CMSC 411 | Computer Architecture

Lecture 11: **Multiplier Design**

Dr. Ergun Simsek

Facit C1-13
https://www.youtube.com/watch?v=zJ3q4zjCKnM

Overview

Previous Lecture

- Constructing an Arithmetic Logic Unit
- Addition, subtraction, slt, and branching

This Lecture

- Algorithms for multiplying unsigned numbers
- Booth’s algorithm for signed number multiplication
- Multiple hardware design for integer multiplier

How Humans Multiply

```
Multiplicand        1000        m bits
Multiplier         x1001        n bits
                    1000
                   0000
                  0000
         +       1000
              ---------
Product         01001000        m+n bits
```

Computer does the almost same thing

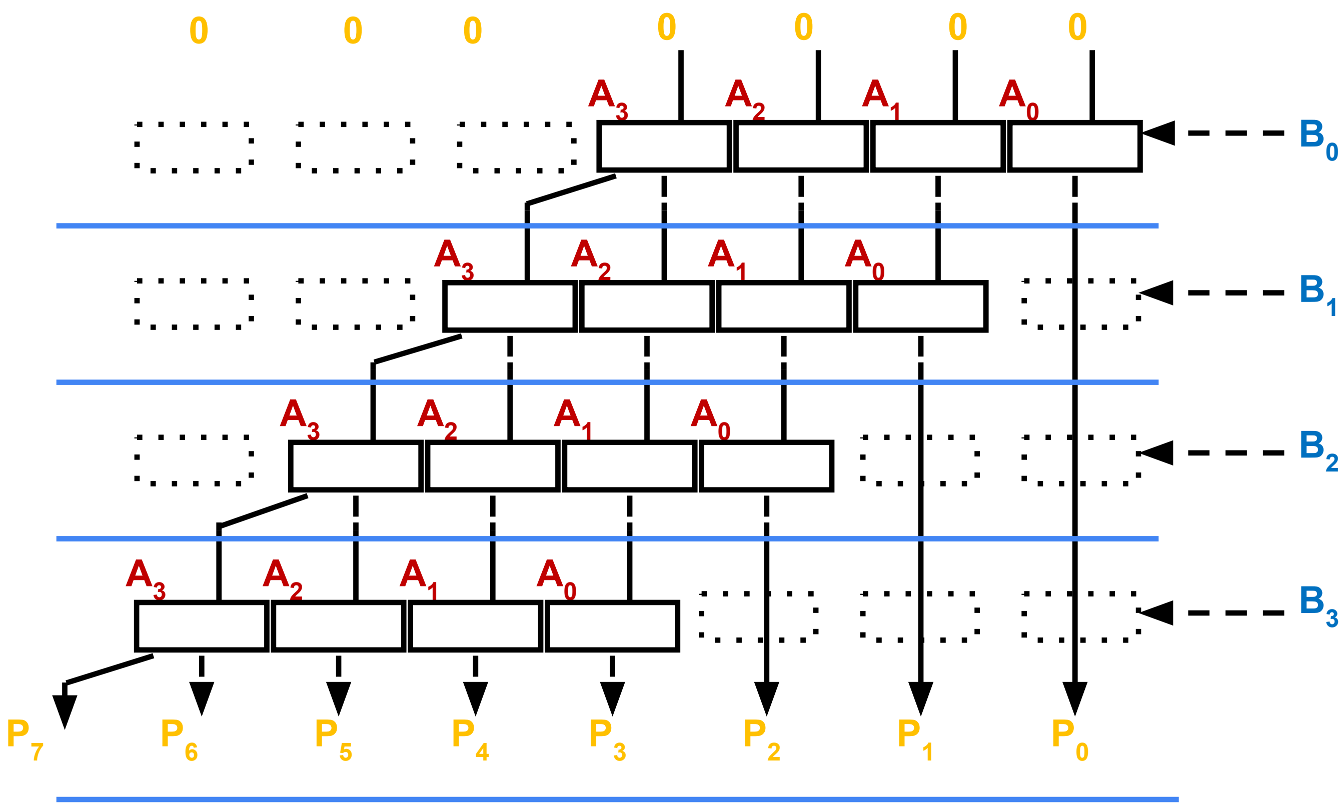
Binary makes things easy

0 => place 0   (0 x multiplicand)
1 => place a copy (1 x multiplicand)

Unsigned mult has no overflow risk, so let's forget about the sign problem and focus on some mult designs

How would you design it?

Unsigned Combinational Multiplier

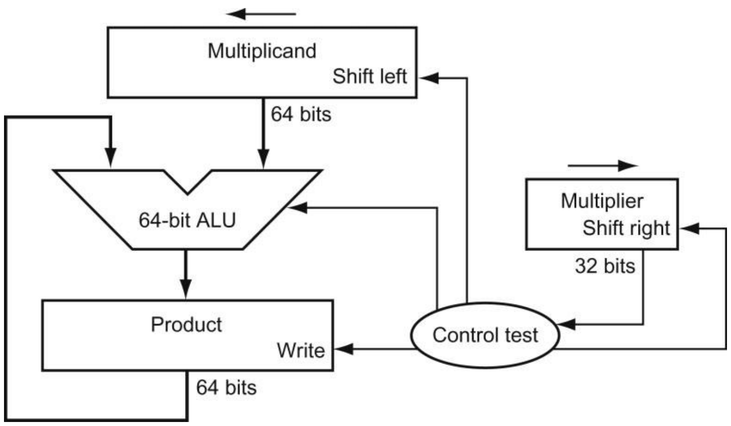


- Stage i accumulates A * 2^i if B_i == 1
- At each stage shift A left (x 2)
- Use next bit of B to determine whether to add in shifted multiplicand
- Accumulate 2n bit partial product at each stage

Unsigned shift-add multiplier (version 1)

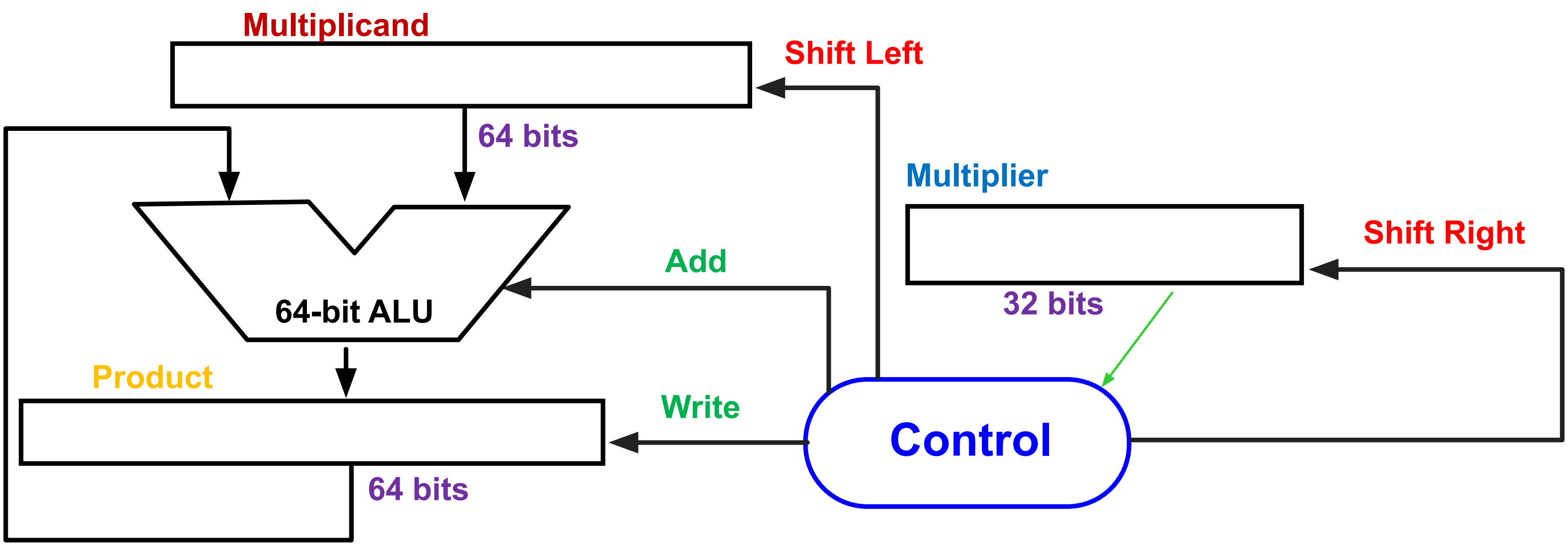
Assume we have a

- 64-bit multiplicand register,
- 64-bit ALU,
- 64-bit product register, and
- 32-bit multiplier register



