiplier	Multiplicand	Prod
0		0011	0000 0010	0000 0000
1	Add	0011	0000 0010	0000 0010
	Shift L MC	0011	0000 0100	0000 0010
	Shift R MP	0001	0000 0100	0000 0010
2	Add	0001	0000 0100	0000 0110
	Shift L MC	0001	0000 1000	0000 0110
	Shift R MP	0000	0000 1000	0000 0110
3	No op	0000	0000 1000	0000 0110
	Shift L MC	0000	0001 0000	0000 0110
	Shift R MP	0000	0001 0000	0000 0110
4	No op	0000	0001 0000	0000 0110
	Shift L MC	0000	0010 0000	0000 0110
	Shift R MP	0000	0010 0000	0000 0110

Multiplication: Version 1

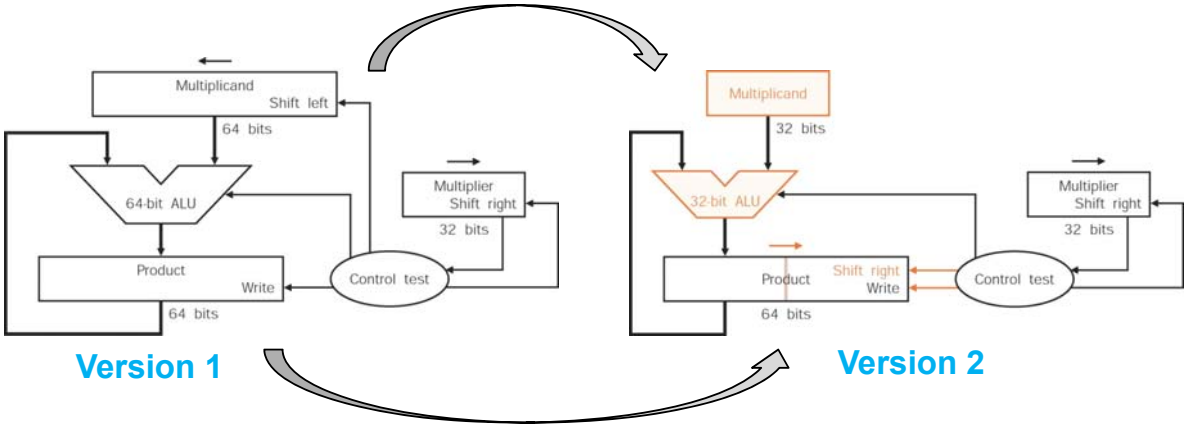


Multiplicand	Multiplier	Product
0000 1011	0101:1	0000 1011
0001 0110	0010:0	0000 1011
0010 1100	0001:1	0011 0111
0101 1000	0000:0	0011 0111



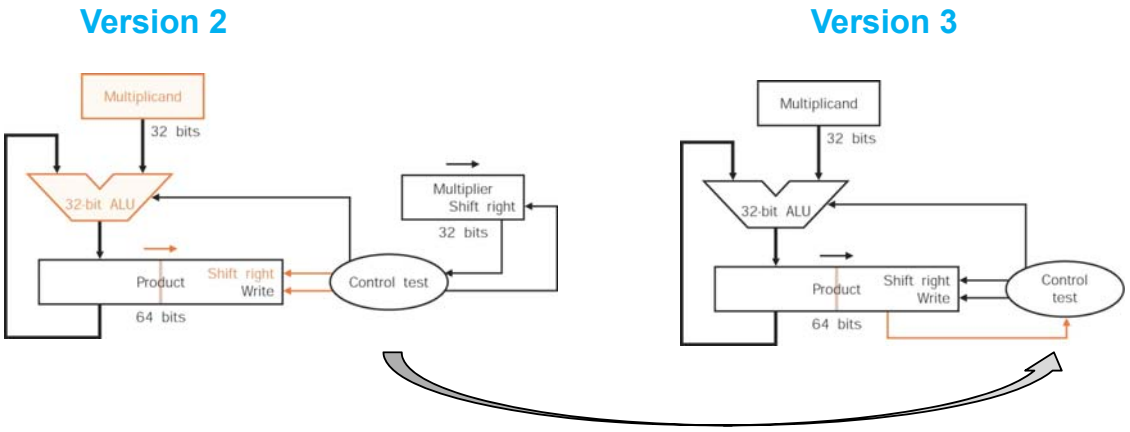
Observations on Multiply Version 1

- 1 clock per cycle => 32 clocks per multiply
- 1/2 bits in multiplicand always 0
  => 64-bit adder is wasted
- 0's inserted in left of multiplicand as shifted
  => least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right.



