15A05402:COMPUTER ORGANISATION

Integer Multipliers

S BABU
AP/CSE
KEC.

Basic Arithmetic and the ALU

- Earlier in the semester
  - Number representations, 2's complement, unsigned
  - Addition/Subtraction
  - Add/Sub ALU
    - Full adder, ripple carry, subtraction
  - Carry-lookahead addition
  - Logical operations
    - and, or, xor, nor, shifts
  - Overflow

Basic Arithmetic and the ALU

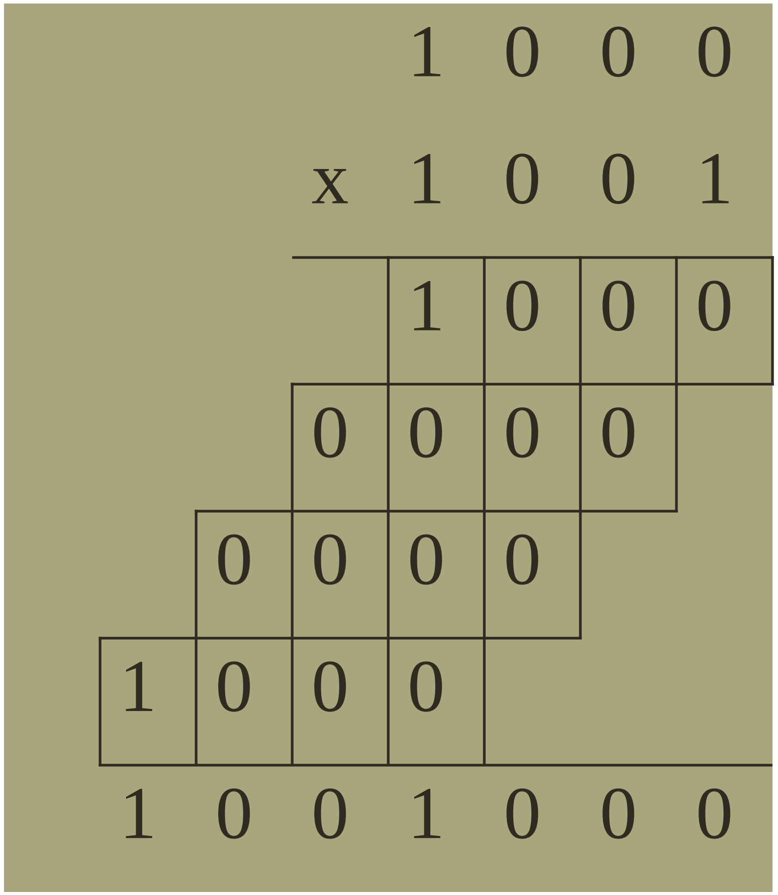
- Now
  - Integer multiplication
    - Booth’s algorithm
- This is not crucial for the project

Multiplication

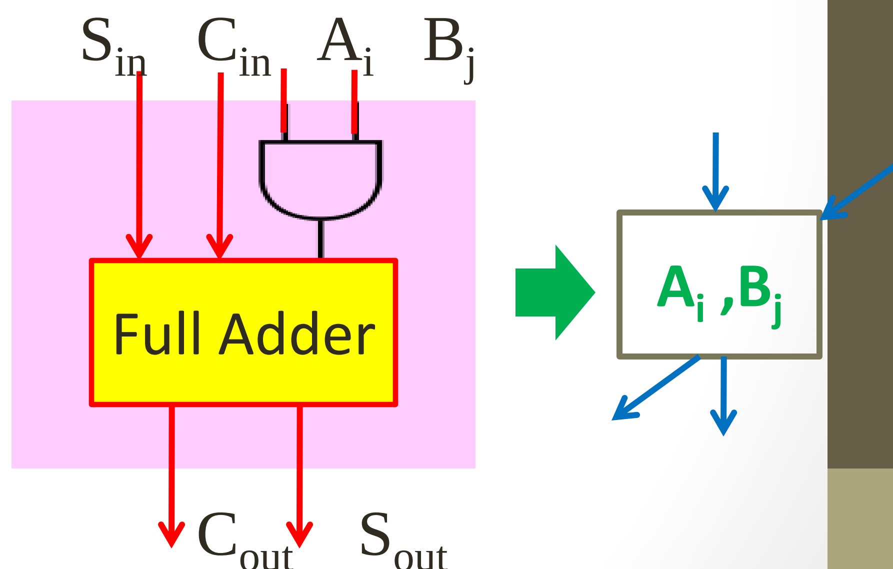
- Flashback to 3rd grade
  - Multiplier
  - Multiplicand
  - Partial products
  - Final sum
- Base 10: 8 x 9 = 72
  - PP: 8 + 0 + 0 + 64 = 72
- How wide is the result?
  - log(n x m) = log(n) + log(m)
  - 32b x 32b = 64b result