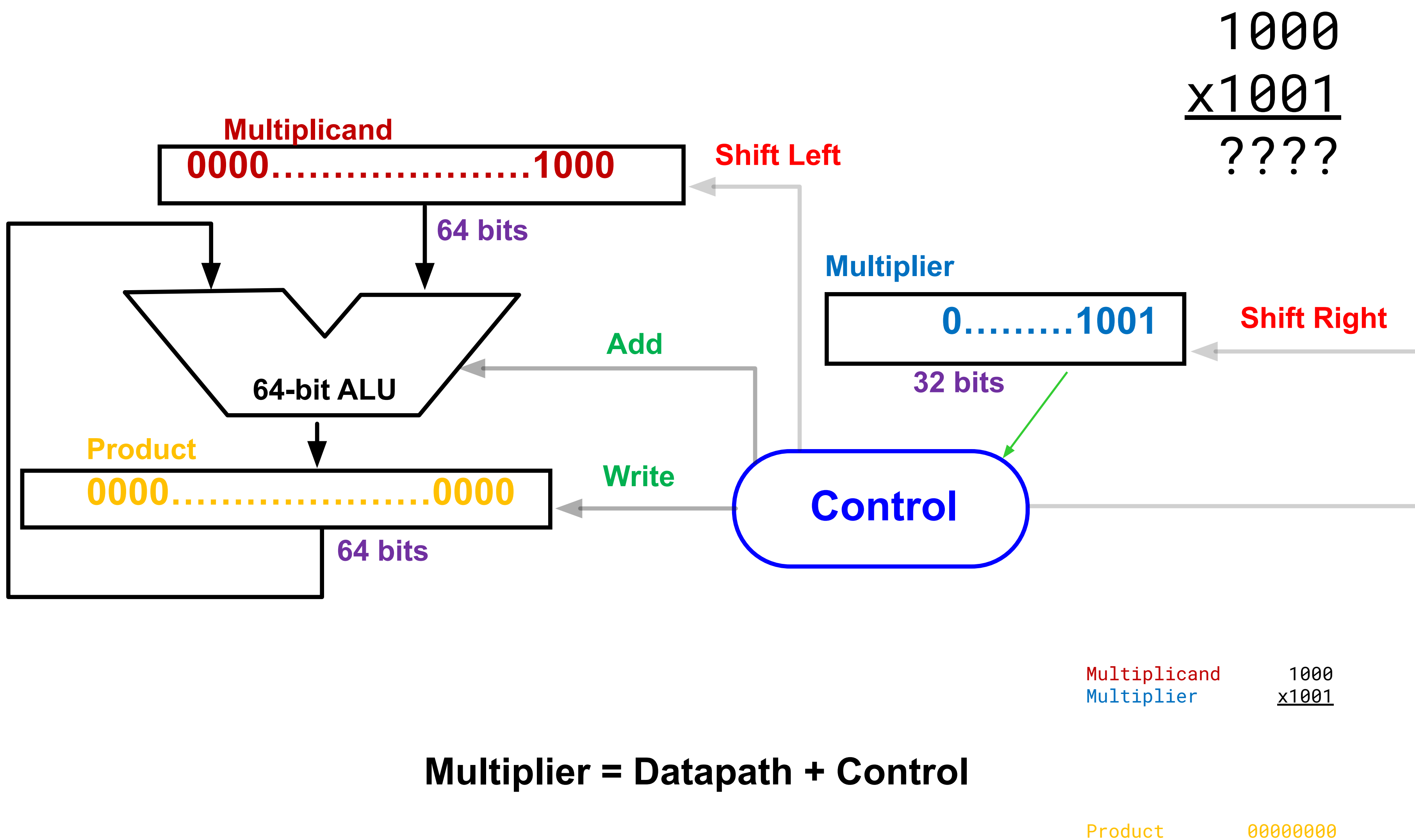
- The 32-bit value of the multiplicand starts in the right half of the 64-bit register
- The multiplier is shifted in the opposite direction of the multiplicand shift
- The product register starts with an initial value of zero
- Control decides when to shift the multiplicand and the multiplier registers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

Unsigned shift-add multiplier (version 1)



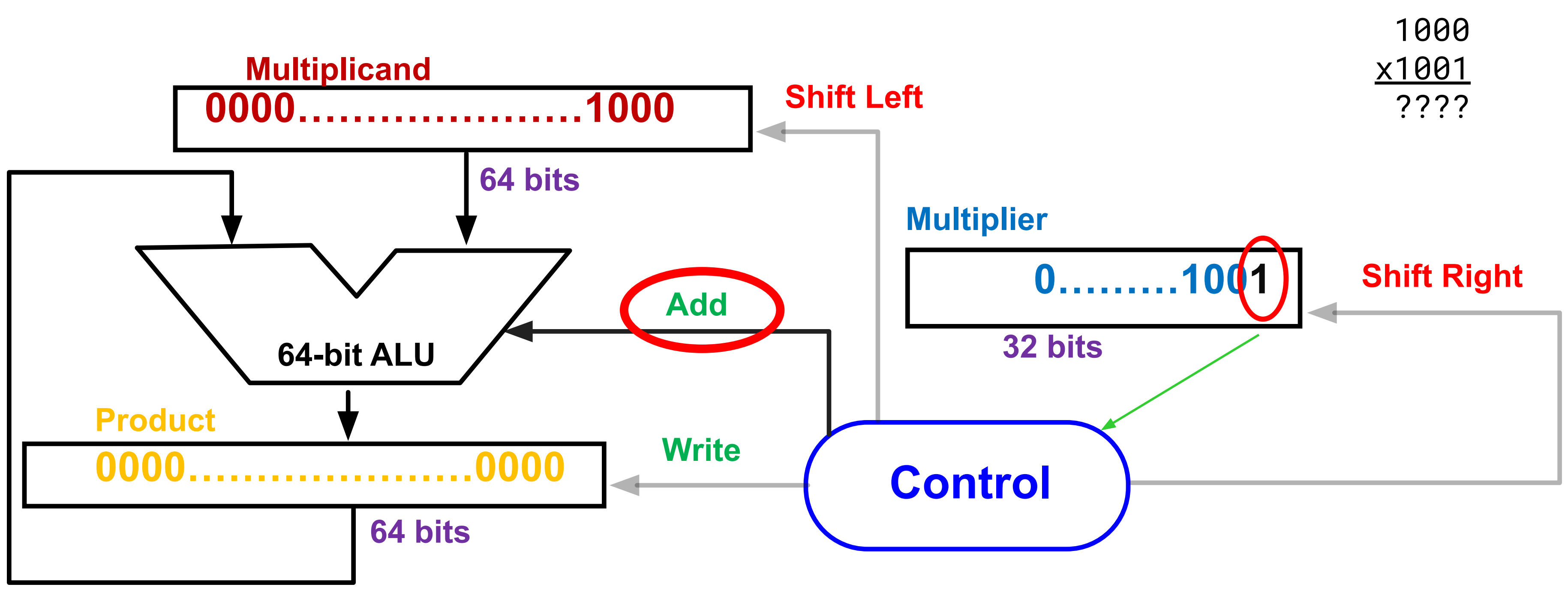
Multiplier = Datapath + Control

Unsigned shift-add multiplier (version 1)



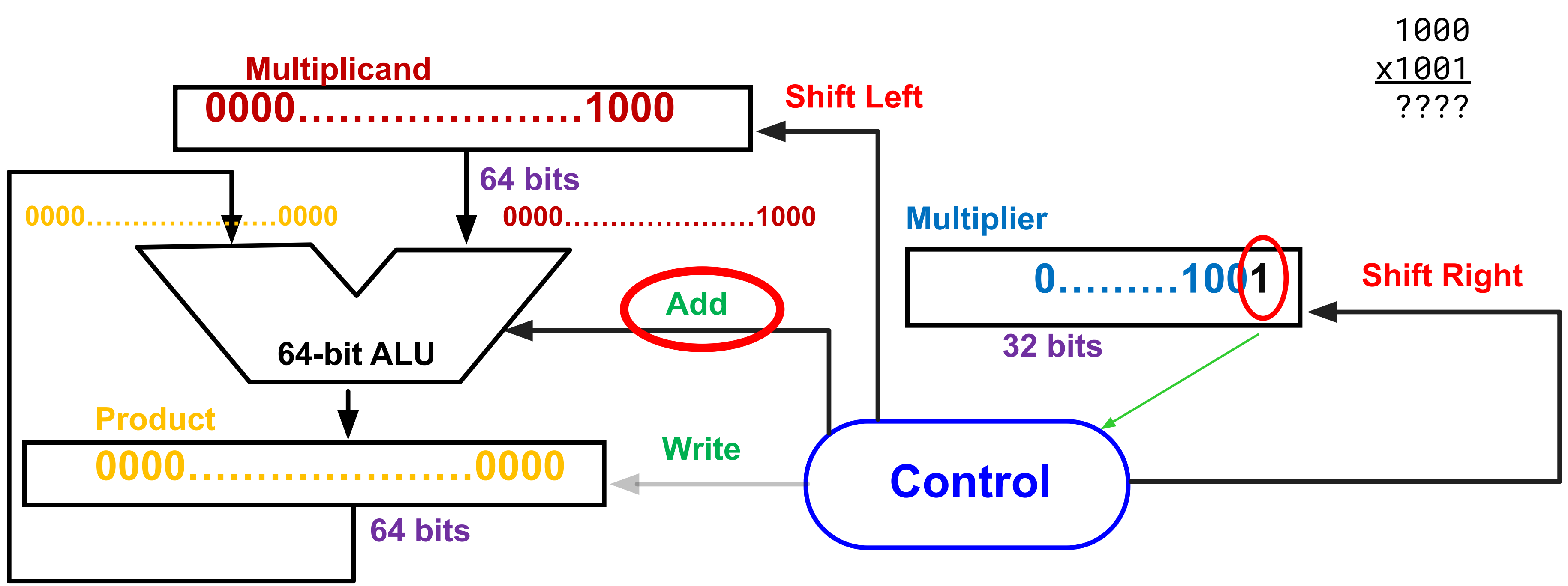
Multiplier = Datapath + Control

Unsigned shift-add multiplier (version 1)