Observations on Multiply Version 2

- 2 cycles per bit because shift is performed after addition
- Product register wastes space that exactly matches size of multiplier
- Both Multiplier register and Product register require right shift
- Combine Multiplier register and Product register

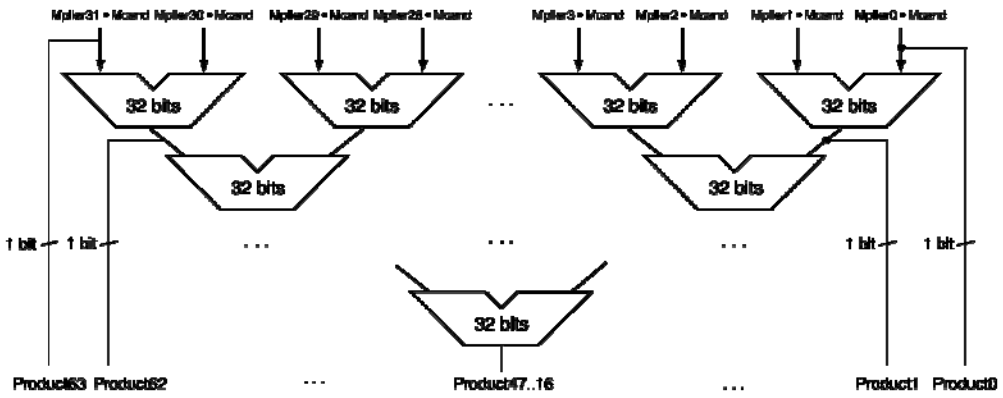


Observations on Multiply Version 3

- 2 cycles per bit because shift is performed after addition
- MIPS registers Hi and Lo are left and right half of Product
- Gives us MIPS instruction MultU
- 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 be modified to handle multiple bits at a time

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff



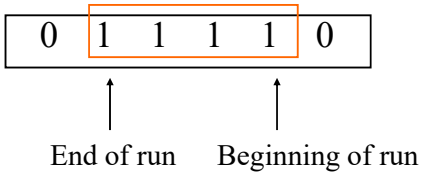
- Can be pipelined
  - Several multiplication performed in parallel

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

Booth's Algorithm

- **Multiplier**



  - **a × 30 = a × (32 – 2)**

- **Depending on the current and previous bits, do one of the following:**
  - **00: no operation**
  - **01: add the multiplicand to the left half of the product**
  - **10: subtract the multiplicand from the left of the product**
  - **11: no operation**

Booth's Algorithm

- Depending on the current and previous bits, do one of the following:
  - 00: no operation
  - 01: add the multiplicand to the left half of the product
  - 10: subtract the multiplicand from the left of the product
  - 11: no operation

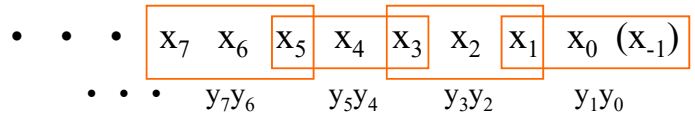
Example: 2 (0010) * 3 (0110)

Iteration	Step	Multiplicand	Prod
0		0010	0000 0110 0
1	00 -> no op	0010	0000 0110 0
	Shift R MP	0010	0000 0011 0
2	10 -> sub	0010	1110 0011 0
	Shift R MP	0010	1111 0001 1
3	11 -> no op	0010	1111 0001 1
	Shift R MP	0010	1111 1000 1
4	01 -> add	0010	0001 1000 1
	Shift R MP	0010	0000 1100 0

Radix-4 Modified Booth's Algorithm

- Considering three consecutive bits:

x_i	x_{i-1}	x_{i-2}	y_i	y_{i-1}	operation	comments
0	0	0	0	0	$+0$	string of zeros
0	1	0	0	1	$+A$	a single 1
1	0	0	$\bar{1}$	0	$-2A$	beginning of 1's
1	1	0	0	$\bar{1}$	$-A$	beginning of 1's
0	0	1	0	1	$+A$	end of 1's
0	1	1	1	0	$+2A$	end of 1's
1	0	1	0	$\bar{1}$	$-A$	a single 0
1	1	1	0	0	$+0$	string of 1's



