CMPE 411
Computer Architecture

Lecture 7

Multiplier's Design

September 21, 2017

www.csee.umbc.edu/~younis/CMPE411/CMPE411.htm

Lecture's Overview

- ***Previous Lecture***
  - ➔ Constructing an Arithmetic Logic Unit

    (Different blocks and gluing them together)
  - ➔ Scaling bit operations to word sizes

    (Ripple carry adder, MIPS ALU)
  - ➔ Optimization for carry handling

    (Measuring performance, Carry lookahead)
- ***This Lecture***
  - ➔ Algorithms for multiplying unsigned numbers
  - ➔ Booth's algorithm for signed number multiplication
  - ➔ Multiple hardware design for integer multiplier

Multiply Unsigned

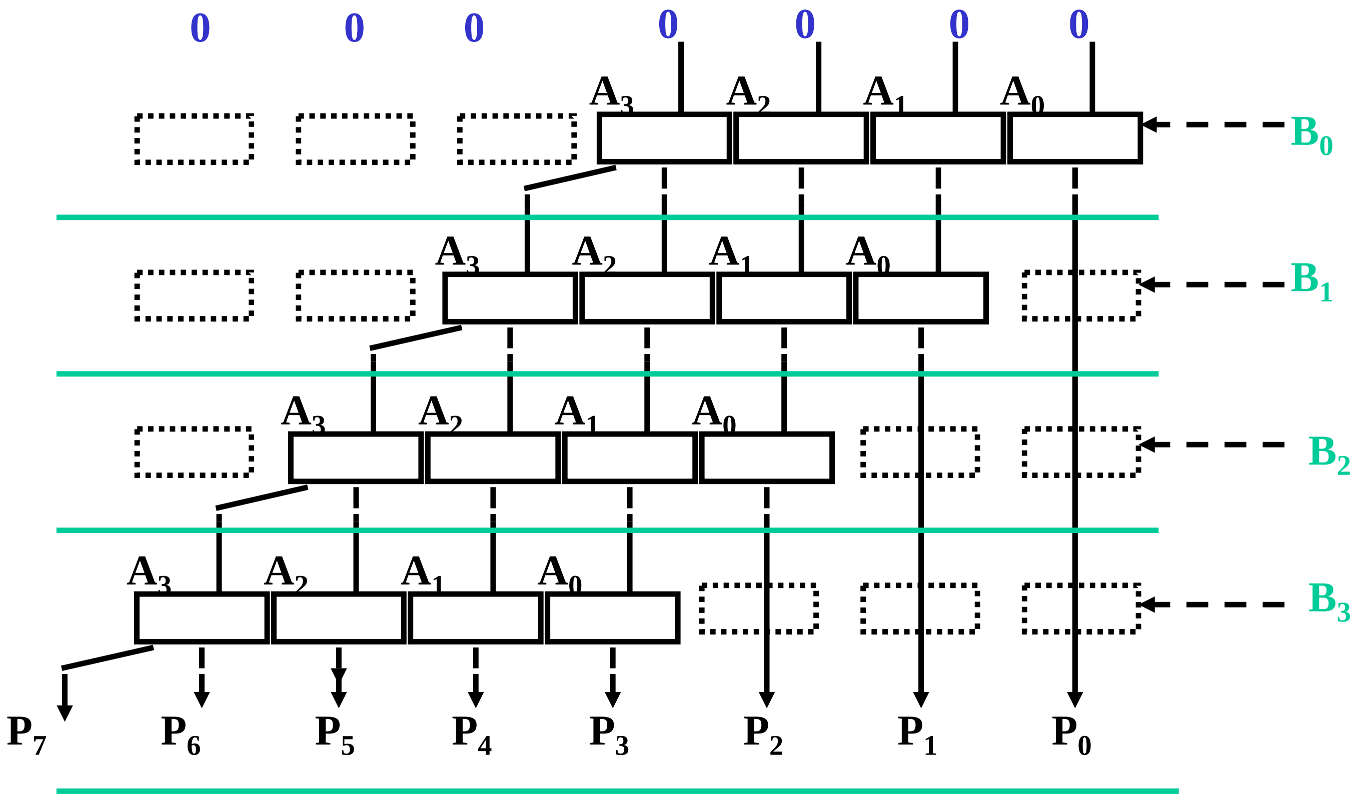
- Paper and pencil example (unsigned):

Multiplicand		**1000**
Multiplier	×	**1001**
		1000
		0000
		0000
		1000
Product		**01001000**

- m bits x n bits = m+n bit product (*Overflow ?*)
- Binary makes it easy:
  - ➔ 0 => place 0 (0 x multiplicand)
  - ➔ 1 => place a copy (1 x multiplicand)
- 4 versions of multiply hardware & algorithm:
  - ➔ successive refinement