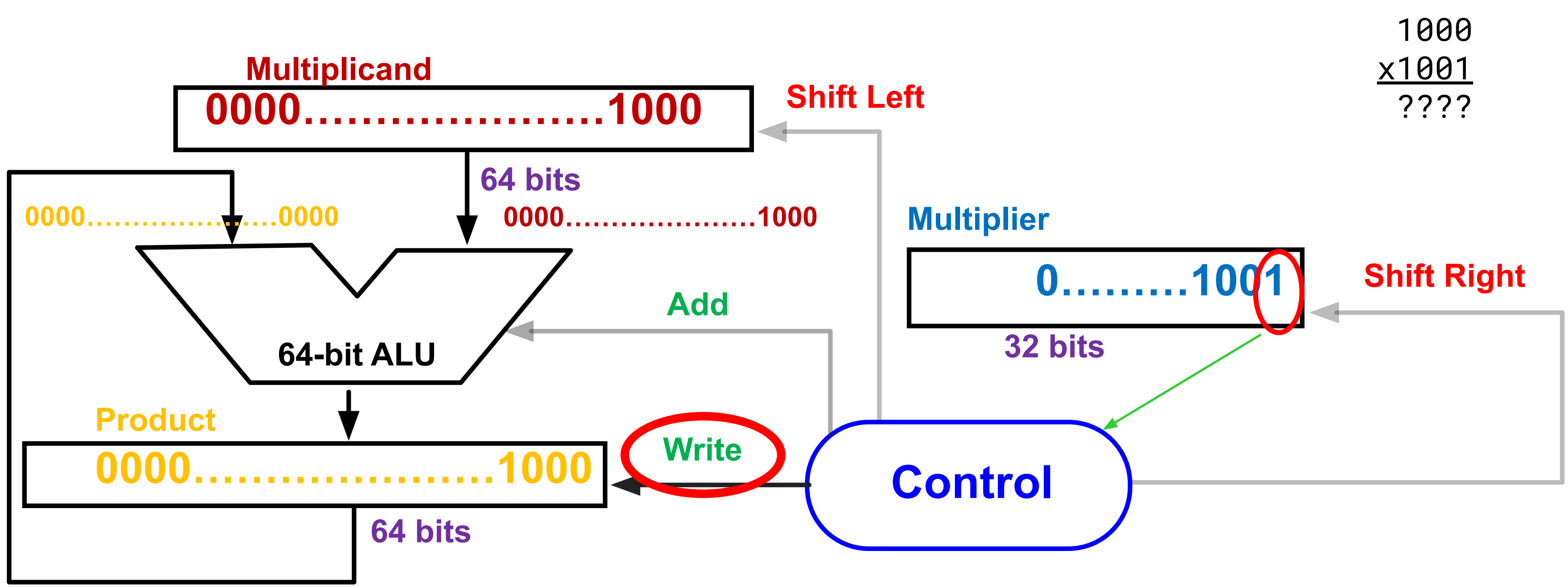
Multiplier = Datapath + Control

Unsigned shift-add multiplier (version 1)



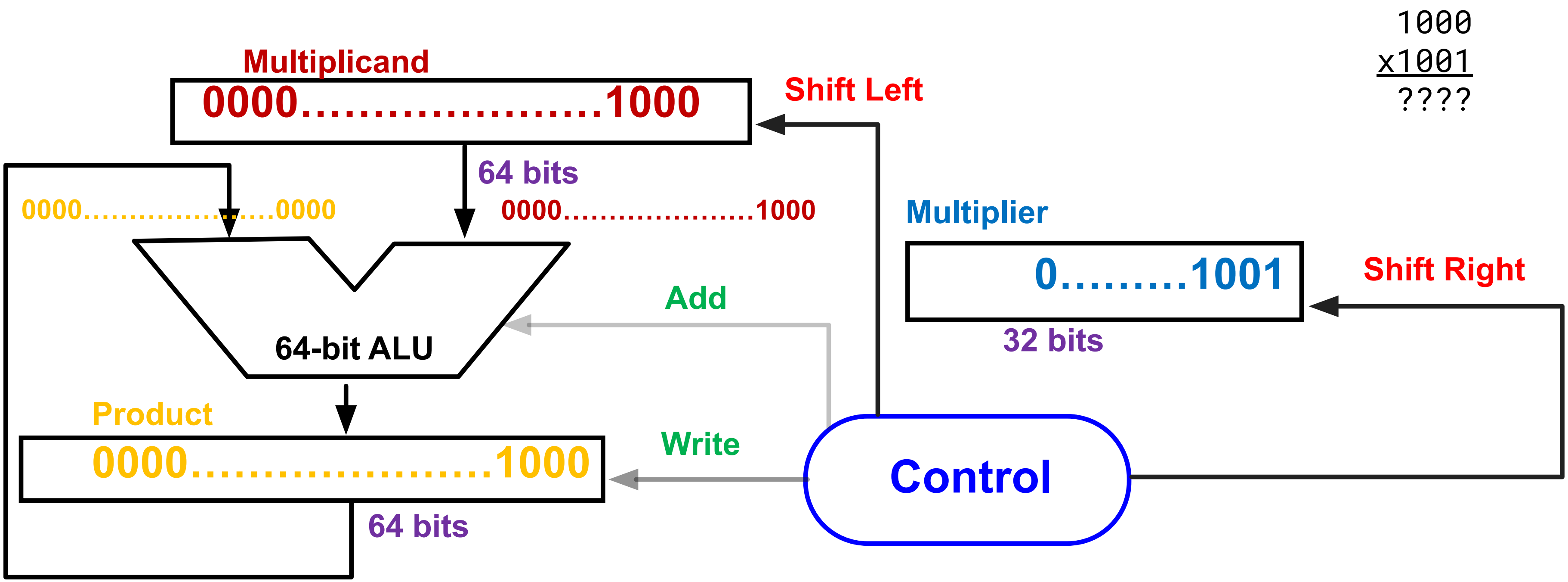
Multiplier = Datapath + Control

Unsigned shift-add multiplier (version 1)



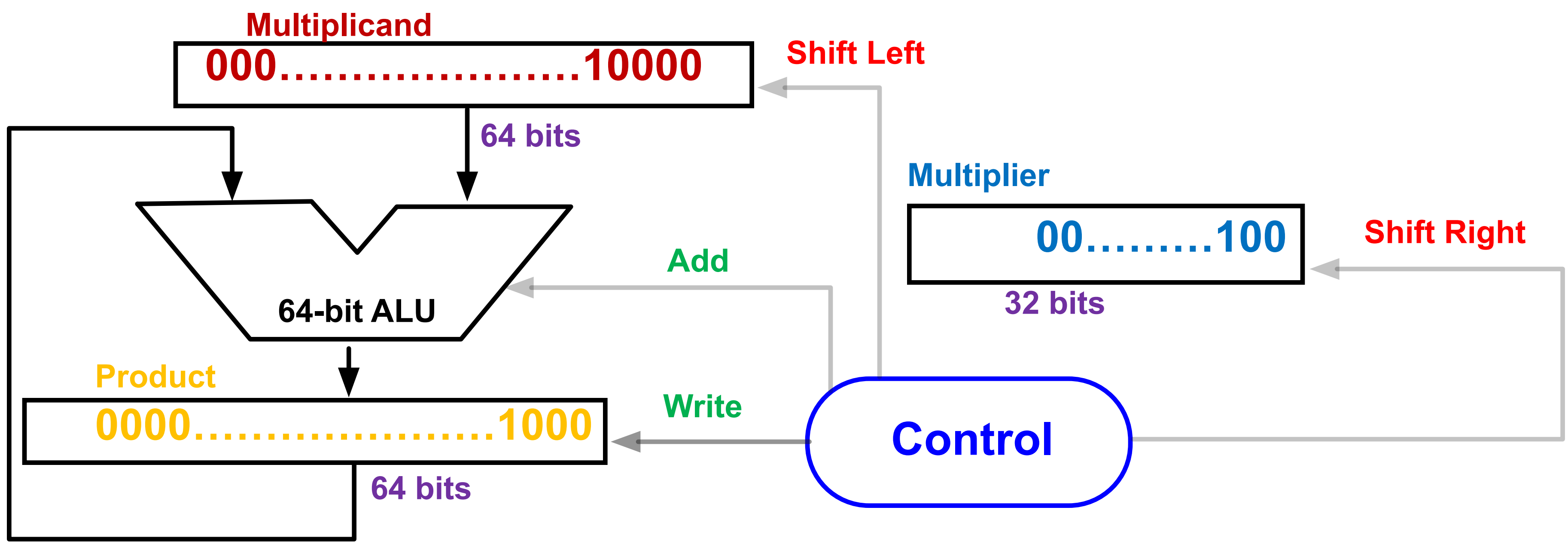
Multiplier = Datapath + Control

Unsigned shift-add multiplier (version 1)