- Further improvements
  - Radix-8 modified Booth's algorithm
  - Other parallel approaches

Modified Booth's Algorithm

- **Depending on the current and previous bits, do one of the following:**

x_i	x_{i-1}	x_{i-2}	y_i	y_{i-1}	operation	comments
0	0	0	0	0	$+0$	string of zeros
0	1	0	0	1	$+A$	a single 1
1	0	0	$\bar{1}$	0	$-2A$	beginning of 1's
1	1	0	0	$\bar{1}$	$-A$	beginning of 1's
0	0	1	0	1	$+A$	end of 1's
0	1	1	1	0	$+2A$	end of 1's
1	0	1	0	$\bar{1}$	$-A$	a single 0
1	1	1	0	0	$+0$	string of 1's

Example: 2 (0010) * 3 (0110)

Iteration	Step	Multiplicand	Prod
0		0010	0000 0110 0
1	100 → –2A	0010	1100 0110 0
	Shift R2 MP	0010	1111 0001 1
2	011 → 2A	0010	0011 0001 1
	Shift R2 MP	0010	0000 1100 0

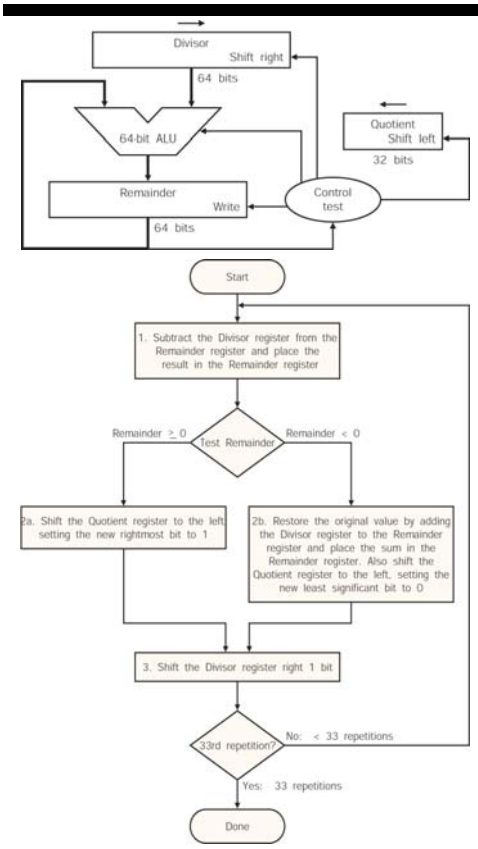
Unsigned Divide: Paper & Pencil

```
                    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

Division: Version 1

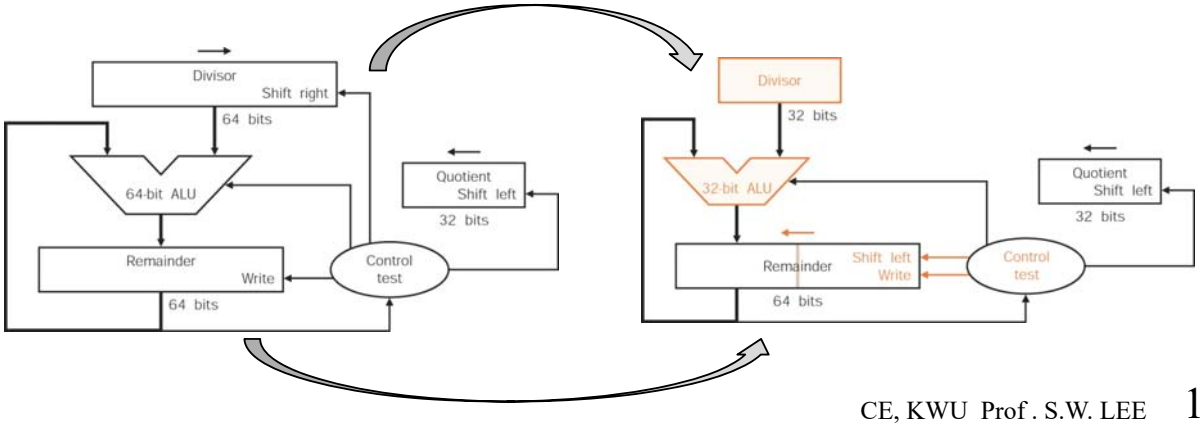


Example: 7 (0111) / 2 (0010)

Iteration	Step	Q	Divisor	Rem
0		0000	0010 0000	0000 0111
1	Rem - Div	0000	0010 0000	1110 0111
	Rest, SLQ, Q=0	0000	0010 0000	0000 0111
	Shift R Div	0000	0001 0000	0000 0111
2	Rem - Div	0000	0001 0000	1111 0111
	Rest, SLQ, Q=0	0000	0001 0000	0000 0111
	Shift R Div	0000	0000 1000	0000 0111
3	Rem - Div	0000	0000 1000	1111 1111
	Rest, SLQ, Q=0	0000	0000 1000	0000 0111
	Shift R Div	0000	0000 0100	0000 0111
4	Rem - Div	0000	0000 0100	0000 0011
	SLQ, Q=1	0001	0000 0100	0000 0011
	Shift R Div	0001	0000 0010	0000 0011
5	Rem - Div	0001	0000 0010	0000 0001
	SLQ, Q=1	0011	0000 0010	0000 0001
	Shift R Div	0011	0000 0001	0000 0001

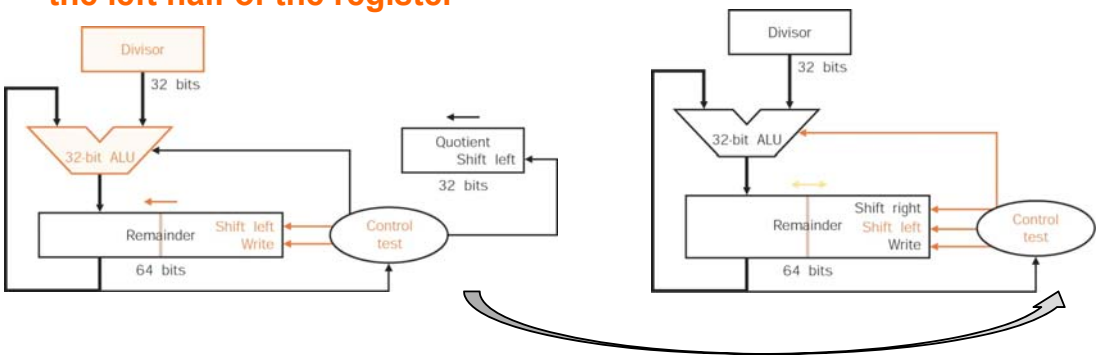
Observations on Divide Version 1

- **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 remainder to left?**