* Slide is courtesy of Dave Patterson

Unsigned Combinational Multiplier

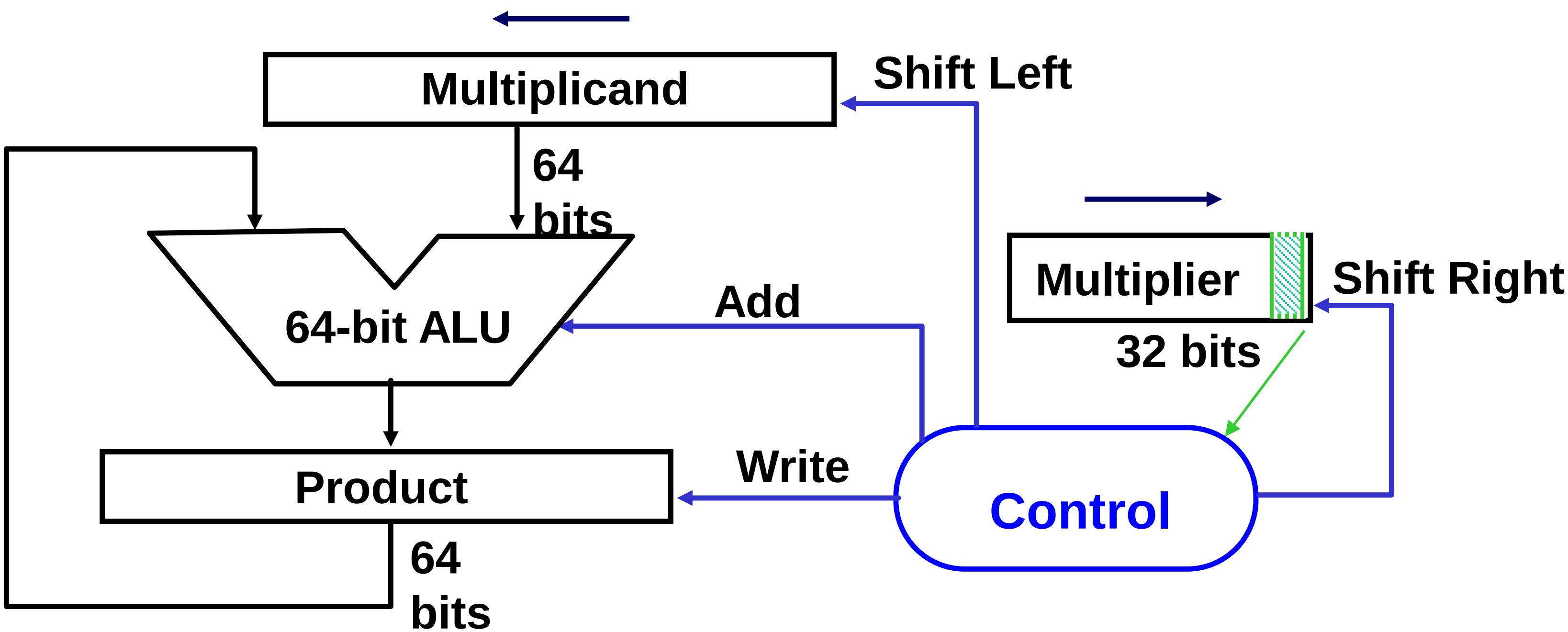


- Stage *i* accumulates A * 2^i if B_i == 1
- At each stage shift A left (x 2)
- Use next bit of B to determine whether to add in shifted multiplicand
- Accumulate 2n bit partial product at each stage

* Slide is courtesy of Dave Patterson

Unsigned shift-add multiplier (version 1)

- 64-bit Multiplicand register, 64-bit ALU, 64-bit Product register, and 32-bit Multiplier register
- The 32-bit value of the Multiplicand starts in the right half of the 64-bit register
- The Multiplier is shifted in the opposite direction of the Multiplicand shift
- The product register starts with an initial value of zero
- Control decides when to shift the Multiplicand and the Multiplier registers and when to write new value into the product register