			1	0	0	0
		x	1	0	0	1
			1	0	0	0
		0	0	0	0	
	0	0	0	0		
1	0	0	0			
1	0	0	1	0	0	0

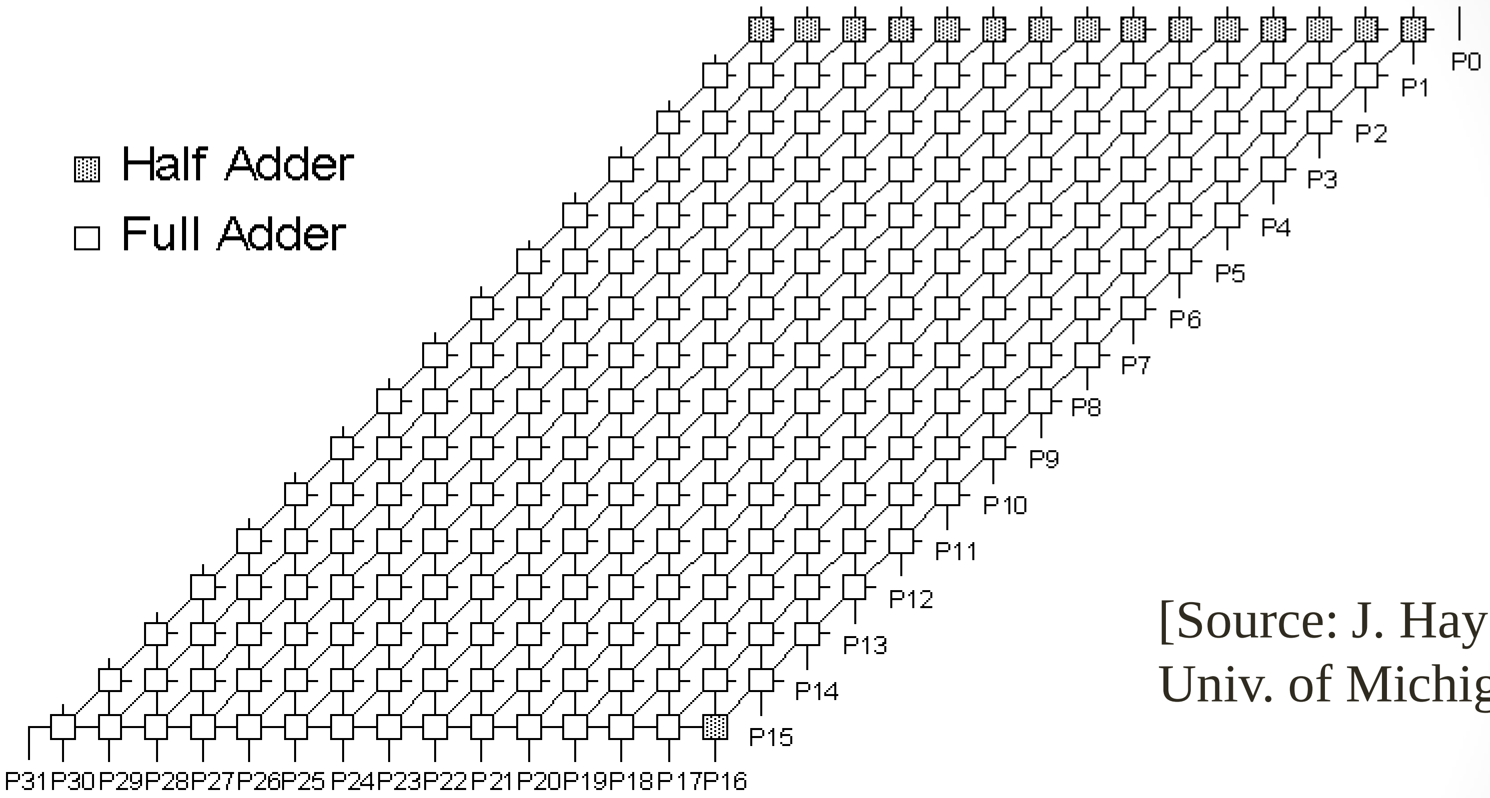
Array Multiplier



- Adding all partial products simultaneously using an array of basic cells



16-bit Array Multiplier



[Source: J. Hayes, Univ. of Michigan]