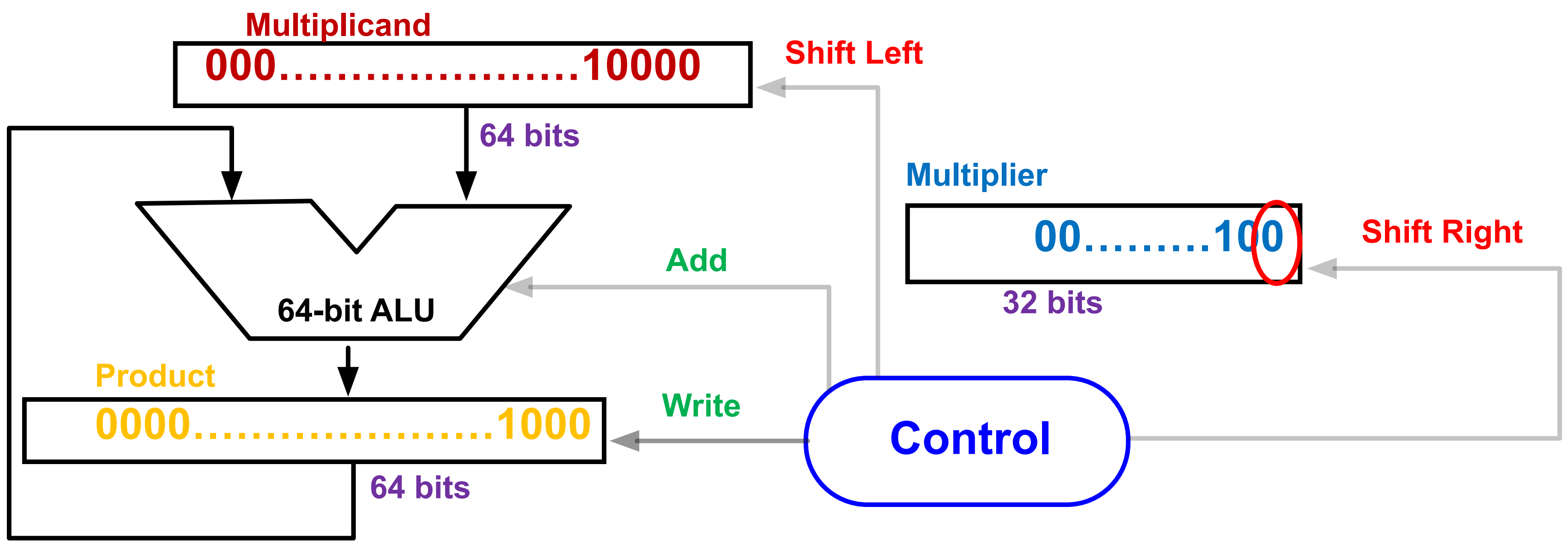
Multiplier = Datapath + Control

Unsigned shift-add multiplier (version 1)



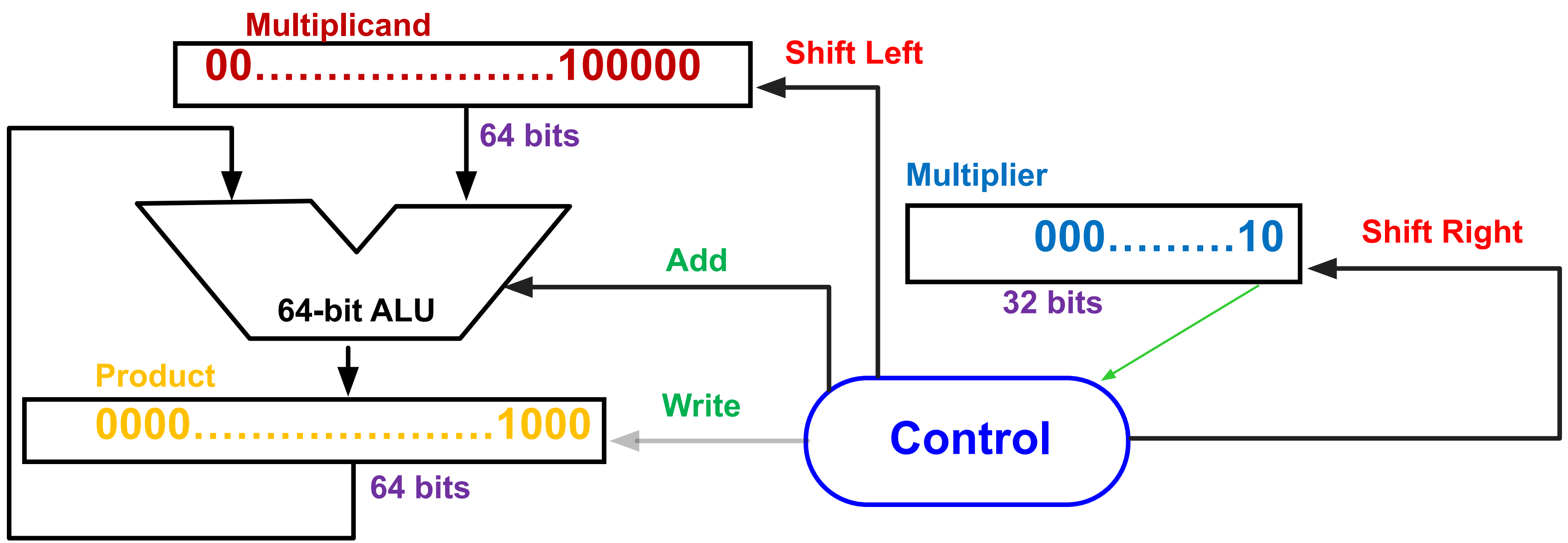
Multiplier = Datapath + Control

Unsigned shift-add multiplier (version 1)



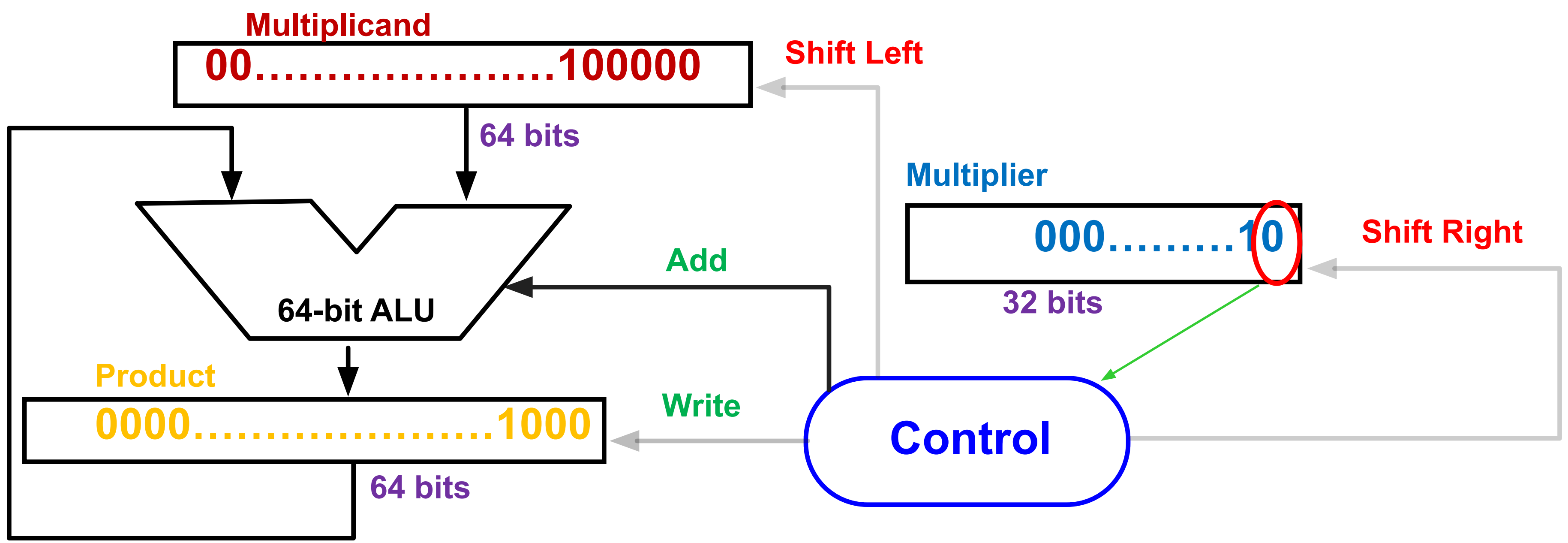
Multiplier = Datapath + Control

Unsigned shift-add multiplier (version 1)