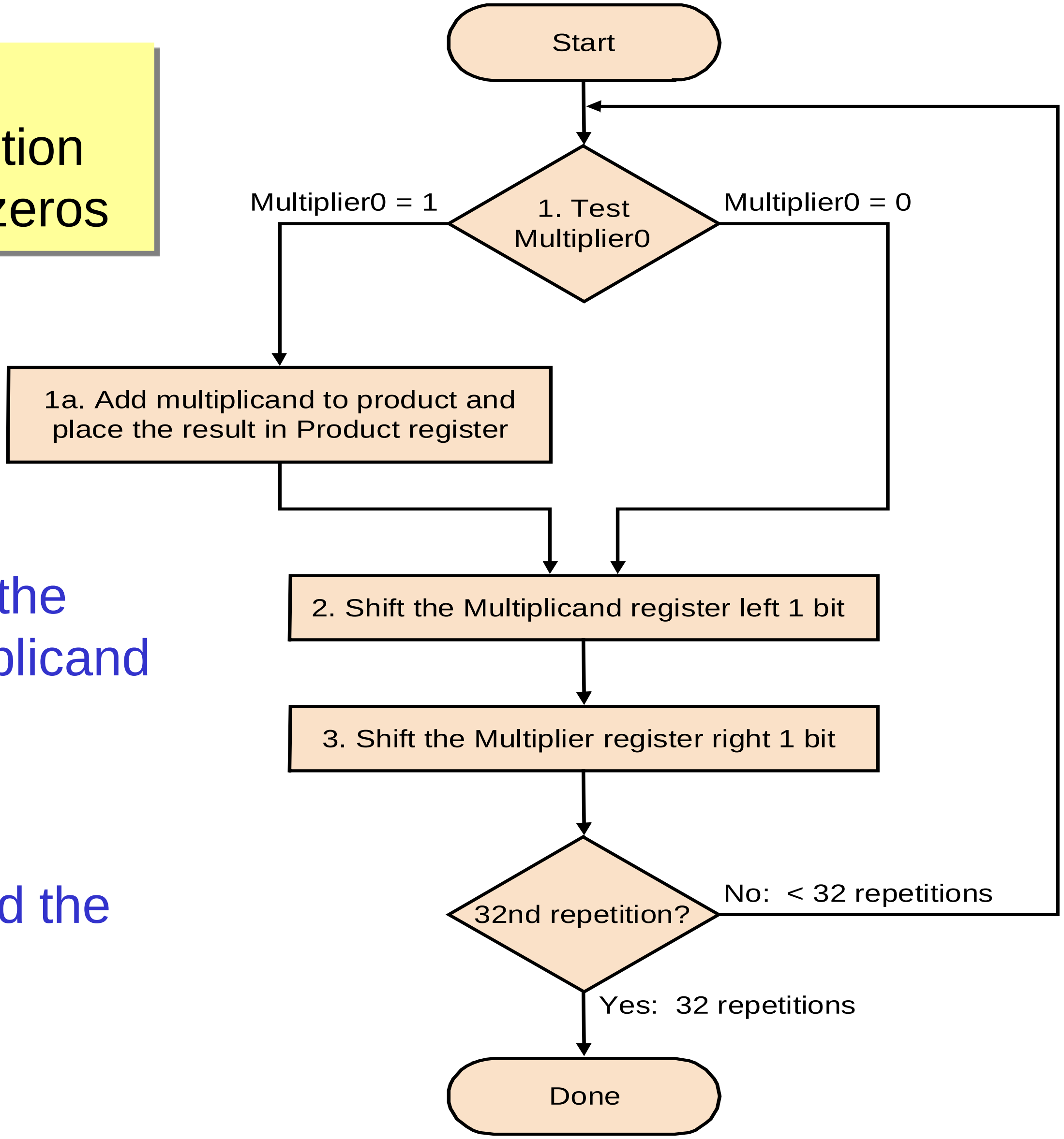


Multiplier = datapath + control

* Figure is courtesy of Dave Patterson

Multiply Algorithm Version 1

Multiplying two *n*-bit numbers needs a maximum of $2n^2$ addition operations mostly for adding zeros



- ➔ If the least significant bit of the multiplier is 1, add the multiplicand to the product
- ➔ If not, go to the next bit
- ➔ shift the multiplicand left and the multiplier right
- ➔ Repeat for 32 times

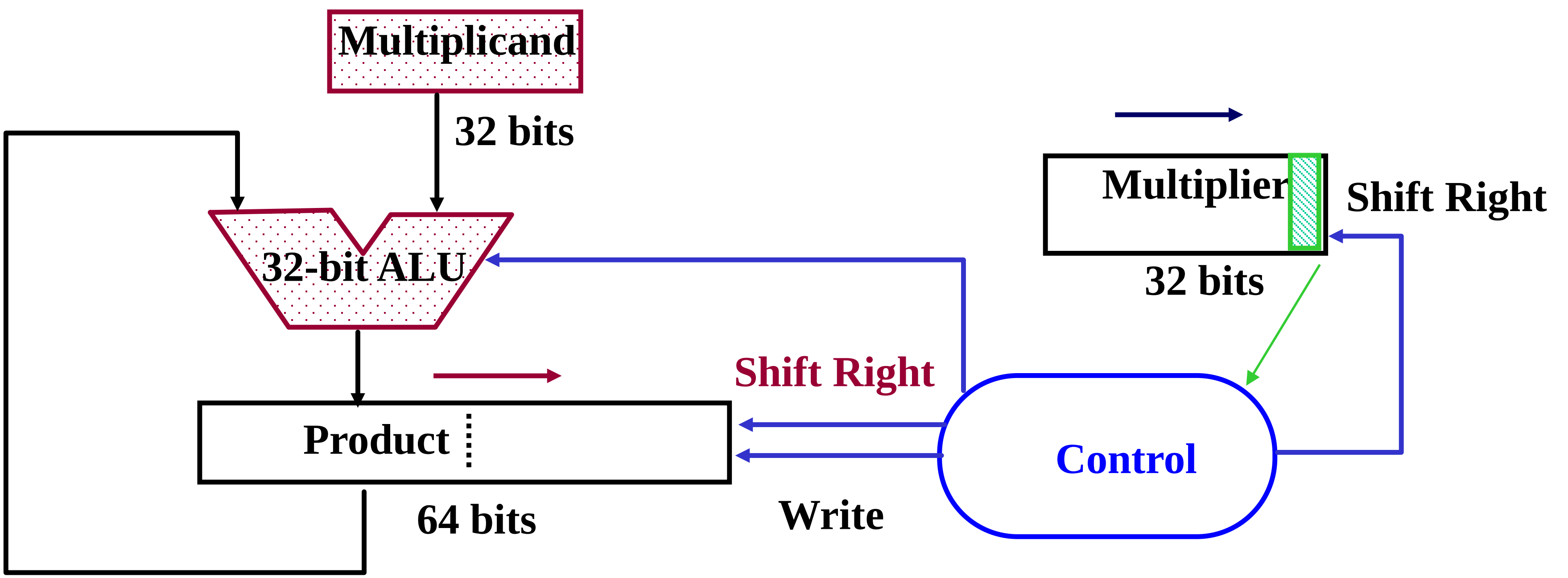
An Example

Follow the multiplication algorithm (version 1) to get the product of 2 $\times$ 3 using only 4-bit binary representation

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial value	0011	0000 0010	0000 0000
1	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1a: 0 $\Rightarrow$ no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1a: 0 $\Rightarrow$ no operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