Conceptually straightforward

Fairly expensive hardware, integer multiplies relatively rare

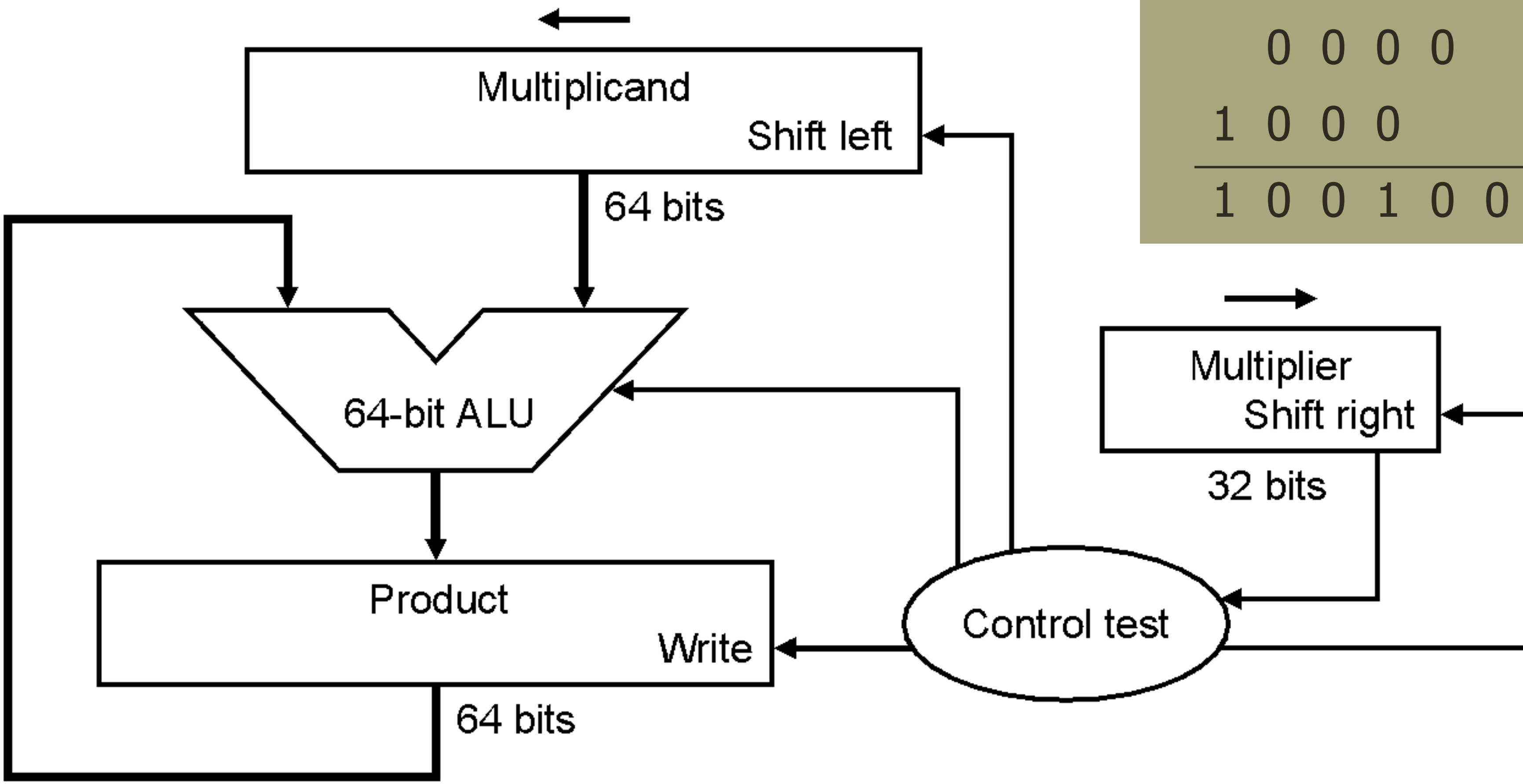
- Most used in array address calc: replace with shifts