- **Notice that there is no way to get a 1 in leading digit! (this would be an overflow, since quotient would have n+1 bits)**
  - **1st step cannot produce a 1 in quotient bit (otherwise too big)**
  - **switch order to shift first and then subtract, can save 1 iteration**



Observations on Divide Version 2

- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
  - The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will shifted left one time too many.
  - Thus the final correction step must shift back only the remainder in the left half of the register



Observations on Divide Version 3

- Same Hardware as Multiply: just need ALU to add or subtract, and 63-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
- Possible for quotient to be too large: if divide 64-bit interger by 1, quotient is 64 bit

Faster Division

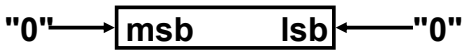
- **Can't use parallel hardware as in multiplier**
  - **Subtraction is conditional on sign of remainder**
- **Faster dividers (e.g. SRT division) generate multiple quotient bits per step**
  - **Still require multiple steps**
  - **SRT division: Sweeney, Robertson, and Tocher.**
    Lookup table based on the dividend and the divisor to determine each quotient digit.

MIPS Division

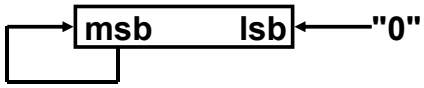
- **Use HI/LO registers for result**
  - **HI: 32-bit remainder**
  - **LO: 32-bit quotient**
- **Instructions**
  - `div rs, rt / divu rs, rt`
  - **No overflow or divide-by-0 checking**
    - Software must perform checks if required
  - **Use `mfhi`, `mflo` to access result**

Shifters

- Two kinds:
  - Logical -- value shifted in is always "0"

