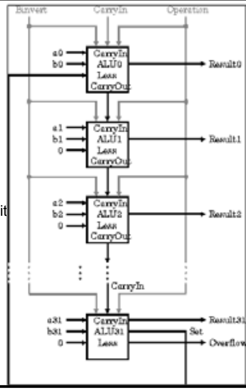
A 32 bit ALU

- A Ripple carry ALU
- Two bits decide operation
  - Add/Sub
  - AND
  - OR
  - LESS
- 1 bit decide add/sub operation
- A carry in bit
- Bit 31 generates overflow and set bit



Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products

Can you see the ripple? How could you get rid of it?

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

$c_2 = b_1c_1 + a_1c_1 + a_1b_1 \quad c_2 =$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2 \quad c_3 =$

$c_4 = b_3c_3 + a_3c_3 + a_3b_3 \quad c_4 =$

Not feasible! Why?

Carry look ahead adder

- An approach in-between our two extremes
- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry? $g_i = a_i\ b_i$
  - When would we propagate the carry? $p_i = a_i + b_i$
- Did we get rid of the ripple?

$c_1 = g_0 + p_0c_0$

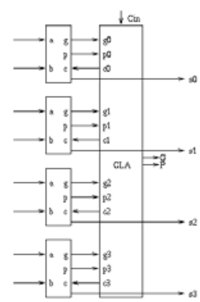
$c_2 = g_1 + p_1c_1 \quad c_2 = g_1 + p_1g_0 + p_1p_0c_0$

$c_3 = g_2 + p_2c_2 \quad c_3 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$

$c_4 = g_3 + p_3c_3 \quad c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$

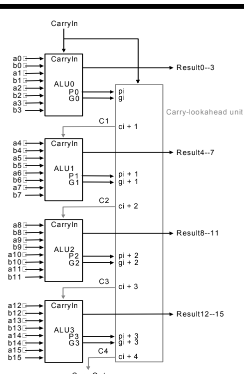
Feasible! Why?

A 4 bit carry look ahead adder



- Generate g and p term for each bit
- Use g's, p's and carry in to generate all C's
- Also use them to generate block G and P
- CLA principle can be used recursively

Use principle to build bigger adders



- A 16 bit adder uses four 4-bit adders
- It takes block g and p terms and cin to generate block carry bits out
- Block carries are used to generate bit carries
  - could use ripple carry of 4-bit CLA adders
  - Better: use the CLA principle again!

Delays in carry look ahead adders

- 4-Bit case
  - Generation of g and p: 1 gate delay
  - Generation of carries (and G and P): 2 more gate delay
  - Generation of sum: 1 more gate delay
- 16-Bit case
  - Generation of g and p: 1 gate delay
  - Generation of block G and P: 2 more gate delay
  - Generation of block carries: 2 more gate delay
  - Generation of bit carries: 2 more gate delay
  - Generation of sum: 1 more gate delay
- 64-Bit case
  - 12 gate delays

Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on grade school algorithm

```
 01010010  (multiplicand)
x01101101  (multiplier)
```

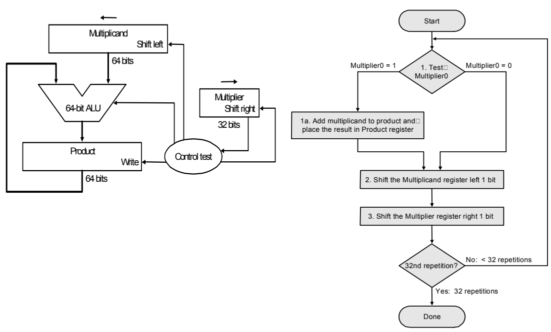
- Negative numbers: convert and multiply
- Use other better techniques like Booth's encoding

Multiplication

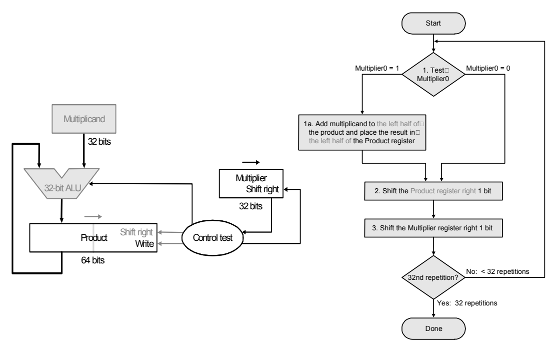
```
        01010010  (multiplicand)
       x01101101  (multiplier)
        00000000
        01010010   x1
        01010010
       000000000   x0
       001010010
      0101001000   x1
      0110011010
     01010010000   x1
     10000101010
    000000000000   x0
    010000101010
   0101001000000   x1
   0111001101010
  01010010000000   x1
  10001011101010
 000000000000000   x0
0010001011101010
```

```
 01010010  (multiplicand)
x01101101  (multiplier)
 00000000
 01010010   x1
 01010010
 000000000   x0
 001010010
 0101001000   x1
 0110011010
 01010010000   x1
 10000101010
 000000000000   x0
 010000101010
 0101001000000   x1
 0111001101010
 01010010000000   x1
 10001011101010
 000000000000000   x0
0010001011101010
```