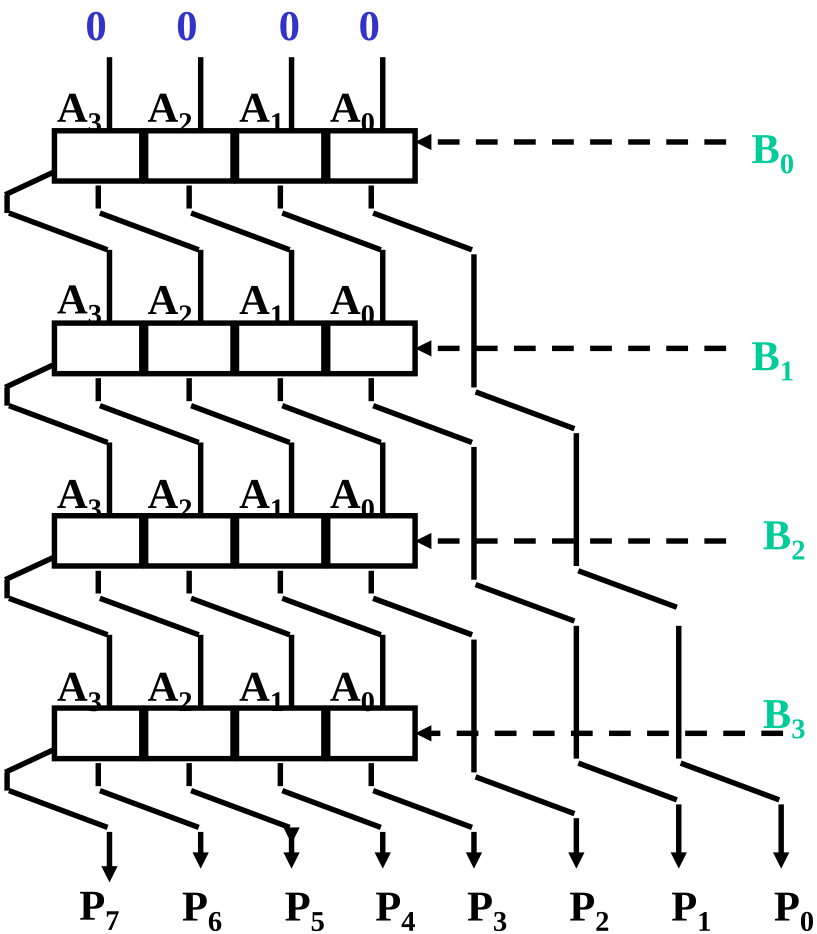
Multiply Hardware Version 2

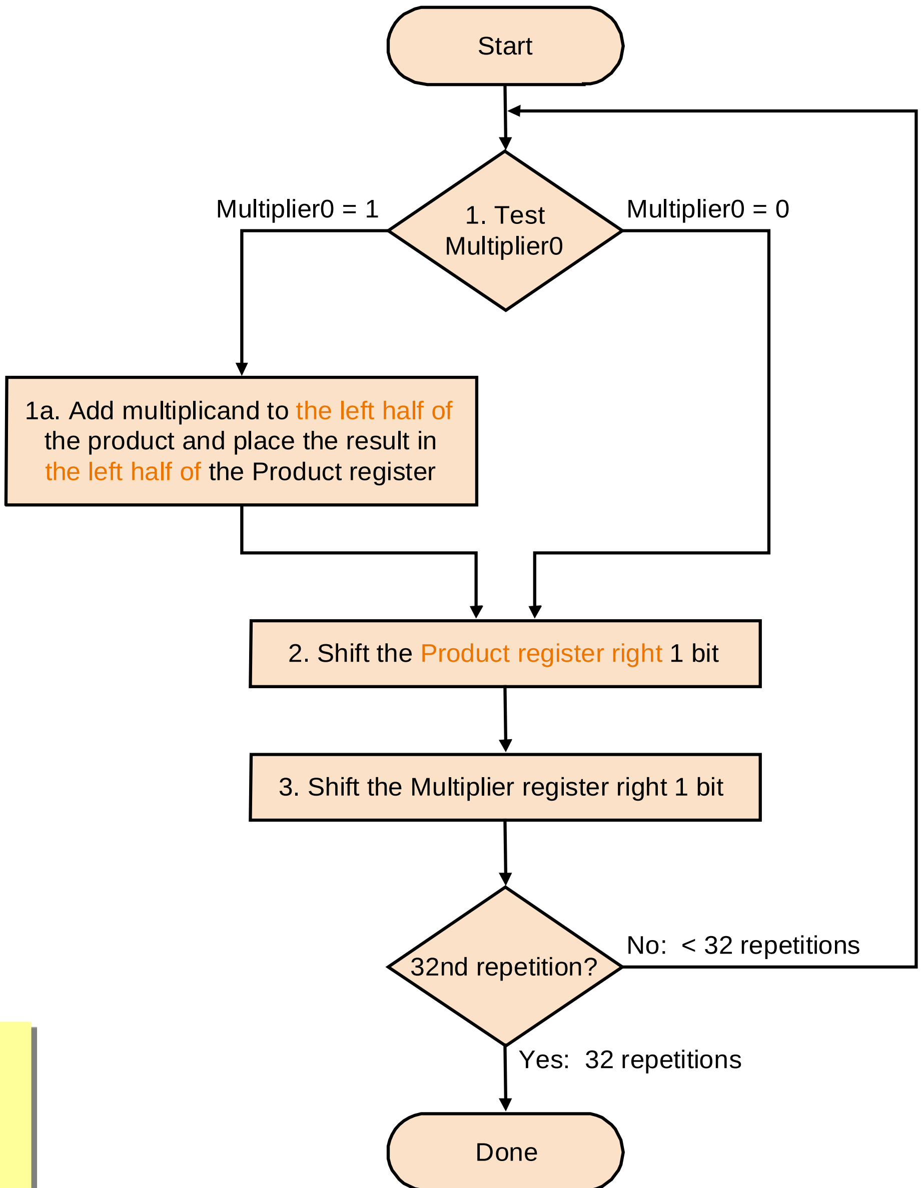
- Since half of the 64-bit Multiplicand are zeros, a 64-bit ALU looks wasteful in the first version of multiplier
- Uses only 32-bit Multiplicand register, 32-bit ALU, 64-bit Product register, and 32-bit Multiplier register
- Since the least significant bits of the product would not change, the product could be shifted to the right instead of shifting the multiplicand
- The most significant 32-bits would be used by the ALU as a result register



