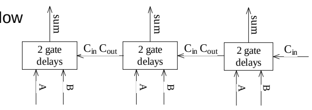
Computer Arithmetic II

Chapter Three P&H

Addition

- Ripple adders are slow



- What about sum-of-products representation?

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

$c_2 = b_1c_1 + a_1c_1 + a_1b_1$ $c_2 =$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2$ $c_3 =$

Carry Look-ahead Adder

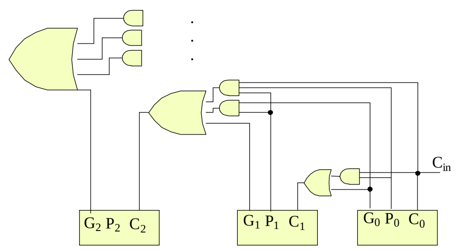
- An approach in-between our two extremes
- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry? $g_i = a_i\ b_i$
  - When would we propagate the carry? $p_i = a_i + b_i$

$c_1 = g_0 + p_0c_0$

$c_2 = g_1 + p_1c_1$ $c_2 =$

$c_3 = g_2 + p_2c_2$ $c_3 =$

Carry Look-ahead Adder



Example

```
a:   0001 1010 0011 0011
b:   1110 0101 1110 1011

gi:  0000 0000 0010 0011
pi:  1111 1111 1111 1011

G:     0    0    1    0
P:     1    1    1    0
C:     1    1    1    0
```

Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 4 versions based on high school algorithm

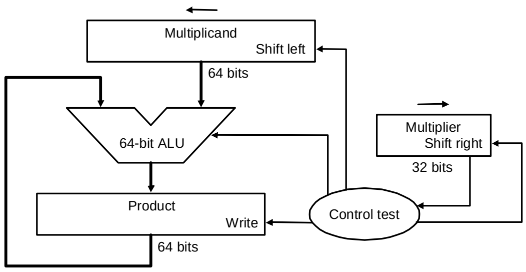
```
    0010  (multiplicand)
__x_1011  (multiplier)
```

Multiplication

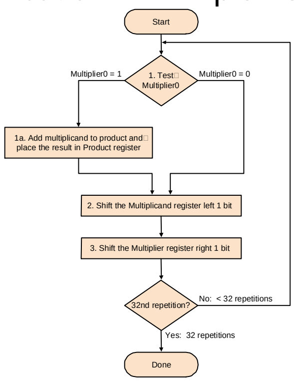
```
   0010    (2) ← multiplicand
 x 1011   (11) ← multiplier
   0010
  0010
 0000
0010
0010110   (22) ← product
```

- What logic is required to implement this?
- What can we say about the size requirements to store the product?

Multiplication: 1st Implementation



Multiplication: 1st Implementation



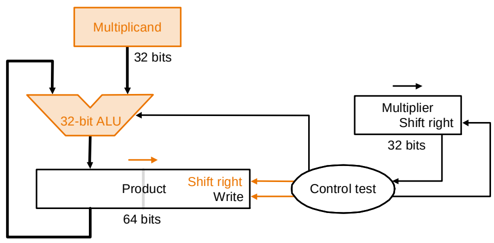
Board Exercise

- 2 x 3 = 6
  - 0010 x 0011 = 0110

Multiplication: 1st implementation performance

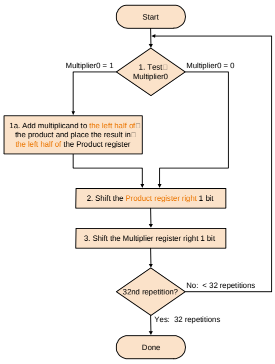
- How many steps does this implementation take?
- Is the implementation wasteful in other areas?

Second Version



use left hand side of product register.

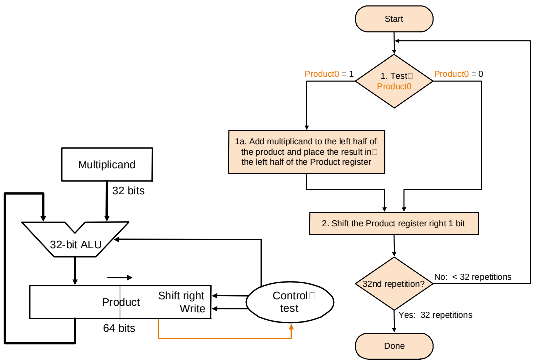
Second Version



Board Exercise

- 2 x 3 = 6
  - 0010 x 0011 = 0110

Third Version



Board Exercise

- 2 x 3 = 6
  - 0010 x 0011 = 0110

Multiplication: Third Implementation performance.

- How many steps does this implementation take?
- What other improvements have been made compared to the first implementation?

Signed multiplication

- An easy (to comprehend) way to do signed multiplication is
  - remember the original signs
  - convert the numbers to positive (temporary working) values
- when is the product negated?
- what extra cycles and resources are required?

Booth's Algorithm

- Does not require conversion cycles
- First step of the third multiplication implementation changes
- Second step (shift product right) remains
- The replacement step depends on the current and previous right-most bits in product
  - 00: no arithmetic op
  - 01: add multiplicand to left half of product
  - 10: sub multiplicand from left half of product
  - 11: no arithmetic op

Board Exercises

- 2 x -3 = -6
  - 0010 x 1101 = 1010

Multiplication by powers of 2

- accomplished using left shift
- e.g. 6 x 8 = 48
  - 0110 x 1000 = 00110000
  - 6 x 2^3 = 48
  - 6 << 3 = 48

Summary

- Multiplication is accomplished by shift and add hardware, using a similar algorithm to that we were taught in school.