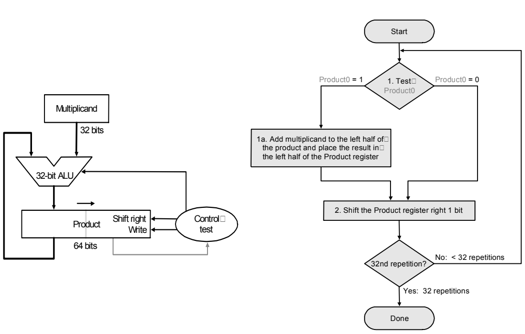

Multiplication: Implementation



Second Version



Final Version



Multiplication Example

Itera-tion	multi-plicand	Orignal algorithm	
		Step	Product
0	0010	Initial values	0000 0110
1	0010	1:0 ⇒ no operation	0000 0110
	0010	2: Shift right Product	0000 0011
2	0010	1a:1⇒ prod = Prod + Mcand	0010 0011
	0010	2: Shift right Product	0001 0001
3	0010	1a:1⇒ prod = Prod + Mcand	0011 0001
	0010	2: Shift right Product	0001 1000
4	0010	1:0 ⇒ no operation	0001 1000
	0010	2: Shift right Product	0000 1100

Signed Multiplication

- Let Multiplier be Q[n-1:0], multiplicand be M[n-1:0]
- Let F = 0 (shift flag)
- Let result A[n-1:0] = 0….00
- For n-1 steps do
  - A[n-1:0] = A[n-1:0] + M[n-1:0] x Q[0] /* add partial product */
  - F<= F .or. (M[n-1] .and. Q[0]) /* determine shift bit */
  - Shift A and Q with F, i.e.,
  - A[n-2:0] = A[n-1:1]; A[n-1]=F; Q[n-1]=A[0]; Q[n-2:0]=Q[n-1:1]
- Do the correction step
  - A[n-1:0] = A[n-1:0] - M[n-1:0] x Q[0] /* subtract partial product */
  - Shift A and Q while retaining A[n-1]
  - This works in all cases excepts when both operands are 10..00

Booth’s Encoding

- Numbers can be represented using three symbols, 1, 0, and -1
- Let us consider -1 in 8 bits
  - One representation is 1 1 1 1 1 1 1 1
  - Another possible one 0 0 0 0 0 0 0 -1
- Another example +14
  - One representation is 0 0 0 0 1 1 1 0
  - Another possible one 0 0 0 1 0 0 -1 0
- We do not explicitly store the sequence
- Look for transition from previous bit to next bit
  - 0 to 0 is 0; 0 to 1 is -1; 1 to 1 is 0; and 1 to 0 is 1
- Multiplication by 1, 0, and -1 can be easily done
- Add all partial results to get the final answer

Using Booth’s Encoding for Multiplication

- Convert a binary string in Booth’s encoded string
- Multiply by two bits at a time
- For n bit by n-bit multiplication, n/2 partial product
- Partial products are signed and obtained by multiplying the multiplicand by 0, +1, -1, +2, and -2 (all achieved by shift)
- Add partial products to obtain the final result
- Example, multiply 0111 (+7) by 1010 (-6)
- Booths encoding of 1010 is -1 +1 -1 0
- With 2-bit groupings, multiplication needs to be carried by -1 and -2
-

```
1 1 1 1 0 0 1 0   (multiplication by -2)
1 1 1 0 0 1 0 0   (multiplication by -1 and shift by 2 positions)
```

- Add the two partial products to get 11010110 (-42) as result

Booth’s algorithm (Neg. multiplier)

Itera-tion	multi-plicand	Booth’s algorithm	
		Step	Product
0	0010	Initial values	0000 1101 0
1	0010	1c: 10⇒ prod = Prod - Mcand	1110 1101 0
	0010	2: Shift right Product	1111 0110 1
2	0010	1b: 01⇒ prod = Prod + Mcand	0001 0110 1
	0010	2: Shift right Product	0000 1011 0
3	0010	1c: 10⇒ prod = Prod - Mcand	1110 1011 0
	0010	2: Shift right Product	1111 0101 1
4	0010	1d: 11 ⇒ no operation	1111 0101 1
	0010	2: Shift right Product	1111 1010 1

Carry Save Addition

- Consider adding six set of numbers (4 bits each in the example)
- The numbers are 1001, 0110, 1111, 0111, 1010, 0110 (all positive)
- One way is to add them pair wise, getting three results, and then adding them again

```
 1001    1111    1010    01111    100101
 0110    0111    0110    10110    10000
01111   10110   10000   100101    110101
```

- Other method is add them three at a time by saving carry

```
 1001    0111    00000    010101    001101
 0110    1010    11110    010100    101000
 1111    0110    01011    001100    110101
00000   01011    010101   001101    SUM
11110   01100    010100   101000    CARRY
```