Instead: Multicycle Multipliers

- Combinational multipliers
  - Very hardware-intensive
  - Integer multiply relatively rare
  - Not the right place to spend resources
- Multicycle multipliers
  - Iterate through bits of multiplier
  - Conditionally add shifted multiplicand

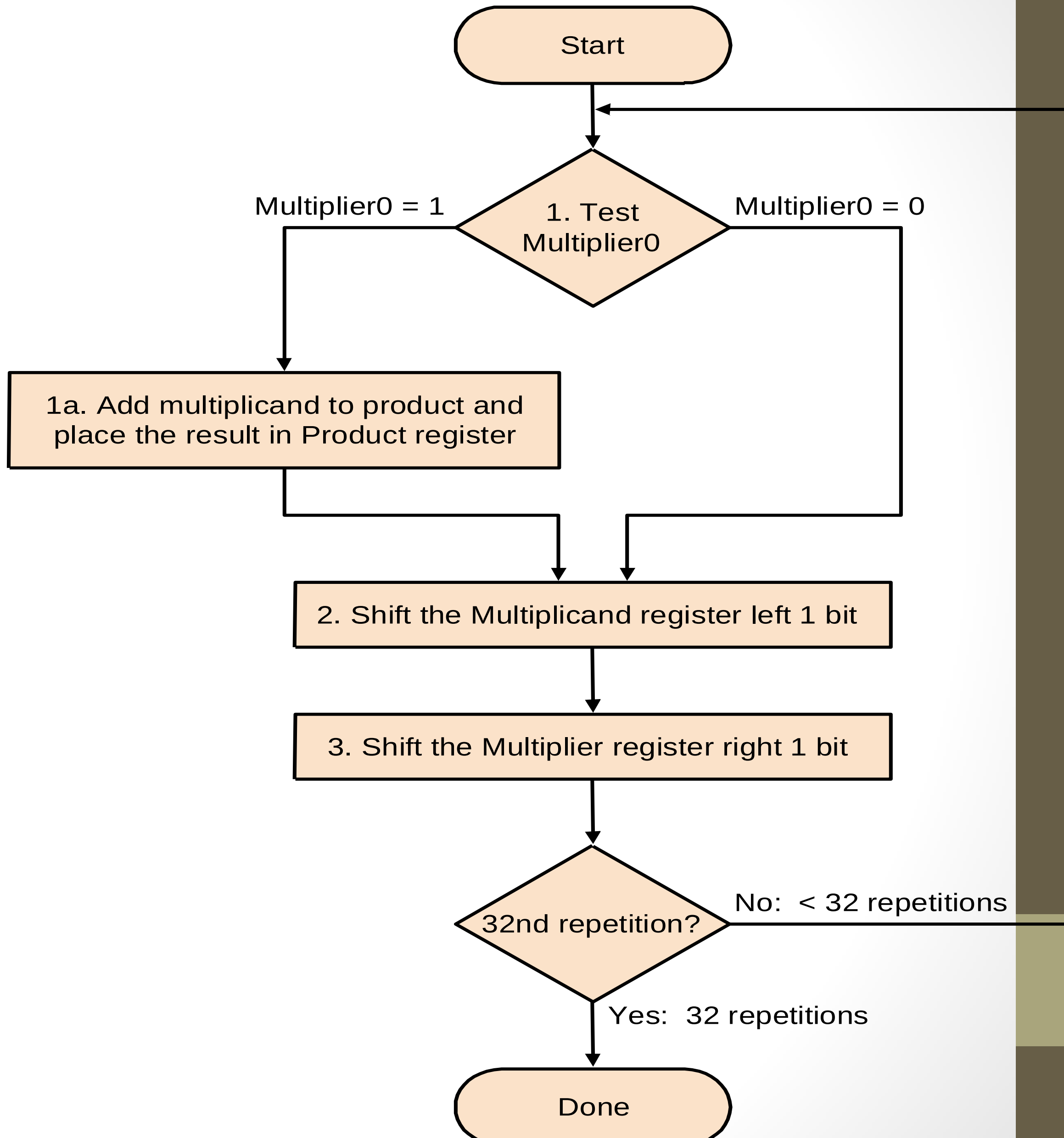
Multiplier

```
      1 0 0 0
    x 1 0 0 1
    ---------
      1 0 0 0
    0 0 0 0
  0 0 0 0
1 0 0 0
-------------
1 0 0 1 0 0 0
```