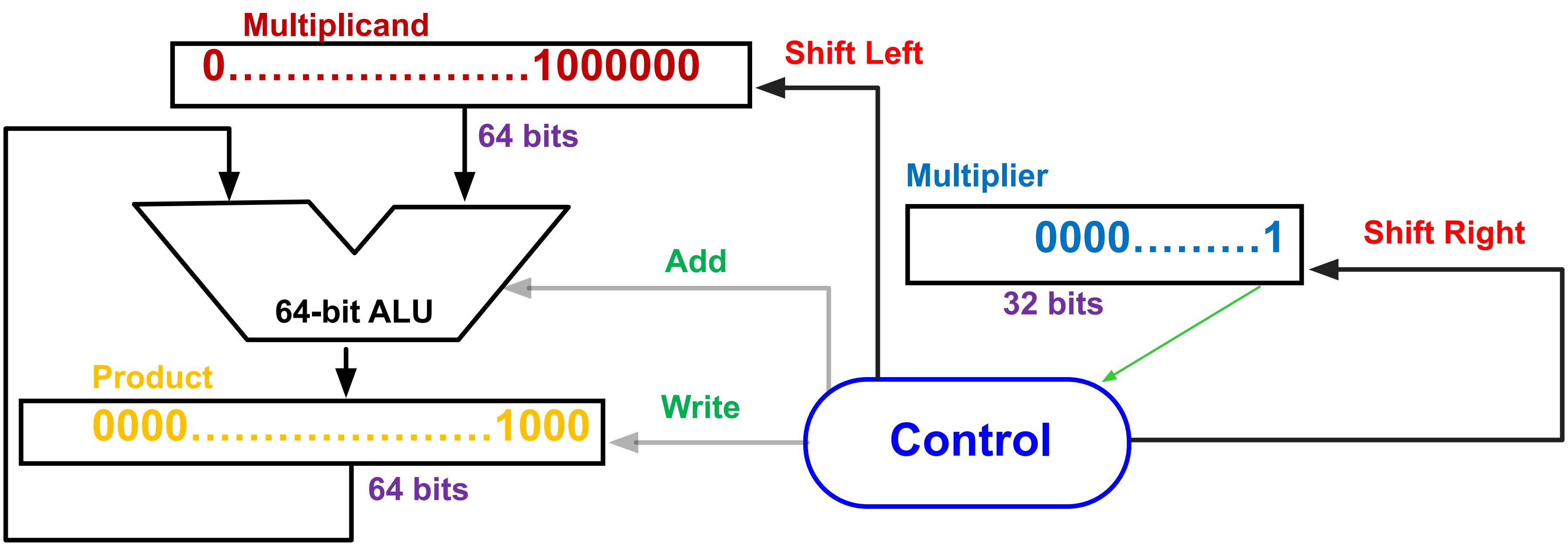
Multiplier = Datapath + Control

Unsigned shift-add multiplier (version 1)



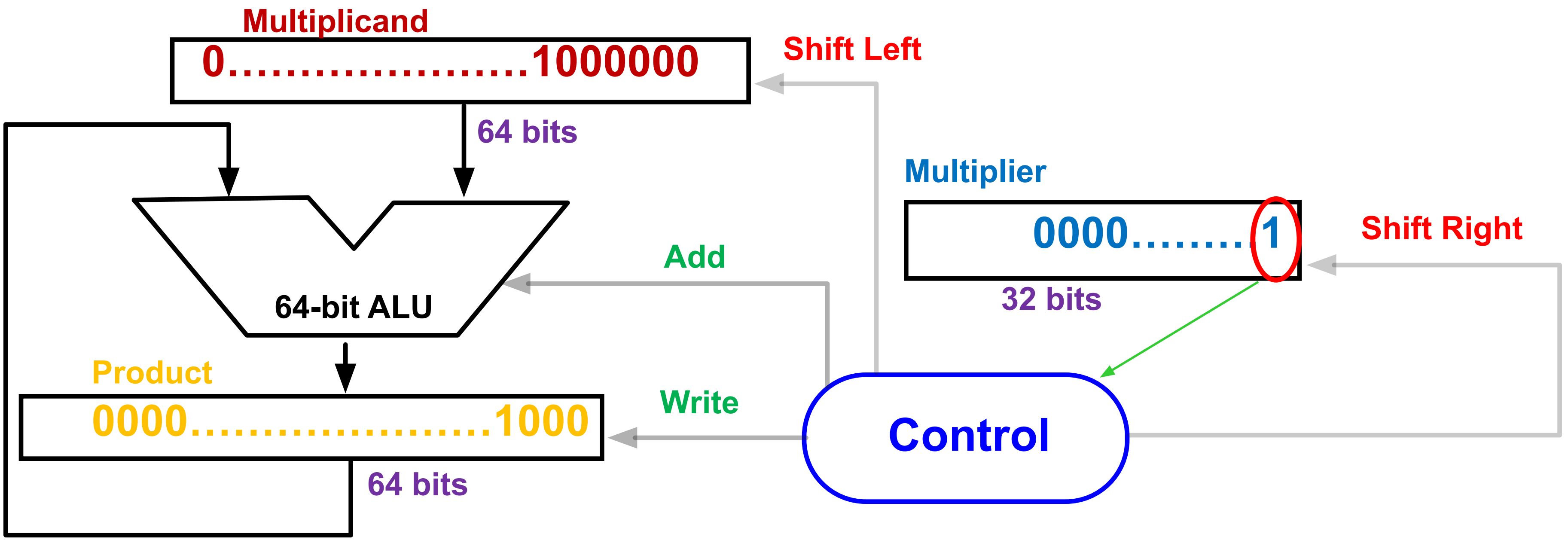
Multiplier = Datapath + Control

Unsigned shift-add multiplier (version 1)



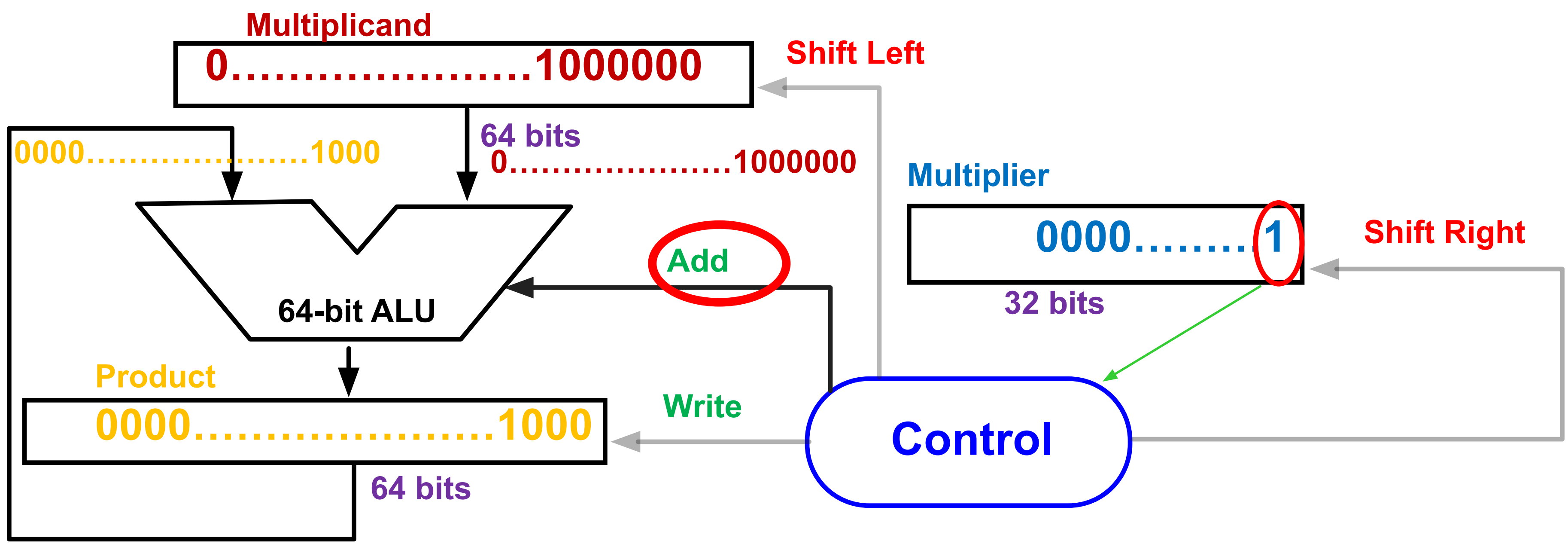
Multiplier = Datapath + Control

Unsigned shift-add multiplier (version 1)