* Figure is courtesy of Dave Patterson

Multiply Algorithm Version 2





Multiplicand stays still and product moves right

* Figure is courtesy of Dave Patterson

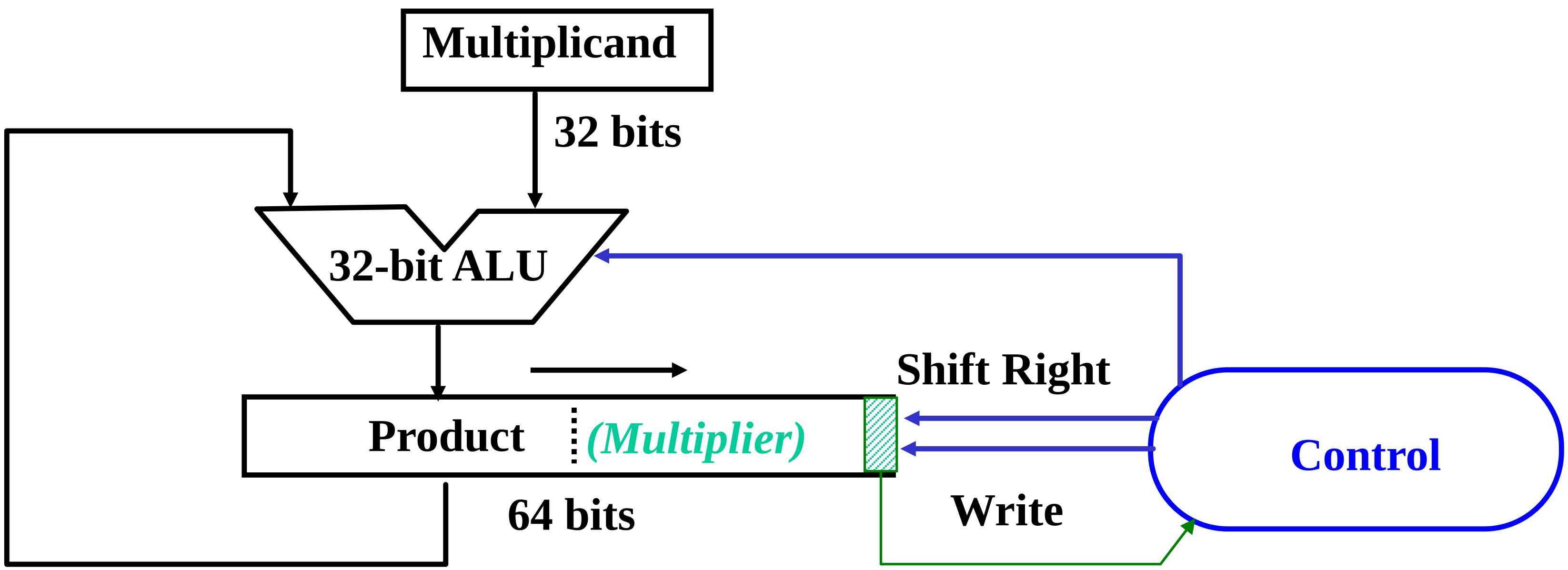
An Example

Follow the multiplication algorithm (version 2) to get the product of 2 $\times$ 3 using only 4-bit binary representation

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial value	0011	0010	0000 0000
1	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	0011	0010	0010 0000
	2: Shift right Product	0011	0010	0001 0000
	3: Shift right Multiplier	0001	0010	0001 0000
2	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	0001	0010	0011 0000
	2: Shift right Product	0001	0010	0001 1000
	3: Shift right Multiplier	0000	0010	0001 1000
3	1a: 0 $\Rightarrow$ no operation	0000	0010	0001 1000
	2: Shift right Product	0000	0010	0000 1100
	3: Shift right Multiplier	0000	0010	0000 1100
4	1a: 0 $\Rightarrow$ no operation	0000	0010	0000 1100
	2: Shift right Product	0000	0010	0000 0110
	3: Shift right Multiplier	0000	0010	0000 0110