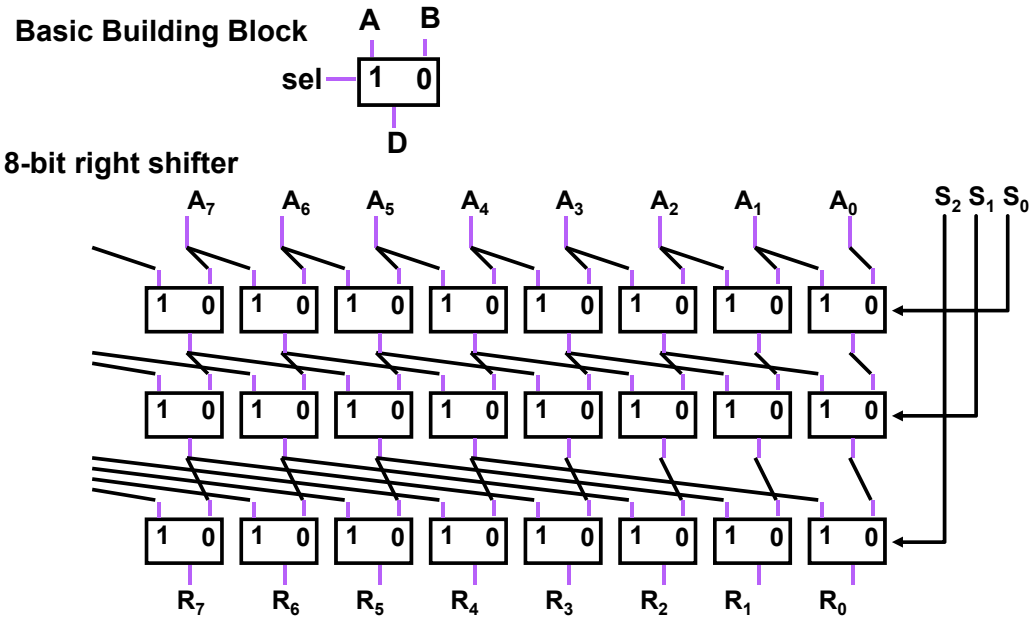


  - Arithmetic -- on right shifts, sign extend



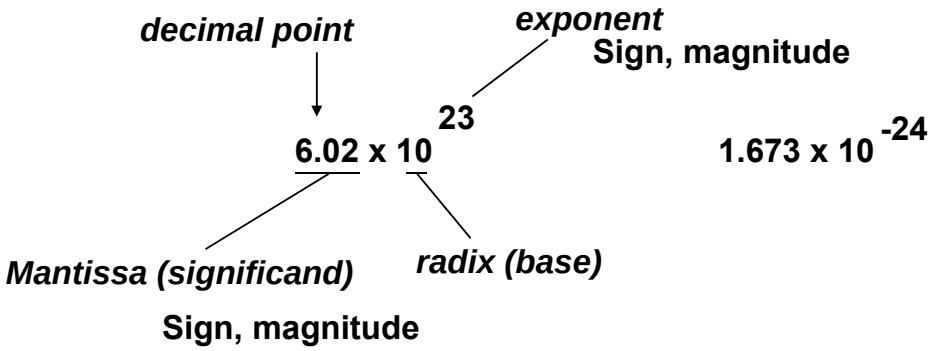
- Note: these are single bit shifts. A given instruction might request 0 to 32 bits to be shifted!

Combinational Shifter from MUXes (Barrel Shifter)



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

Recall Scientific Notation



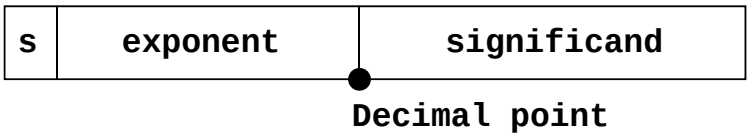
Floating Point (a brief look)

- We need a way to represent
  - numbers with fractions, e.g., 3.141592
  - very small numbers, e.g., 0.000000001
  - very large numbers, e.g., 3.15576×10^9
- Representation:
  - sign, exponent, significand: $(-1)^{sign} \times significand \times 2^{exponent}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision (32 bits): 8 bit exponent, 23 bit significand
  - double precision (64 bits): 11 bit exponent, 52 bit significand

s	exponent	significand

IEEE 754 floating-point standard

- Leading "1" bit of significand is implicit



- Exponent is "biased" to make sorting easier
  - all 0s is smallest exponent all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - summary: $(-1)^{sign} \times (1+significand) \times 2^{exponent - bias}$

- Example:

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

  - decimal: -0.75 = -3/4 = $-3/2^2 = -3 \times 2^{-2}$
  - binary: -0.11 = -1.1 x 2^{-1}
  - floating point: exponent = 126 = 01111110
  - IEEE single precision: 10111111010000000000000000000000

Converting from Binary to Decimal Floating Point

- What is the decimal single-precision floating point number that corresponds to the bit pattern 01000100010010010000000000000000?
- Use the equation

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

  where