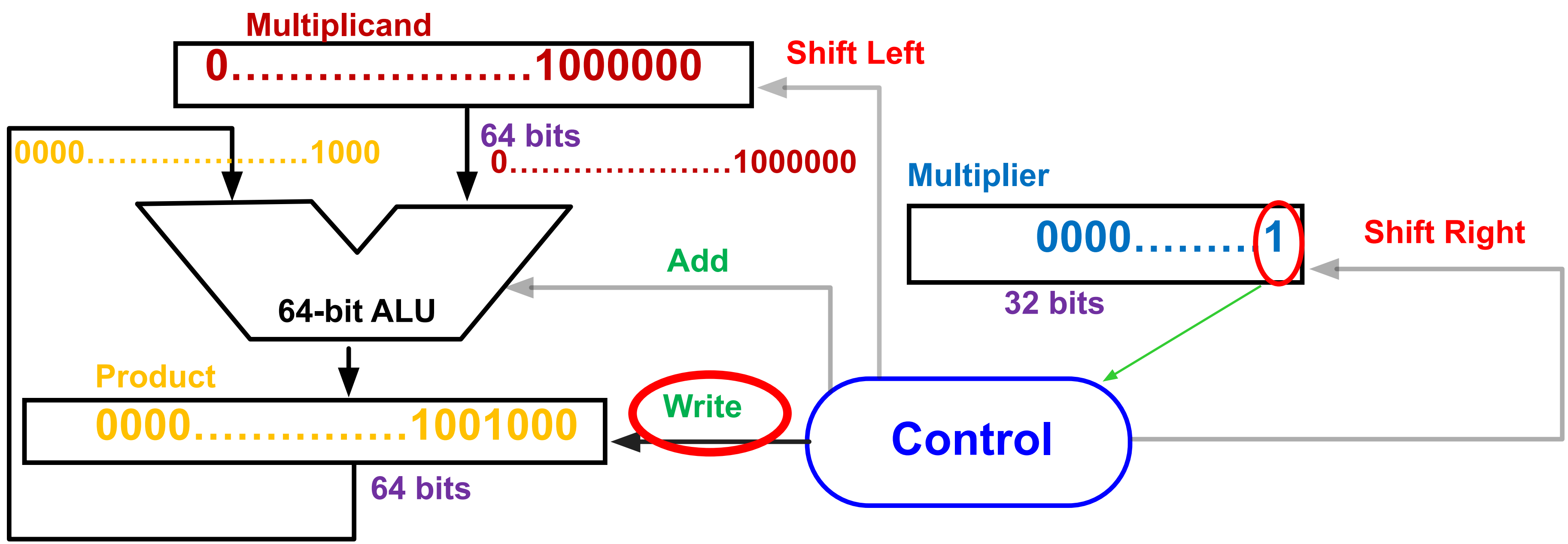
Multiplier = Datapath + Control

Unsigned shift-add multiplier (version 1)



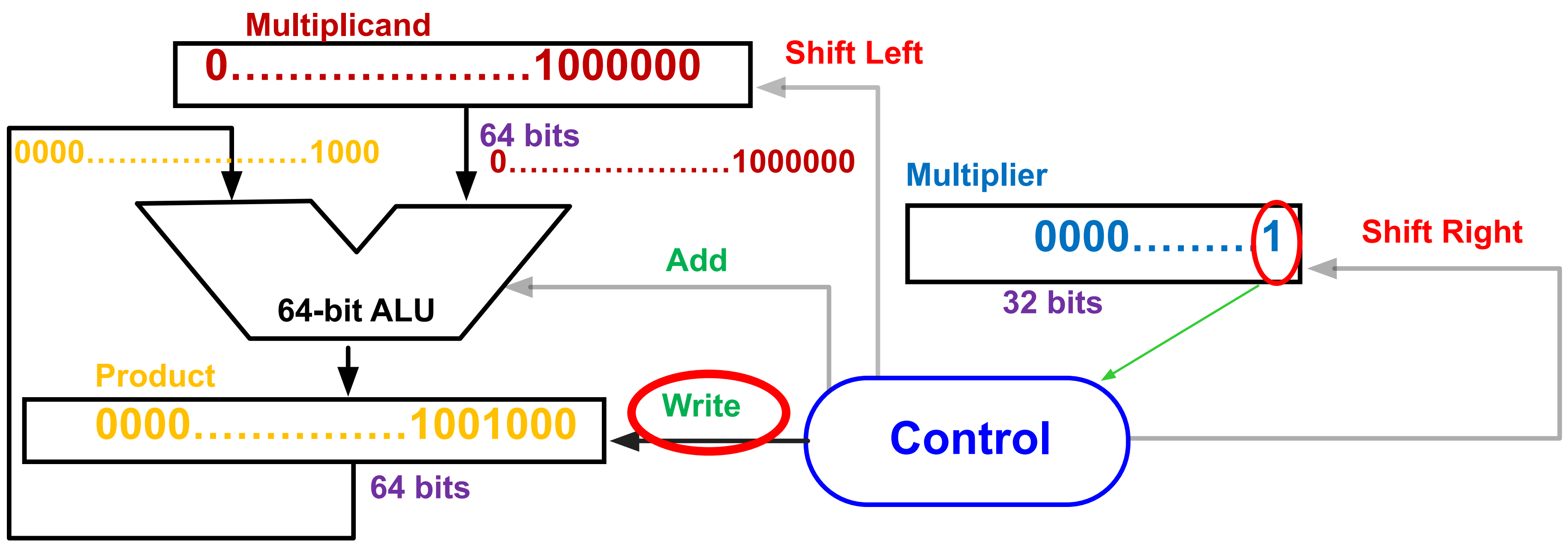
Multiplier = Datapath + Control

Unsigned shift-add multiplier (version 1)



Multiplier = Datapath + Control

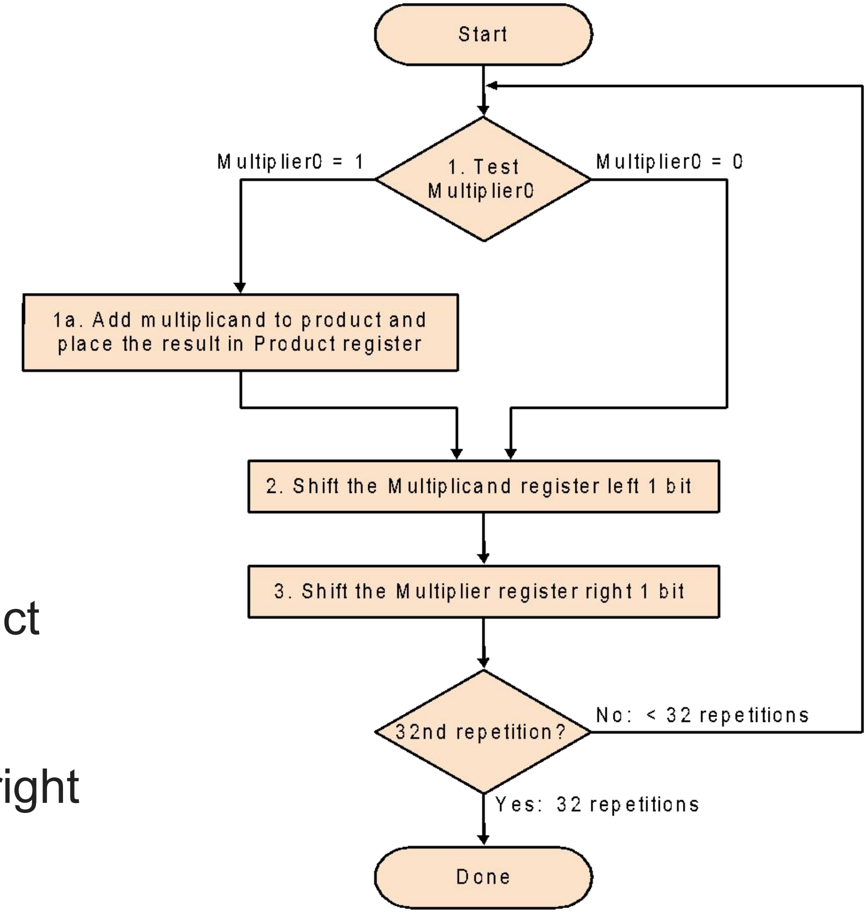
Unsigned shift-add multiplier (version 1)



Multiplier = Datapath + Control

Multiply Algorithm Version 1

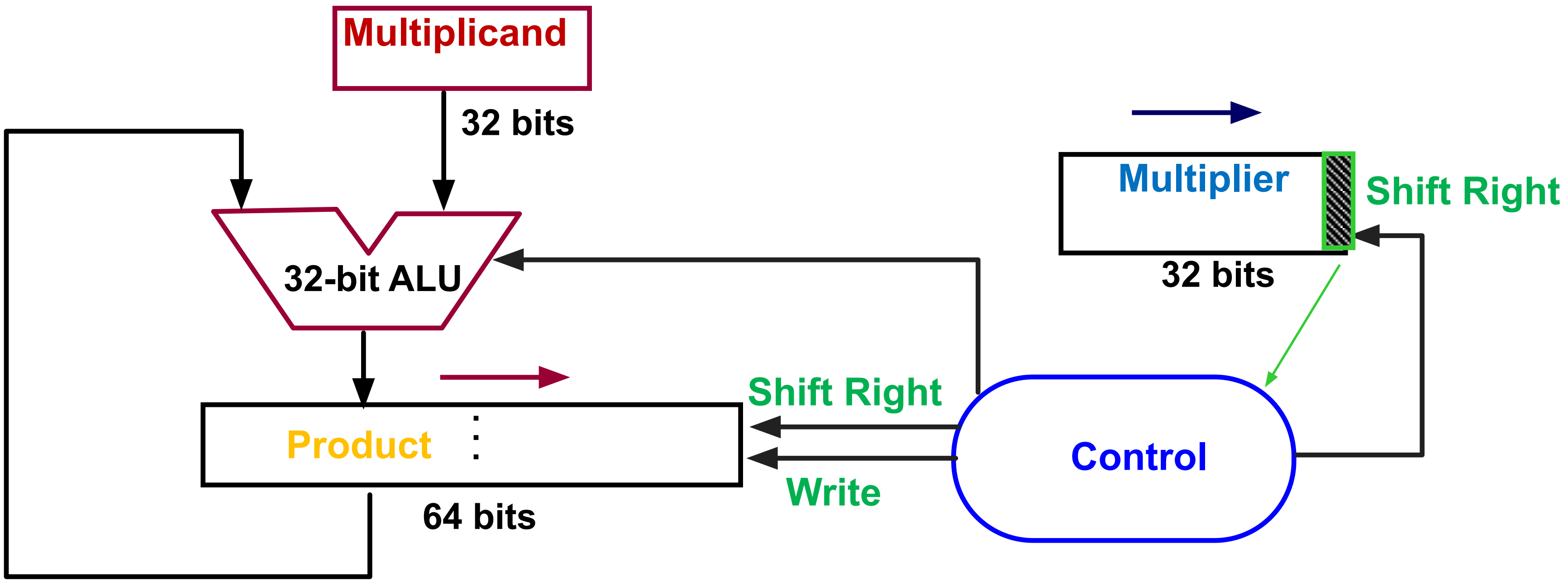
Multiplying two n-bit numbers needs a maximum of $2n^2$ addition operations mostly for adding zeros