Multiply Hardware Version 3

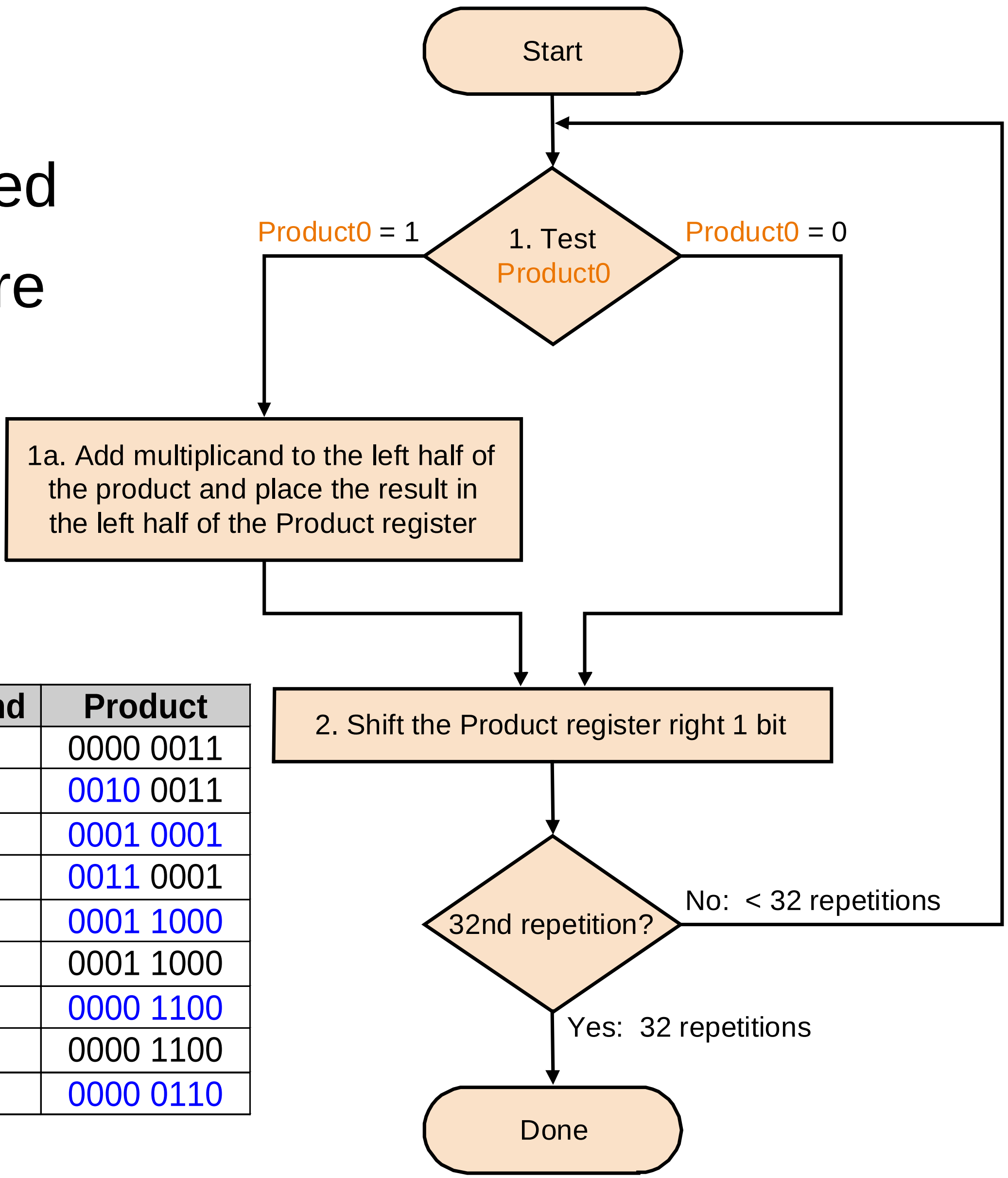
- Product register wastes space that exactly matches size of multiplier
  ⇒ combine Multiplier register and Product register
- Uses only 32-bit Multiplicand register, 32-bit ALU, 64-bit Product register, and 0-bit Multiplier register
- Shifting the product register would remove the least significant bit which is already used in the multiplication
- The most significant 32-bits are still being used by ALU as a result register



* Figure is courtesy of Dave Patterson

Multiply Algorithm Version 3

- ➔ 2 steps per bit because Multiplier & Product combined
- ➔ MIPS registers Hi and Lo are left and right half of Product



Iteration	Step	Multiplicand	Product
0	Initial value	0010	0000 0011
1	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	0010	0010 0011
	2: Shift right Product	0010	0001 0001
2	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	0010	0011 0001
	2: Shift right Product	0010	0001 1000
3	1a: 0 $\Rightarrow$ no operation	0010	0001 1000
	2: Shift right Product	0010	0000 1100
4	1a: 0 $\Rightarrow$ no operation	0010	0000 1100
	2: Shift right Product	0010	0000 0110

The product of 2 $\times$ 3

Multiplying Signed Number

❶ Easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps)

❷ Apply definition of 2's complement $\Rightarrow$ need to sign-extend partial products

Example: multiply 1001 (-7) by 0010 (+2)

Iteration	Step	Multiplicand	Product
0	Initial value	1001	0000 0010
1	1a: 0 $\Rightarrow$ no operation	1001	0000 0010
	2: Shift right Product	1001	0000 0001
2	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	1001	1001 0001
	2: Shift right Product	1001	1100 1000
3	1a: 0 $\Rightarrow$ no operation	1001	1100 1000
	2: Shift right Product	1001	1110 0100
4	1a: 0 $\Rightarrow$ no operation	1001	1110 0100
	2: Shift right Product	1001	1111 0010

Does it work for all cases?

Multiply 0111 (+7) by 1110 (-2)

Iteration	Step	Multiplicand	Product
0	Initial value	0111	0000 1110
1	1a: 0 ⇒ no operation	0111	0000 1110
	2: Shift right Product	0111	0000 0111
2	1a: 1 ⇒ Prod = Prod + Mcand	0111	0111 0111
	2: Shift right Product	0111	0011 1011
3	1a: 0 ⇒ Prod = Prod + Mcand	0111	1010 1011
	2: Shift right Product	0111	1101 0101
4	1a: 0 ⇒ Prod = Prod + Mcand	0111	0100 0101
	2: Shift right Product	0111	0010 0010

Iteration	Step	Multiplicand	Product
0	Initial value	0111	1111 1110
1	1a: 0 ⇒ no operation	0111	1111 1110
	2: Shift right Product	0111	1111 0111
2	1a: 1 ⇒ Prod = Prod + Mcand	0111	0110 0111
	2: Shift right Product	0111	0011 0011
3	1a: 0 ⇒ Prod = Prod + Mcand	0111	1010 0011
	2: Shift right Product	0111	1101 0001
4	1a: 0 ⇒ Prod = Prod + Mcand	0111	0110 0001
	2: Shift right Product	0111	0011 0000