Carry Save Addition for Multiplication

- n-bit carry-save adder take 1FA time for any n
- For n x n bit multiplication, n or n/2 (for 2 bit at time Booth’s encoding) partial products can be generated
- For n partial products n/3 n-bit carry save adders can be used
- This yields 2n/3 partial results
- Repeat this operation until only two partial results are remaining
- Add them using an appropriate size adder to obtain 2n bit result
- For n=32, you need 30 carry save adders in eight stages taking 8T time where T is time for one-bit full adder
- Then you need one carry-propagate or carry-look-ahead adder

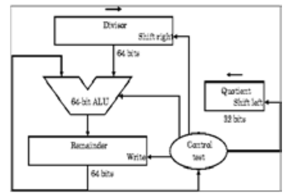
Division

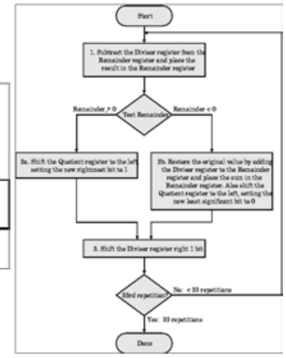
- Even more complicated
  - can be accomplished via shifting and addition/subtraction
- More time and more area
- We will look at 3 versions based on grade school algorithm

```
0011 | 0010 0010   (Dividend)
```

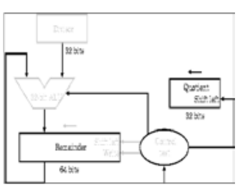
- Negative numbers: Even more difficult
- There are better techniques, we won't look at them

Division, First Version

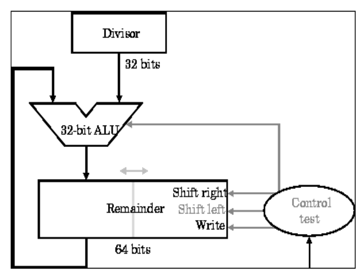




Division, Second Version



Division, Final Version



Restoring Division

Iteration	Divisor	Divide algorithm	
		Step	Remainder
0	0010	Initial values	0000 0111
	0010	Shift Rem left 1	0000 1110
1	0010	2: Rem = Rem - Div	1110 1110
	0010	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0001 1100
2	0010	2: Rem = Rem - Div	1111 1100
	0010	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0011 1000
3	0010	2: Rem = Rem - Div	0001 1000
	0010	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0011 0001
4	0010	2: Rem = Rem - Div	0001 0001
	0010	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010 0011
Done	0010	shift left half of Rem right 1	0001 0011

Non-Restoring Division

Iteration	Divisor	Divide algorithm	
		Step	Remainder
0	0010	Initial values	0000 1110
1	0010	1: Rem = Rem - Div	1110 1110
	0010	2b: Rem < 0 ⇒,sll R, R0 = 0	1101 1100
	0010	3b: Rem = Rem + Div	1111 1100
2	0010	2b: Rem < 0 ⇒ sll R, R0 = 0	1111 1000
	0010	3b: Rem = Rem + Div	0001 1000
3	0010	2a: Rem > 0 ⇒ sll R, R0 = 1	0011 0001
	0010	3a: Rem = Rem - Div	0001 0001
4	0010	2a: Rem > 0 ⇒ sll R, R0 = 1	0010 0011
Done	0010	shift left half of Rem right 1	0001 0011

Floating Point (a brief look)

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .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: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit 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: $(-1)^{sign} \times (1+\text{significand}) \times 2^{exponent - bias}$
- Example:
  - decimal: $-.75 = -3/4 = -3/2^2$
  - binary: $-.11 = -1.1 \times 2^{-1}$
  - floating point: exponent = 126 = 01111110
  - IEEE single precision: `10111111010000000000000000000000`

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 two extra bits, guard and round
  - four rounding modes
  - 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!

Floating Point Add/Sub

- To add/sub two numbers
  - We first compare the two exponents
  - Select the higher of the two as the exponent of result
  - Select the significand part of lower exponent number and shift it right by the amount equal to the difference of two exponent
  - Remember to keep two shifted out bit and a guard bit
  - add/sub the signifand as required according to operation and signs of operands
  - Normalize significand of result adjusting exponent
  - Round the result (add one to the least significant bit to be retained if the first bit being thrown away is a 1
  - Re-normalize the result

Floating Point Multiply

- To multiply two numbers
  - Add the two exponent (remember access 127 notation)
  - Produce the result sign as exor of two signs
  - Multiple significand portions
  - Results will be 1x.xxxxx… or 01.xxxx….
  - In the first case shift result right and adjust exponent
  - Round off the result
  - This may require another normalization step

Floating Point Divide

- To divide two numbers
  - Subtract divisor's exponent from the dividend's exponent (remember access 127 notation)
  - Produce the result sign as exor of two signs
  - Divide dividend's significand by divisor's significand portions
  - Results will be 1.xxxxx… or 0.1xxxx….
  - In the second case shift result left and adjust exponent
  - Round off the result
  - This may require another normalization step