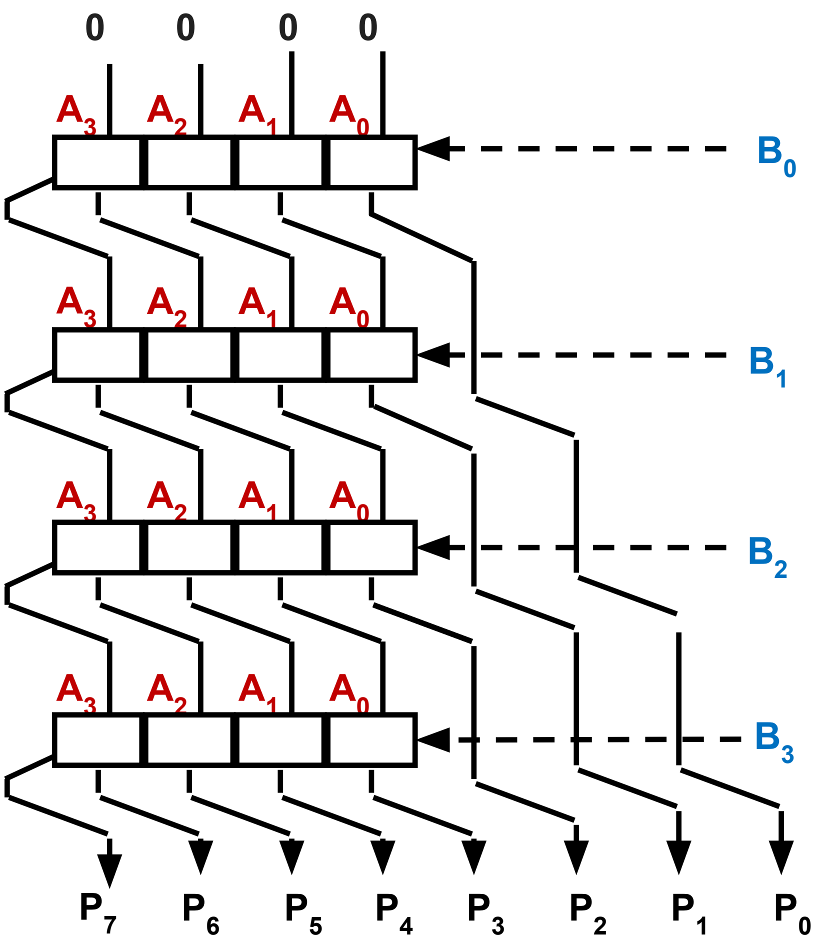
- If the least significant bit of the multiplier is
  - 1, then add the multiplicand to the product
  - 0, then go to the next bit
- Shift the multiplicand left and the multiplier right
- Repeat for 32 times

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
- This version 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



Multiply Algorithm Version 2



Multiplicand stays still
Product shifts right
Multiplier shifts right!

An Example

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

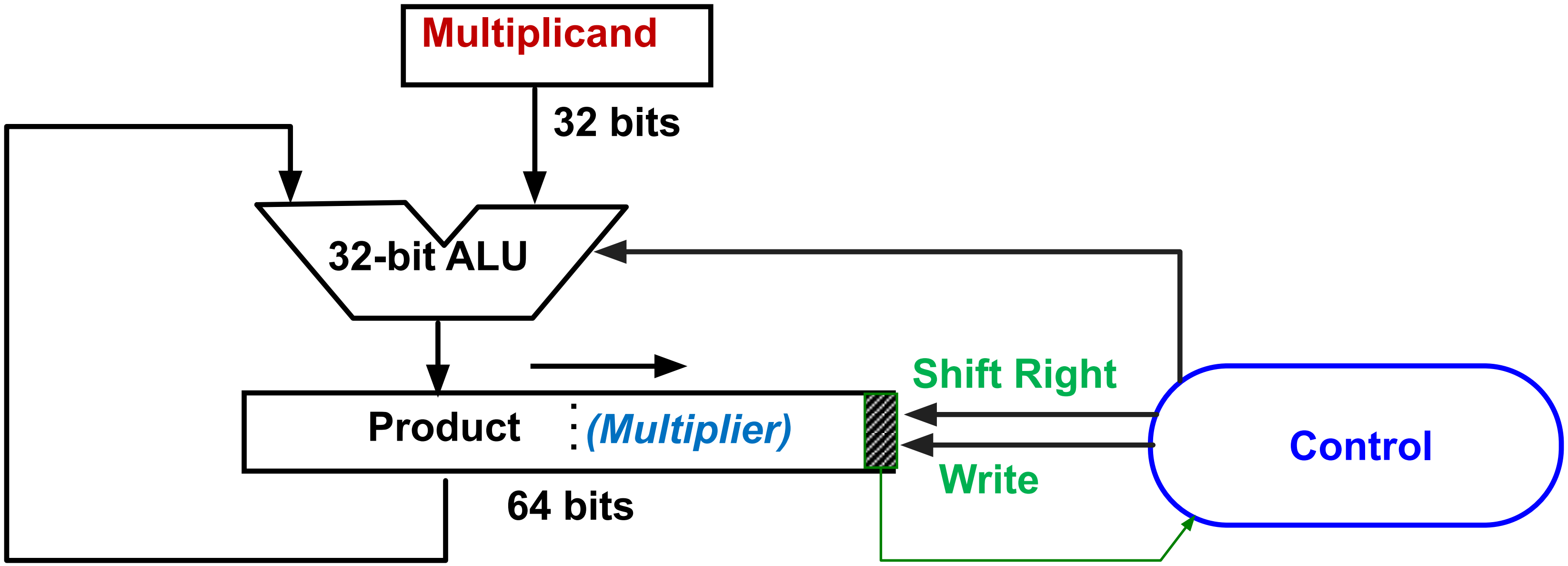
Doesn't Change

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial value	0011	0010	0000 0000
1				
2				
3				
4				

Can we do something better?

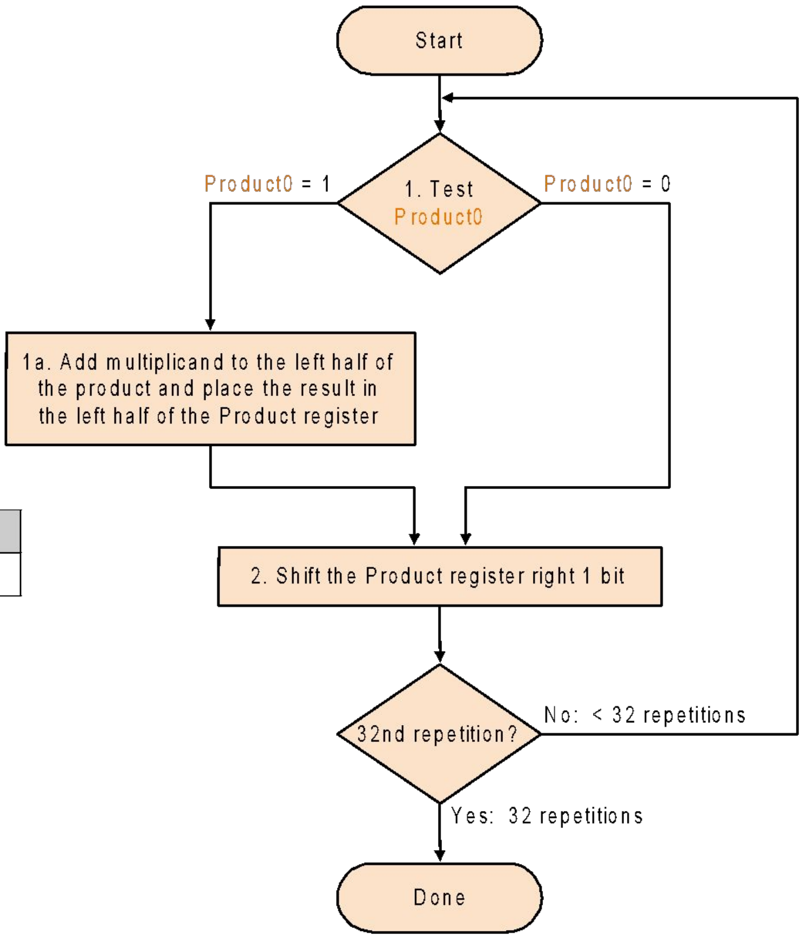
Multiply Hardware Version 3

- 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



Multiply Algorithm Version 3

2 steps per bit because
Multiplier & Product combined



The product of 2 × 3

Iteration	Step	Multiplicand	Product
0	Initial value	0010	0000 0011
1			
2			
3			
4			