Multiply 1111 (-7) by 1110 (-2)

Iteration	Step	Multiplicand	Product
0	Initial value	1001	0000 1110
1	1a: 0 ⇒ no operation	1001	0000 1110
	2: Shift right Product	1001	0000 0111
2	1a: 1 ⇒ Prod = Prod + Mcand	1001	1001 0111
	2: Shift right Product	1001	1100 1011
3	1a: 0 ⇒ Prod = Prod + Mcand	1001	0101 0011
	2: Shift right Product	1001	0010 1001
4	1a: 0 ⇒ Prod = Prod + Mcand	1001	1011 0100
	2: Shift right Product	1001	1101 0010

Iteration	Step	Multiplicand	Product
0	Initial value	1001	1111 1110
1	1a: 0 ⇒ no operation	1001	1111 1110
	2: Shift right Product	1001	1111 0111
2	1a: 1 ⇒ Prod = Prod + Mcand	1001	1000 0111
	2: Shift right Product	1001	1100 0011
3	1a: 0 ⇒ Prod = Prod + Mcand	1001	0101 0011
	2: Shift right Product	1001	0010 1001
4	1a: 0 ⇒ Prod = Prod + Mcand	1001	1011 0100
	2: Shift right Product	1001	1101 0010

Multiplying Signed Number

❶ Easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps)

❷ Apply definition of 2's complement ⇒ need to sign-extend partial products

Example: multiply 1001 (-7) by 0010 (+2)

Iteration	Step	Multiplicand	Product
0	Initial value	1001	0000 0010
1	1a: 0 ⇒ no operation	1001	0000 0010
	2: Shift right Product	1001	0000 0001
[illegible]	[illegible]	[illegible]	[illegible]1
	[illegible]	[illegible]	[illegible]0
3	1a: 0 ⇒ no operation	1001	1100 1000
	2: Shift right Product	1001	1110 0100
4	1a: 0 ⇒ no operation	1001	1110 0100
	2: Shift right Product	1001	1111 0010

Does NOT it work for all cases?

❸ Booth's Algorithm is elegant way to multiply signed numbers using same hardware as before and save cycles

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)
+     0010        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

- Booth observed that there are multiple ways to compute a product with the ability to add and subtract

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

* Slide is courtesy of Dave Patterson

Intuition behind Booth's Algorithm

- Which of the following decimal multiplications is easier?

  789634×10001 and 789634×9999

- How about doing the second multiplication as follows:

  $789634 \times 9999 = 789634 \times (10000-1)$

- Let's consider binary, which is of the following involves fewer additions
  - ➔ 1101×0100 ==> needs 1 8-bit numbers addition = 8 1-bit addition
  - ➔ 1101×0111 ==> needs 3 8-bit numbers addition = 24 1-bit addition
  - ➔ $1101 \times 0111 = 1101 \times (1000-0001)$

    ==> needs 1 8-bit addition and 1 8-bit subtraction

- Booth's observed and proved that this can be partially applied to multiplication of long binary numbers

- **Advantage:**
  - ➔ Fast multiplication (for consecutive 0's or 1's in the multiplier).
  - ➔ Handling of signed multiplication as well

- For more about the theoretical basis of Booth Algorithm, you may check: http://fourier.eng.hmc.edu/e85_old/lectures/arithmetic_html/node10.html

Booth's Algorithm

Current bit	Bit to the right	Explanation	Example
1	0	Beginning of a run of 1s	00001111000
1	1	Middle of a run of 1s	00001111000
0	1	End of a run of 1s	00001111000
0	0	Middle of a run of 0s	00001111000

End of run   Middle of run   Beginning of run

0	1	1 1	1	0

❶ Depending on the current and previous bits, do one of the following

00: Middle of a string of 0s ⇒ no arithmetic operation

01: End of a string of 1s ⇒ add the multiplicand to the left half of the product

10: Beginning of a string of 1s ⇒ subtract multiplicand from left half of the product

11: Middle of a string of 1s ⇒ no arithmetic operation

❷ Shift the Product register to the right for 1 bit

Booth's algorithm works for both signed and unsigned numbers

Example (unsigned numbers)

Compare the multiplication algorithm (version 3) and Booth's algorithm applied to getting the product of 2 $\times$ 6 using only 4-bit binary representation

Multiplicand	Original Algorithm		Booth's Algorithm	
	Step	Product	Step	Product
0010	Initial value	0000 0110	Initial value	0000 0110 0
0010	1a: 0 $\Rightarrow$ no operation	0000 0110	1a: 00 $\Rightarrow$ no operation	0000 0110 0
0010	2: Shift right Product	0000 0011	2: Shift right Product	0000 0011 0
0010	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	0010 0011	1a: 10 $\Rightarrow$ Prod = Prod - Mcand	1110 0011 0
0010	2: Shift right Product	0001 0001	2: Shift right Product	1111 0001 1
0010	1a: 1 $\Rightarrow$ Prod = Prod + Mcand	0011 0001	1a: 11 $\Rightarrow$ no operation	1111 0001 1
0010	2: Shift right Product	0001 1000	2: Shift right Product	1111 1000 1
0010	1a: 0 $\Rightarrow$ no operation	0001 1000	1a: 01 $\Rightarrow$ Prod = Prod + Mcand	0001 1000 1
0010	2: Shift right Product	0000 1100	2: Shift right Product	0000 1100 0

- Booth's algorithm uses both the current bit and the previous bit to determine its course of action
- Extend the sign when shifting to preserve the sign (*arithmetic right shift*)