Multiplier

```
      1 0 0 0
    x 1 0 0 1
    ---------
      1 0 0 0
    0 0 0 0
  0 0 0 0
1 0 0 0
-------------
1 0 0 1 0 0 0
```

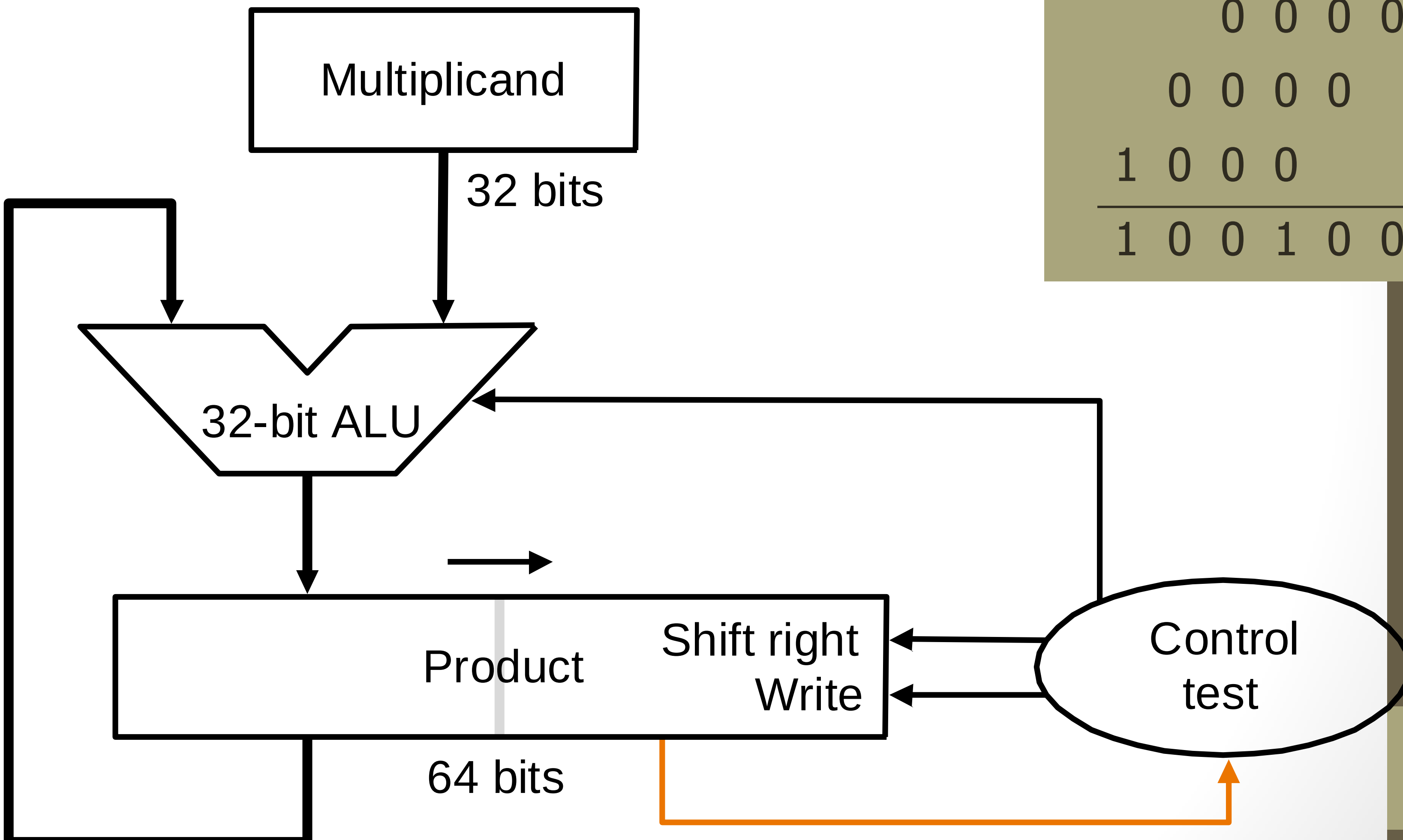


Multiplier Improvements

- Do we really need a 64-bit adder?
  - No, since low-order bits are not involved
  - Hence, just use a 32-bit adder
    - Shift product register right on every step
- Do we really need a separate multiplier register?
  - No, since low-order bits of 64-bit product are initially unused
  - Hence, just store multiplier there initially