MIPS registers Hi & Lo

Multiplying Signed Number

- Easiest solution is to make both positive and remember whether to complement product when done (leave out the sign bit, run for 31 steps)
- Alternative idea: 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			
2			
3			
4			

-128+(64+32+16+2) = -14

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

Solution: Booth's Algorithm

An elegant way to multiply signed numbers using same hardware as before and save cycles

The Idea Behind Booth's Algorithm

Let's say we want to compute the following multiplication

$$1101 \times 0111$$

Since there are three 1's in 0111, normally this would require
3 "8-bit numbers addition" = 24 "1-bit addition"

How about writing the same multiplication in the following format?

$$1101 \times (1000-0001)$$

There are two 1's here, this would require
1 "8-bit numbers addition" and 1 "8-bit numbers subtraction"

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

Booth's Algorithm

Let's say we want to multiply M by n = 6

$n = 6 = (0110)_2$

Product $= M * (8 – 2)$

$= (M * 8) - (M * 2)$

$= (M * 2^3) - (M * 2^1)$

= shifting M left by 3 - shifting M left by 1

How can this be implemented on ALU? Booth came up with the following algo

Scan left from the LSB, compare current "bit *i*" with the previous "bit *i*-1" to take the arithmetic action

00 ⇒ no arithmetic operation

01 ⇒ add multiplicand to left half of product

10 ⇒ subtract multiplicand from left half of product

11 ⇒ no arithmetic operation

Important: If it is the FIRST pass, use 0 as the previous LSB.

Binary Arithmetic

2^5	2^4	2^3	2^2	2^1	2^0	
0	1	1	0	0	0	$\Rightarrow$ 24 $= 2^5 - 2^3 = 32 - 8$

Starting from LSB, scan left

`0 0` $\Rightarrow$ no arithmetic operation

`0 1` $\Rightarrow$ add 2^i to the sum

`1 0` $\Rightarrow$ subtract 2^i from the sum

`1 1` $\Rightarrow$ no arithmetic operation

```
  0010             2
x 0110          x  6
```

Product = 2 * (8 – 2)

= (2 * 8) - (2 * 2)

= ($0010 * 2^3$) - ($0010 * 2^1$)

= (shift 0010 left by 3) – (shifting 0010 left by 1)

= 0001 0000 – 0000 0100

= 0001 0000 + 1111 1100 (2's complement of 4)

Booth’s Algorithm

Why should we use Booth’s Algorithm?

- Provides fast multiplication for consecutive 0's or 1's in the multiplier
- Handles signed multiplication
  - Extend the sign when shifting to preserve the sign (arithmetic right shift)

Example (unsigned numbers)

Compare the multiplication algorithm (version 3) and Booth's algorithm applied to getting the product of 2 × 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 ⇒ no operation	0000 0110	1a: 00 ⇒ no operation	0000 0110 0
0010	2: Shift right Product	0000 0011	2: Shift right Product	0000 0011 0
0010	1a: 1 ⇒ Prod = Prod + Mcand	0010 0011	1a: 10 ⇒ Prod = Prod - Mcand	1110 0011 0
0010	2: Shift right Product	0001 0001	2: Shift right Product	1111 0001 1
0010	1a: 1 ⇒ Prod = Prod + Mcand	0011 0001	1a: 11 ⇒ no operation	1111 0001 1
0010	2: Shift right Product	0001 1000	2: Shift right Product	1111 1000 1
0010	1a: 0 ⇒ no operation	0001 1000	1a: 01 ⇒ 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 × -3 using only 4-bit binary representation

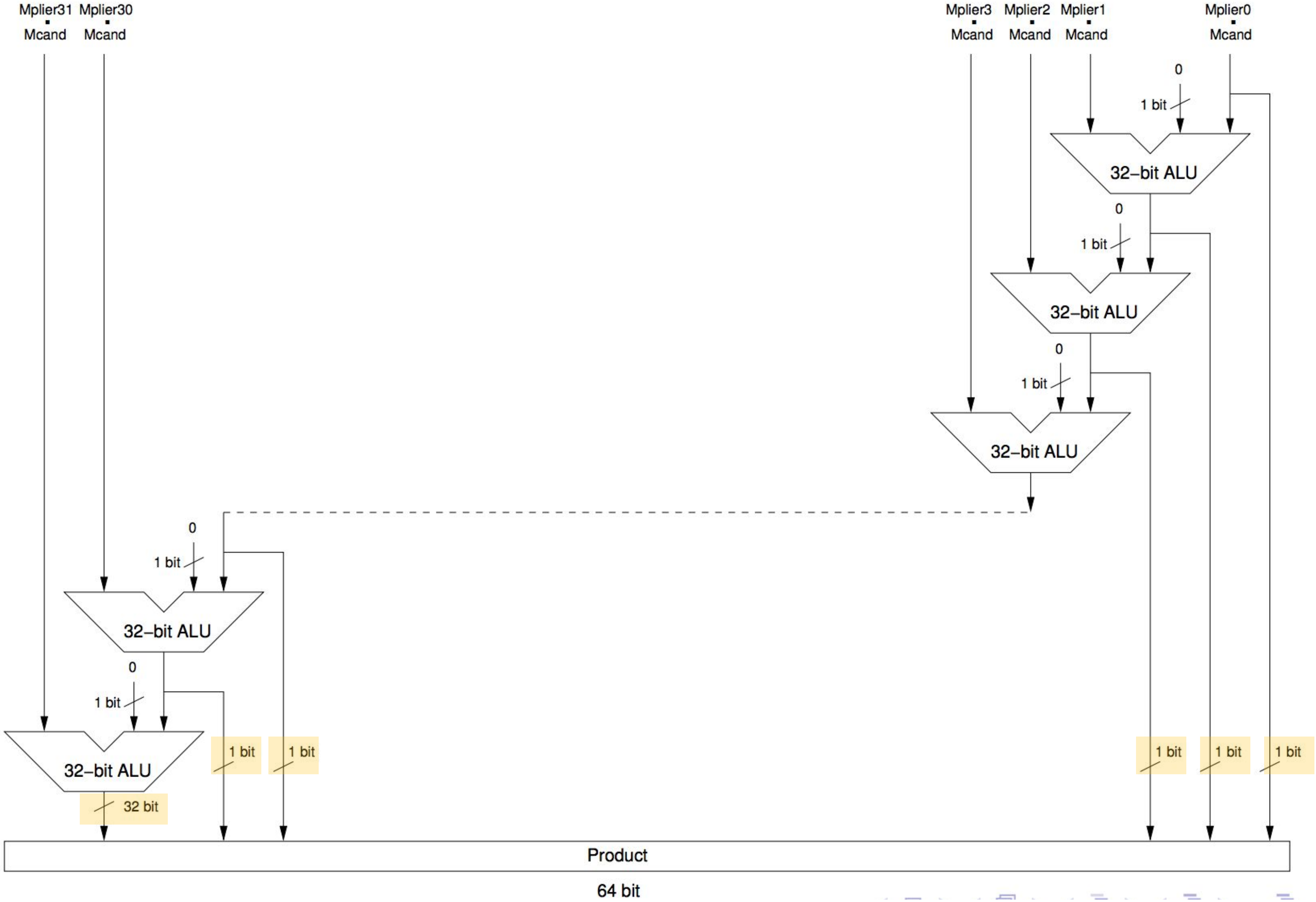
Iteration	Step	Multiplicand	Product
0	Initial value	0010	0000 1101 0
1	1a: 10 ⇒ Prod = Prod - Mcand	0010	1110 1101 0
	2: Shift right Product	0010	1111 0110 1
2	1a: 01 ⇒ Prod = Prod + Mcand	0010	0001 0110 1
	2: Shift right Product	0010	0000 1011 0
3	1a: 10 ⇒ Prod = Prod - Mcand	0010	1110 1011 0
	2: Shift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

Faster Multiplication-1

- We know whether the multiplicand is to be added or not at the beginning of the multiplication by looking at each of the 32 multiplier bits
- Faster multiplications are possible by essentially providing **one 32-bit adder for each bit of the multiplier**
  - Input-1: the multiplicand ANDed with a multiplier bit
  - Input-2: the output of a prior adder

Faster Multiplication-1

Since we deal with bit-wise ANDs, no data storage is required



Faster Multiplication-2