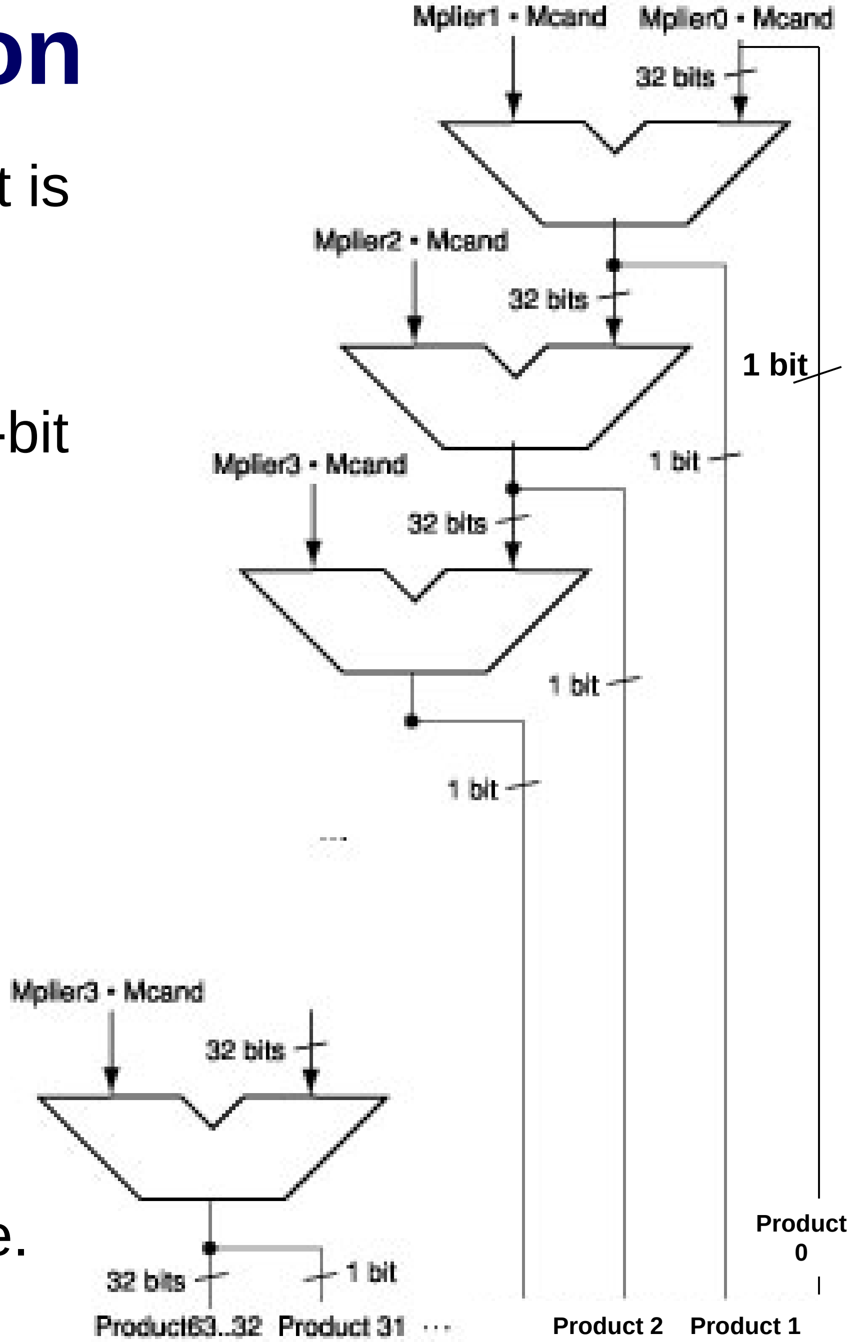
Example (signed numbers)

Follow Booth's algorithm to get the product of 2 $\times$ -3 using only 4-bit binary representation

Iteration	Step	Multiplicand	Product
0	Initial value	0010	0000 1101 0
1	1a: 10 $\Rightarrow$ Prod = Prod - Mcand	0010	1110 1101 0
	2: Shift right Product	0010	1111 0110 1
2	1a: 01 $\Rightarrow$ Prod = Prod + Mcand	0010	0001 0110 1
	2: Shift right Product	0010	0000 1011 0
3	1a: 10 $\Rightarrow$ Prod = Prod - Mcand	0010	1110 1011 0
	2: Shift right Product	0010	1111 0101 1
4	1a: 11 $\Rightarrow$ no operation	0010	1111 0101 1
	2: Shift right Product	0010	1111 1010 1

Fast Multiplication

- Whether to add the multiplicand or not is known by looking at the individual multiplier's bits
- To multiply fast, one can provide a 32-bit adder for every bit in the multiplier
- The multiplier bit is ANDed with the multiplicand (each of its bits)



Why is this faster?

- No data storage required (no need for clock)
- Clock based operation slows down the multiplication since we are effectively adding once per clock cycle.

Conclusion

- ***Summary***
  - ➔ Algorithms for multiplying unsigned numbers
    (Evolution of optimization, complexity)
  - ➔ Booth's algorithm for signed number multiplication
    (Different approach to multiplying, 2-bit based operation selection)
  - ➔ Multiple hardware design for integer multiplier
    (Hardware cost-driven optimization, fast multiplication)

- ***Next Lecture***
  - ➔ Algorithms for dividing unsigned numbers
  - ➔ Handling of sign while performing a division
  - ➔ Hardware design for integer division

Read section 3.3 in 5th Ed., or section 3.3 in 4th Ed. of the textbook