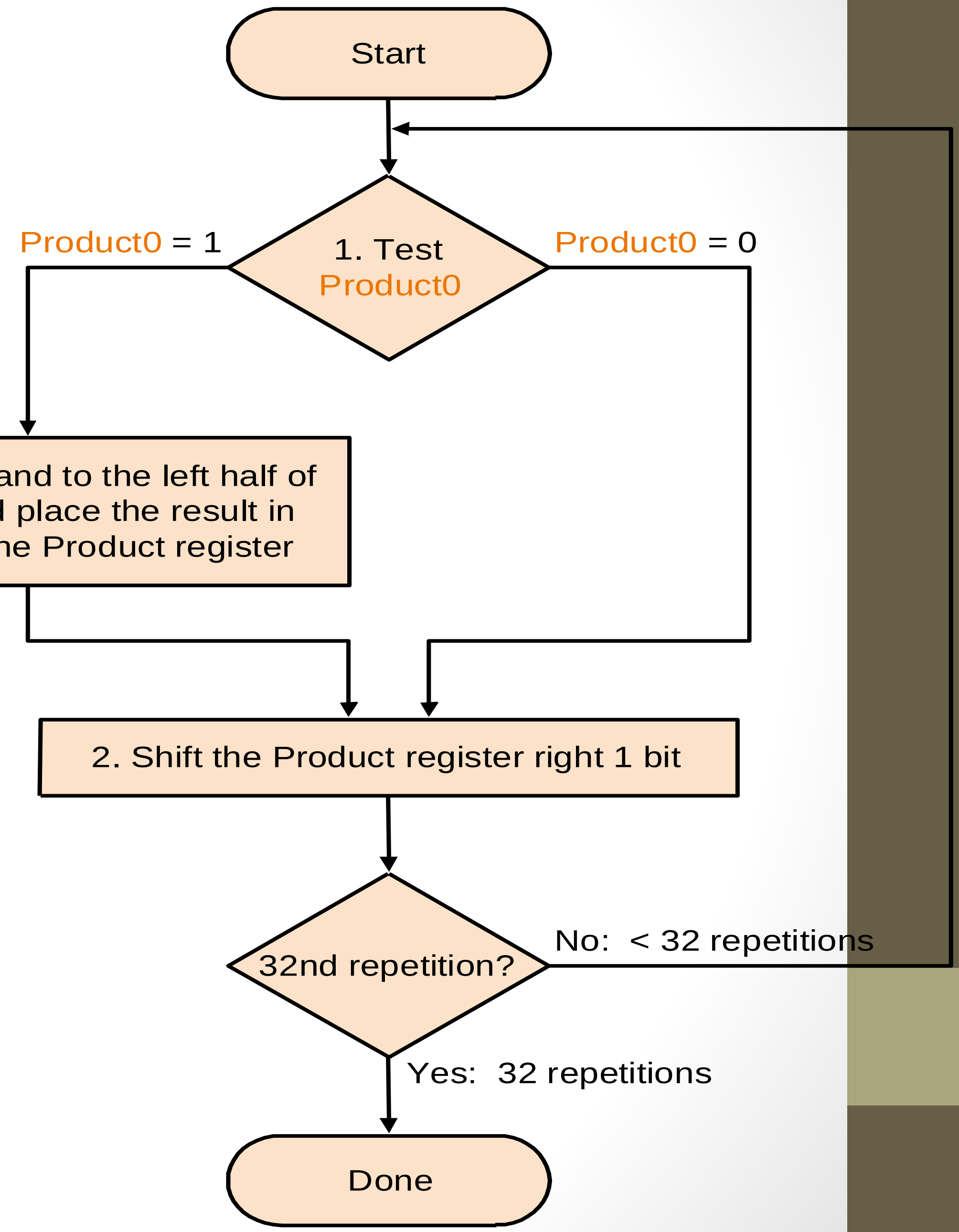
Multiplier

```
       1 0 0 0
     x 1 0 0 1
     ---------
       1 0 0 0
     0 0 0 0
   0 0 0 0
 1 0 0 0
 -------------
 1 0 0 1 0 0 0
```



Multiplier

```
        1 0 0 0
      x 1 0 0 1
      ---------
        1 0 0 0
      0 0 0 0
    0 0 0 0
  1 0 0 0
  -------------
  1 0 0 1 0 0 0
```