  S = 0

  E = $10001000_2 = 136_{10}$

  1.M = 1.00100100000000000000000 = $1 + 2^{-1} + 2^{-4} + 2^{-7} = 1.5703125$

  so

  X = $(-1)^0 \times 2^{136-127} \times 1.5703125 = 804 = 8.04 \times 10^2$

Converting from Decimal to Binary Floating Point

- What is the binary representation for the single-precision floating point number that corresponds to X = -12.25_{10}?
- What is the normalized binary representation for the number?

  $-12.25_{10} = -1100.01_2 = -1.10001_2 \times 2^3$
- What are the sign, stored exponent, and normalized mantissa?

  S = 1 (since the number is negative)

  E = 3 + 127 = 130 = 128 + 2 = 10000010_2

  M = $10001000000000000000000_2$

  X = $11000001010001000000000000000000_2$
- What is the binary representation for the double-precision floating point number that corresponds to X = -12.25_{10}?

  $1100000000101000100000000000000000000000000000000000000000000000_2$
- NOTE: Zero is represented using 0 for the exponent and 0 for the mantissa. Either, +0 or -0 can be represented, based on the sign bit.

Special Numbers

- E = 0 is special for this reason.
  - Zero significand: number is 0

    $(-1)^S (0.0) \times 2^{-126} = \pm 0$
  - Non-zero significand: denormalized number

    $(-1)^S (0.M) \times 2^{-126}$
  - Denormal numbers are used to extend the range of floating-point numbers.
- Infinity and NaNs
  - Representation: exponent is all 1's (255 or 2047).
  - If significand is 0, ∞; otherwise NaN (not a number).

Exponent	Zero Significand	Non-zero Significand	Equation
00_H	zero, −0	Subnormal Numbers	$(-1)^{sign} \times 2^{-126} \times$ 0.significand
01_H, ..., FE_H	normalized value		$(-1)^{sign} \times 2^{exponent-127} \times$ 1.significand