Signed Multiplication

- Recall
  - For p = a x b, if a<0 or b<0, then p < 0
  - If a<0 and b<0, then p > 0
  - Hence sign(p) = sign(a) xor sign(b)
- Hence
  - Convert multiplier, multiplicand to positive number with (n-1) bits
  - Multiply positive numbers
  - Compute sign, convert product accordingly
- Or,
  - Perform sign-extension on shifts for prev. design
  - Right answer falls out

Booth’s Encoding

- Recall grade school trick
  - When multiplying by 9:
    - Multiply by 10 (easy, just shift digits left)
    - Subtract once
  - E.g.
    - 123454 x 9 = 123454 x (10 – 1) = 1234540 – 123454
    - Converts addition of six partial products to one shift and one subtraction
- Booth’s algorithm applies same principle
  - Except no ‘9’ in binary, just ‘1’ and ‘0’
  - So, it’s actually easier!

Booth's Encoding

- Search for a run of '1' bits in the multiplier
  - E.g. '0110' has a run of 2 '1' bits in the middle
  - Multiplying by '0110' (6 in decimal) is equivalent to multiplying by 8 and subtracting twice, since 6 x m = (8 – 2) x m = 8m – 2m
- Hence, iterate right to left and:
  - Subtract multiplicand from product at first '1'
  - Add multiplicand to product after last '1'
  - Don't do either for '1' bits in the middle

Booth's Algorithm

Current bit	Bit to right	Explanation	Example	Operation
1	0	Begins run of '1'	0000111100 0	Subtract
1	1	Middle of run of '1'	0000111100 0	Nothing
0	1	End of a run of '1'	0000111100 0	Add
0	0	Middle of a run of '0'	0000111100 0	Nothing

Booth's Encoding

- Really just a new way to encode numbers
  - Normally positionally weighted as 2^n
  - With Booth, each position has a sign bit
  - Can be extended to multiple bits

<table>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>Binary</td></tr>
<tr><td>+1</td><td>0</td><td>-1</td><td>0</td><td>1-bit Booth</td></tr>
<tr><td colspan="2">+2</td><td colspan="2">-2</td><td>2-bit Booth</td></tr>
</table>

2-bits/cycle Booth Multiplier

- For every pair of multiplier bits
  - If Booth's encoding is '-2'
    - Shift multiplicand left by 1, then subtract
  - If Booth's encoding is '-1'
    - Subtract
  - If Booth's encoding is '0'
    - Do nothing
  - If Booth's encoding is '1'
    - Add
  - If Booth's encoding is '2'
    - Shift multiplicand left by 1, then add

2 bits/cycle Booth's

1 bit Booth	
00	+0
01	+M;
10	-M;
11	+0

Current	Previous	Operation	Explanation
00	0	+0;shift 2	[00] => +0, [00] => +0; 2x(+0)+(+0)=+0
00	1	+M; shift 2	[00] => +0, [01] => +M; 2x(+0)+(+M)=+M
01	0	+M; shift 2	[01] => +M, [10] => -M; 2x(+M)+(-M)=+M
01	1	+2M; shift 2	[01] => +M, [11] => +0; 2x(+M)+(+0)=+2M
10	0	-2M; shift 2	[10] => -M, [00] => +0; 2x(-M)+(+0)=-2M
10	1	-M; shift 2	[10] => -M, [01] => +M; 2x(-M)+(+M)=-M
11	0	-M; shift 2	[11] => +0, [10] => -M; 2x(+0)+(-M)=-M
11	1	+0; shift 2	[11] => +0, [11] => +0; 2x(+0)+(+0)=+0

Booth's Example

- Negative multiplicand:

  -6 x 6 = -36

  1010 x 0110, 0110 in Booth's encoding is +0-0

  Hence:

1111 1010	x 0	0000 0000
1111 0100	x –1	0000 1100
1110 1000	x 0	0000 0000
1101 0000	x +1	1101 0000
	Final Sum:	1101 1100 (-36)

Booth's Example

- Negative multiplier:

  -6 x -2 = 12

  1010 x 1110, 1110 in Booth's encoding is 00-0

  Hence:

1111 1010	x 0	0000 0000
1111 0100	x –1	0000 1100
1110 1000	x 0	0000 0000
1101 0000	x 0	0000 0000
	Final Sum:	0000 1100 (12)

Summary

- Integer multiply
  - Combinational
  - Multicycle
  - Booth's algorithm