FF_H	±infinity	NaN	

Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have "underflow"
- Accuracy can be a big problem
  - IEEE 754 keeps extra bits, guard, round, and sticky
    - Guard bit for aligned exponents
    - Four rounding modes for round-off
  - positive divided by zero yields "infinity"
  - zero divide by zero yields "not a number"
  - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

Additional Bits

- The maximum permissible error is ½ ulp
- Guard bit and Round bit
  - Example: 4-bit significand

    $1.0000 \times 2^0 - 1.0001 \times 2^{-2}$
  - Align

    1.0000×2^0

    $- 0.010001 \times 2^0$
  - Subtract

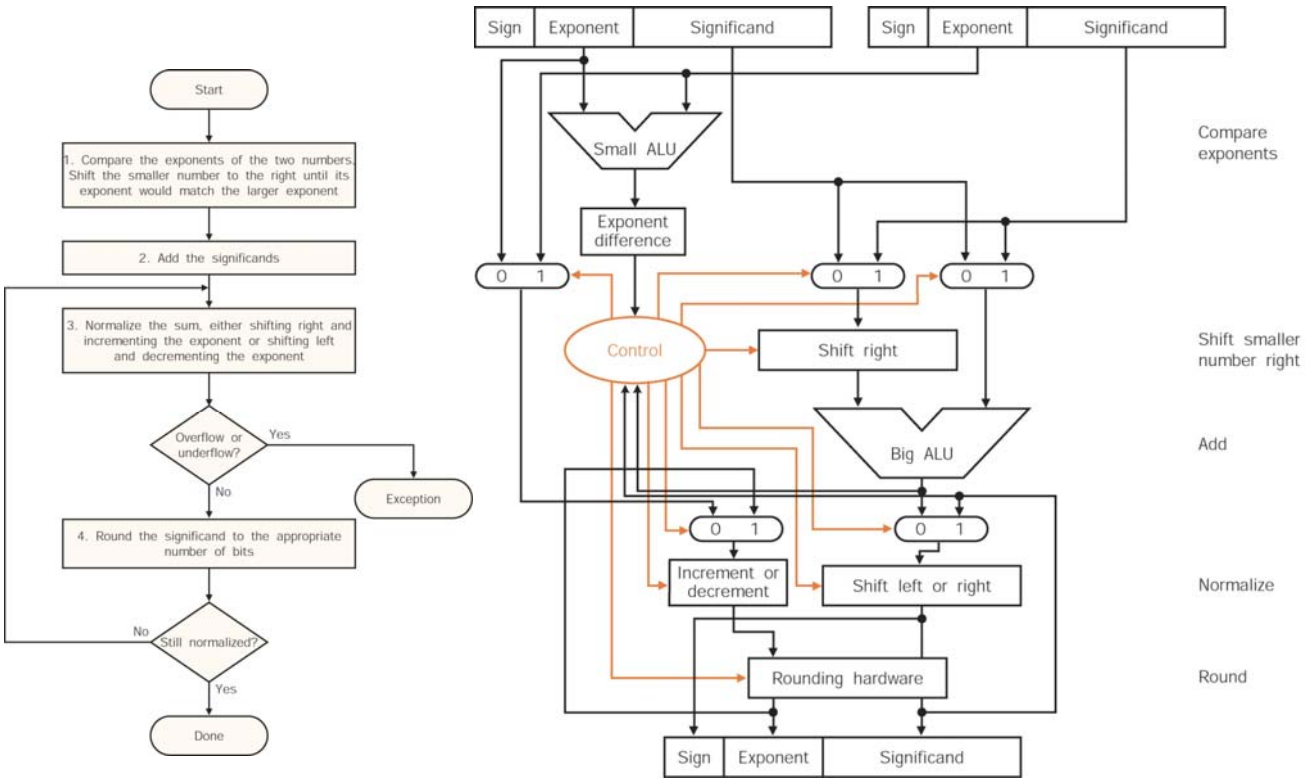
    0.101111×2^0
  - Normalize/Round

    1.01111×2^{-1}

    1.1000×2^{-1} (simple round up)
  - Without extra bit, result would be 1.0111×2^{-1}.
- Sticky bit
  - keep track of whether the bits "shifted out" are non-zero.

4 rounding modes:
- round to nearest even (default)
- round toward +∞
- round toward -∞
- round toward 0

Addition Block Diagram



Multiplication

Consider a 4-digit binary example

$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)

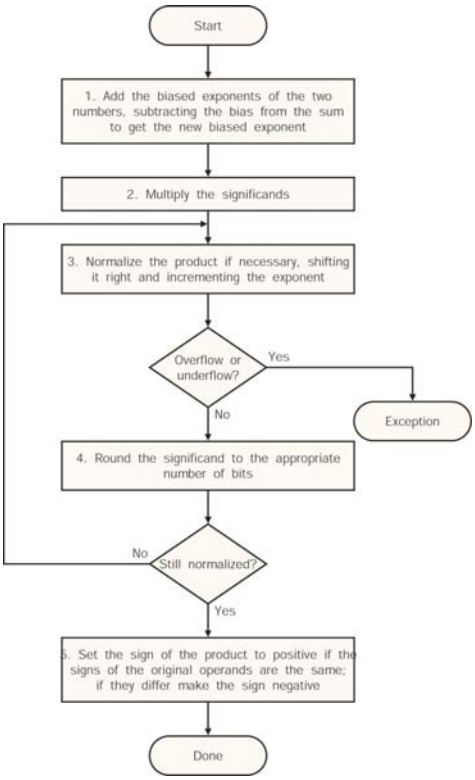
1. Add exponents
   Unbiased: $-1 + -2 = -3$
   Biased: $(-1 + 127) + (-2 + 127) - 127 = -3 + 127$

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 × –ve ⇒ –ve
   $-1.110_2 \times 2^{-3} = -0.21875$



Summary

- Computer arithmetic is constrained by limited precision
- Bit patterns have no inherent meaning but standards do exist
  - two's complement
  - IEEE 754 floating point
- Computer instructions determine "meaning" of the bit patterns
- Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation).