Lecture 7
Multiplication

Byung-gi Kim

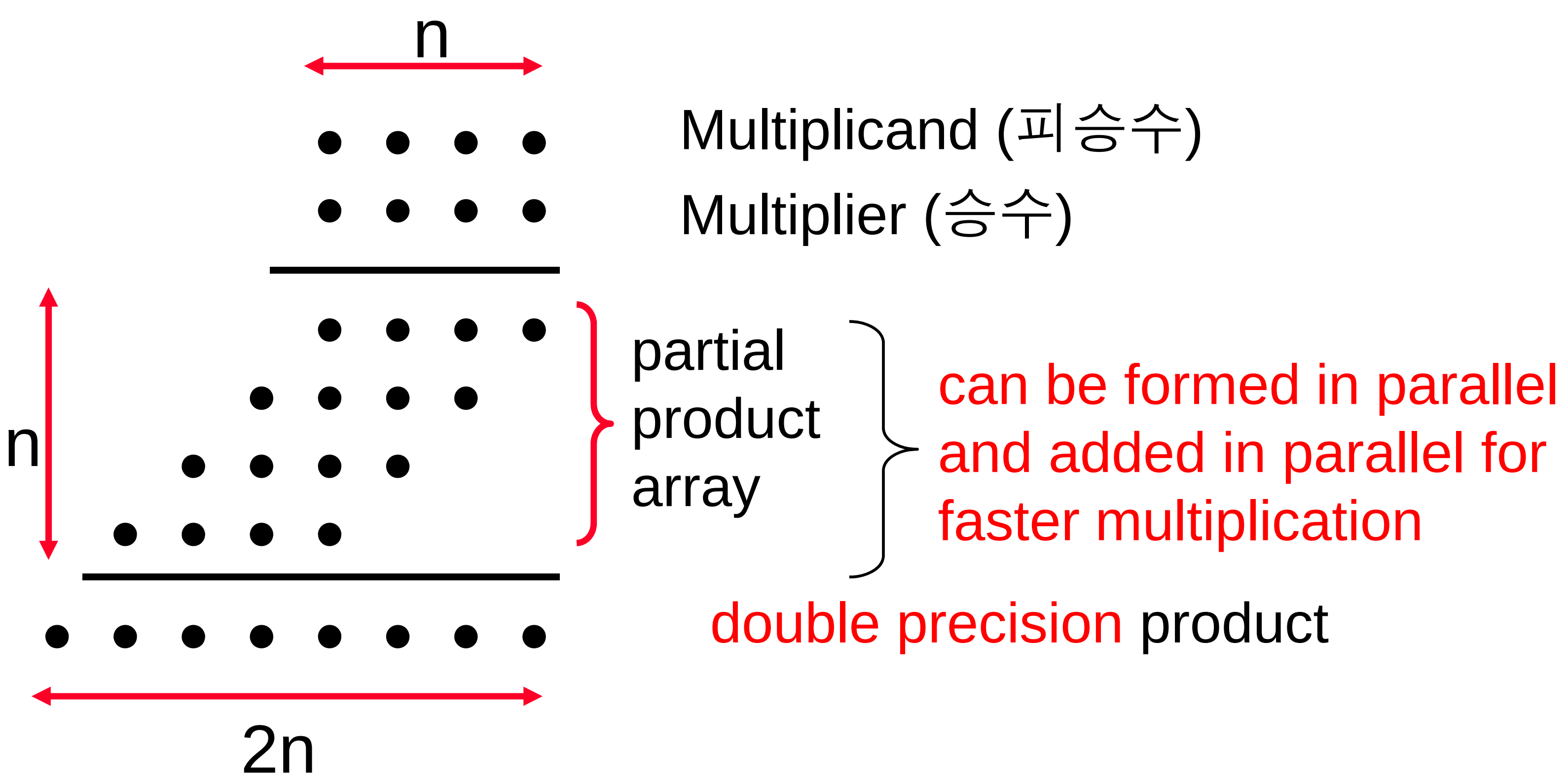
School of Computer Science and Engineering

Soongsil University

3. Arithmetic for Computers

3.3 Multiplication

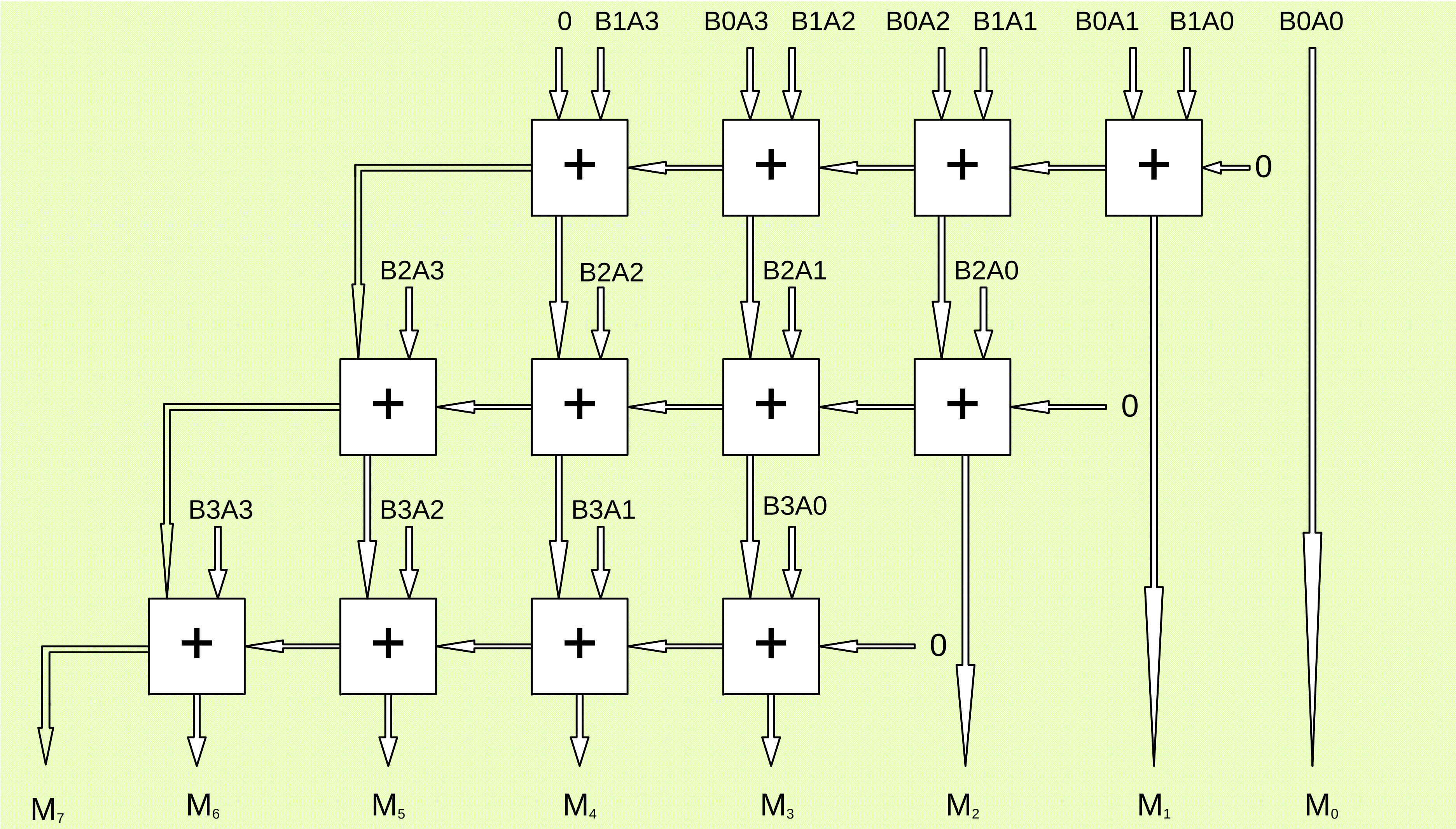
- Binary multiplication is just a bunch of left shifts and adds



Pencil and Paper Algorithm

```
                                          A3      A2      A1      A0
                                       *  B3      B2      B1      B0
                                         --------------------------------
                                         B0•A3   B0•A2   B0•A1   B0•A0
                                 B1•A3   B1•A2   B1•A1   B1•A0
                         B2•A3   B2•A2   B2•A1   B2•A0
                 B3•A3   B3•A2   B3•A1   B3•A0
         ----------------------------------------------------------------
         M7      M6      M5      M4      M3      M2      M1      M0
```

Array Multiplier



Binary Multiplication

- **An example: 8 x 9 = 72**

```
Multiplicand:                   1  0  0  0
Multiplier:                *    1  0  0  1
                              -------------
                                1  0  0  0
                             0  0  0  0
                          0  0  0  0
                       1  0  0  0
                      ---------------------
Product:               1  0  0  1  0  0  0
```

- **Ignoring the sign bits,**
  - ***n***-bit multiplicand and ***m***-bit multiplier
  - Then **(*n*+*m*)**-bit product
- **Each step of the multiplication**
  - If multiplier bit = 1, copy the multiplicand.
  - If multiplier bit = 0, place all 0's.

Sequential Version of the Multiplication Algorithm and Hardware

- **First version of the multiplication hardware**

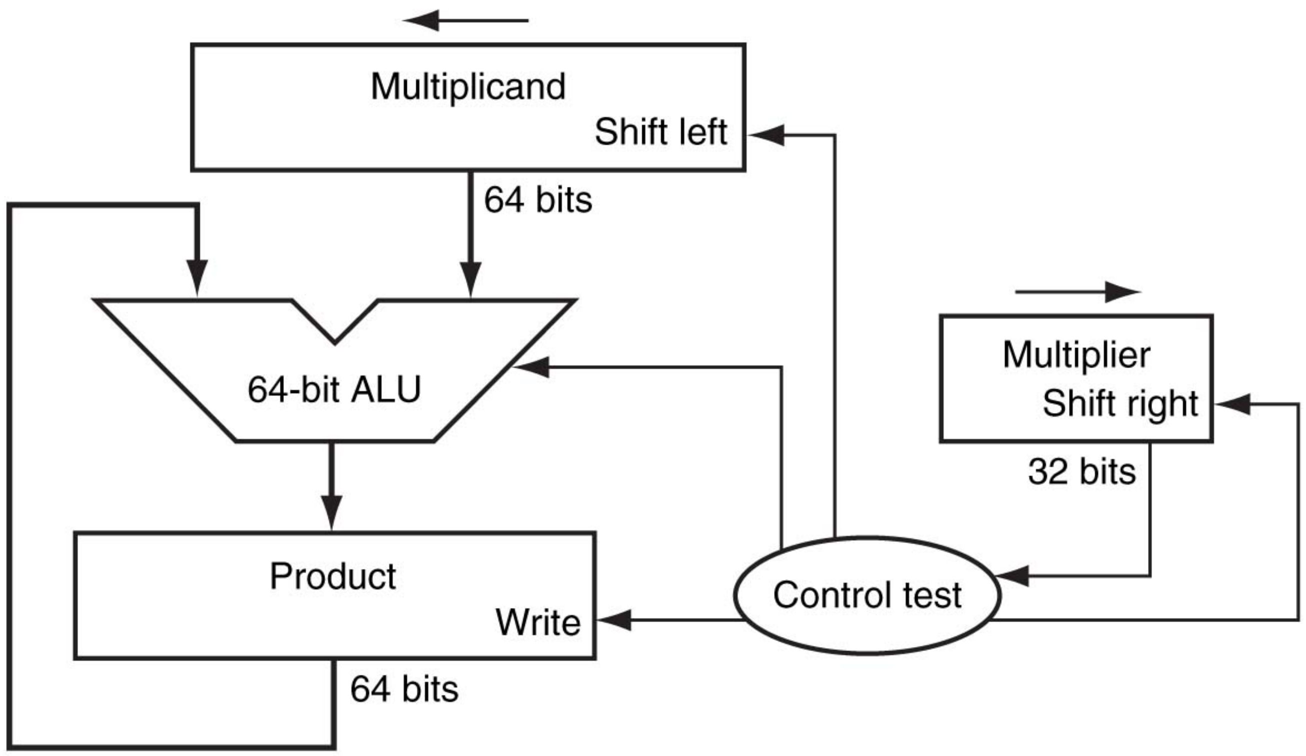


Figure 3.4

First Version - Algorithm

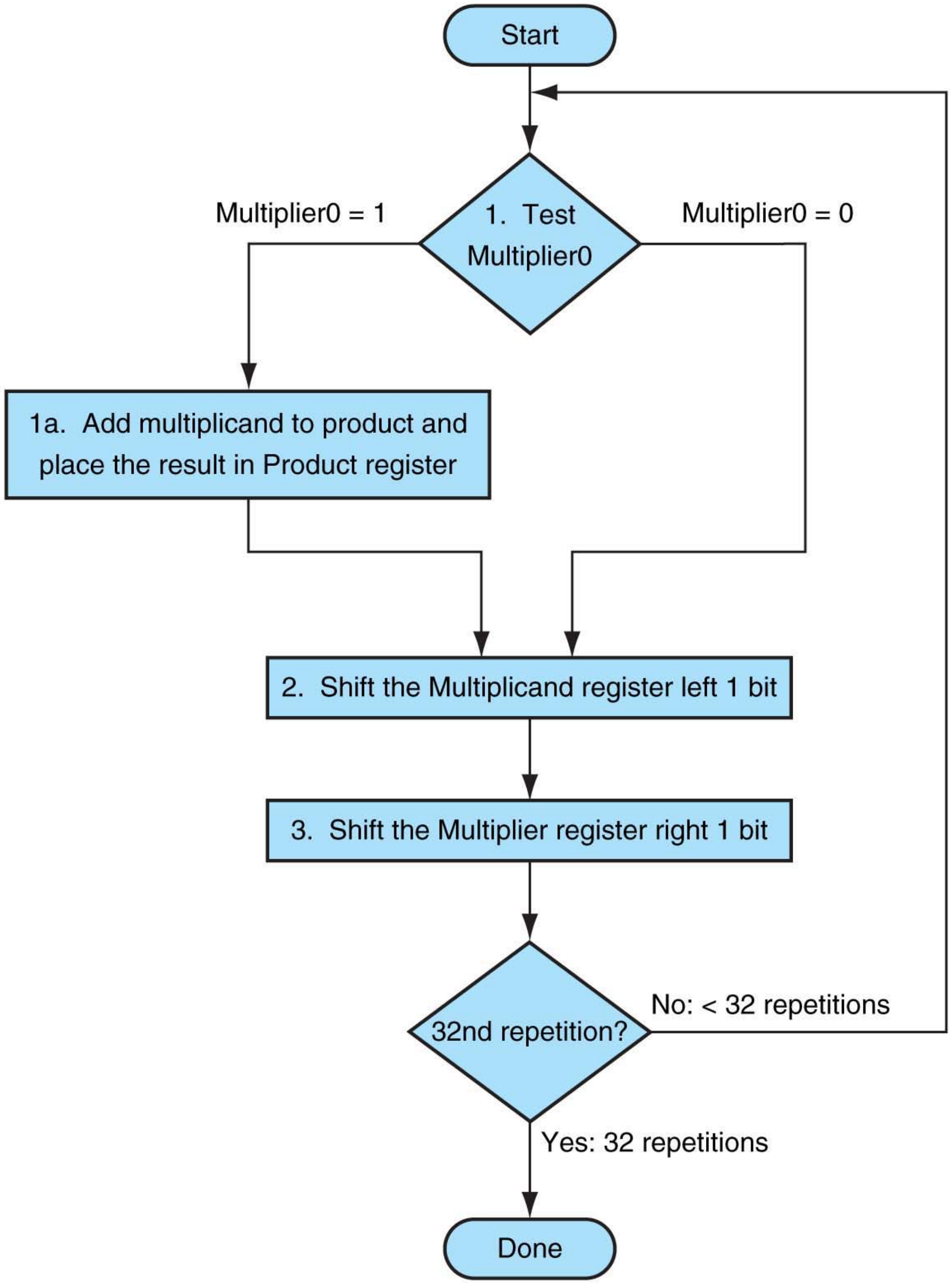


Figure 3.5

Example: First Version

- **Multiply 0010$_{two}$ X 0011$_{two}$.**

  [Answer] Multiplication time … almost 100 clock cycles

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Product += Multiplicand	0011		0000 0010
	2: Shift Multiplicand left		0000 0100	
	3: Shift Multiplier right	0001		
2	1a: 1 ⇒ Product += Multiplicand	0001		0000 0110
	2: Shift Multiplicand left		0000 1000	
	3: Shift Multiplier right	0000		
3	1: 0 ⇒ no operation	0000		0000 0110
	2: Shift Multiplicand left		0001 0000	
	3: Shift Multiplier right	0000		
4	1: 0 ⇒ no operation	0000		0000 0110
	2: Shift Multiplicand left		0010 0000	
	3: Shift Multiplier right	0000		0000 0110

Second Version - Hardware

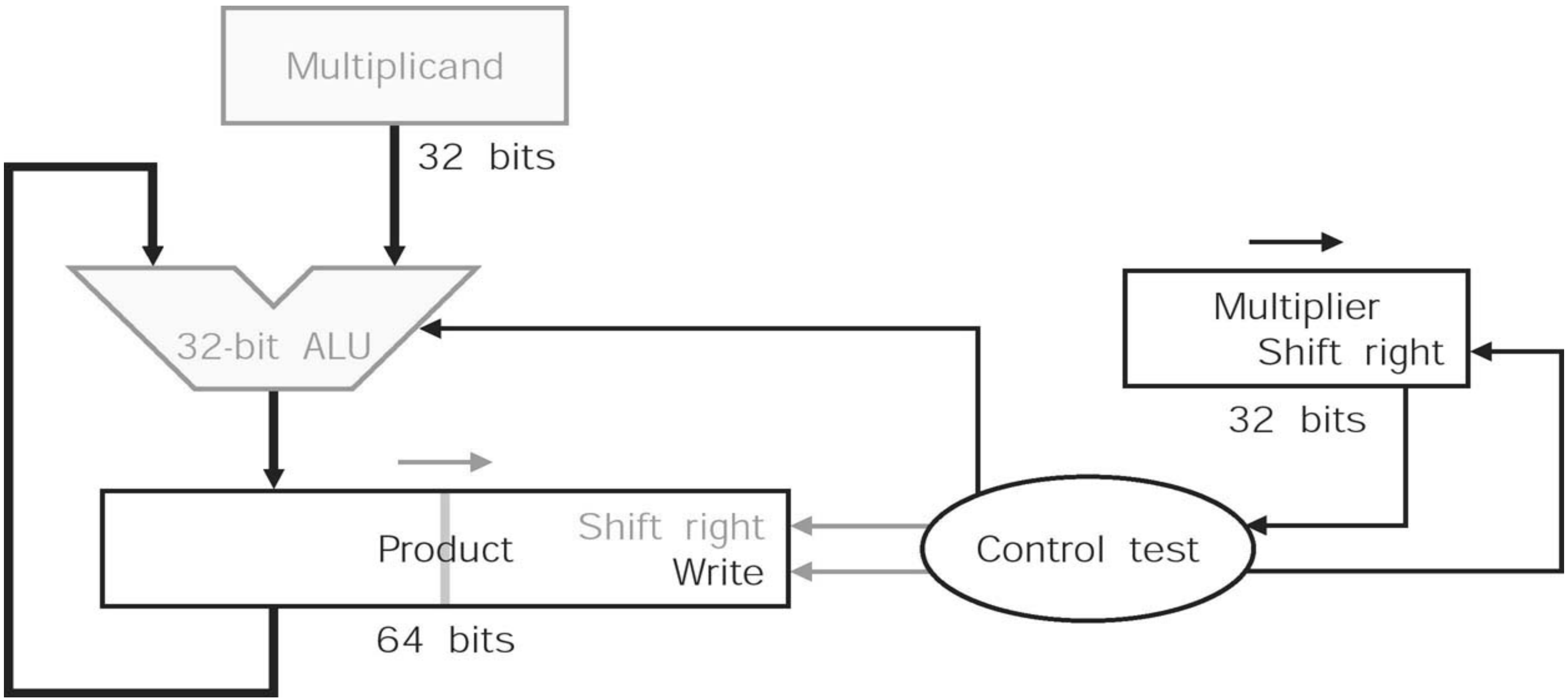


Figure 4.28 of 2ed.

Example: Second Version

- **Multiply 0010_{two} X 0011_{two}.**
  [Answer]

Figure 4.30 of 2ed.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0010	0000 0000
1	1a: 1 ⇒ Product += Multiplicand	0011		0010 0000
	2: Shift Product right			0001 0000
	3: Shift Multiplier right	0001		
2	1a: 1 ⇒ Product += Multiplicand	0001		0011 0000
	2: Shift Product right			0001 1000
	3: Shift Multiplier right	0000		
3	1: 0 ⇒ no operation	0000		0001 1000
	2: Shift Product right			0000 1100
	3: Shift Multiplier right	0000		
4	1: 0 ⇒ no operation	0000		0000 1100
	2: Shift Product right			0000 0110
	3: Shift Multiplier right	0000		0000 0110

Refined Version of the Multiplication Hardware

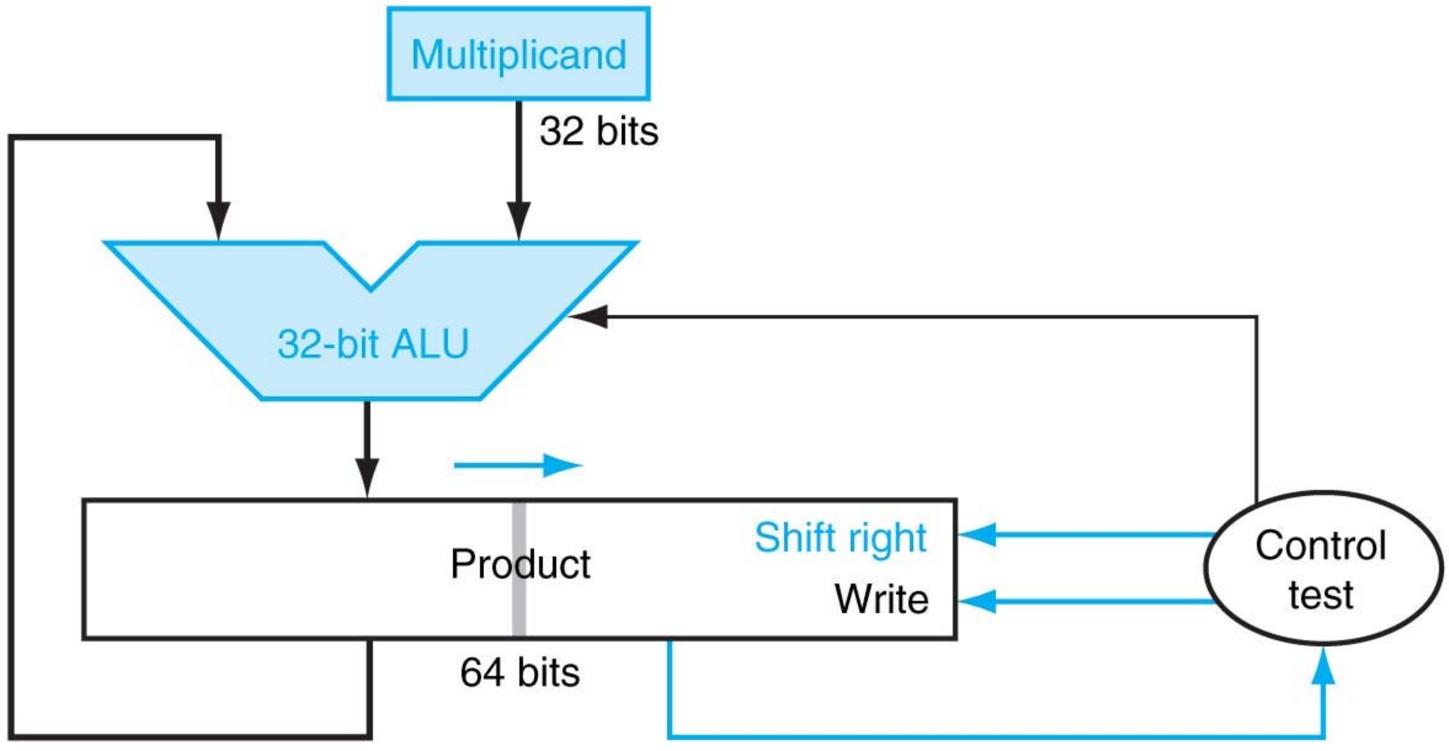


Figure 3.6

Refined Version - Algorithm

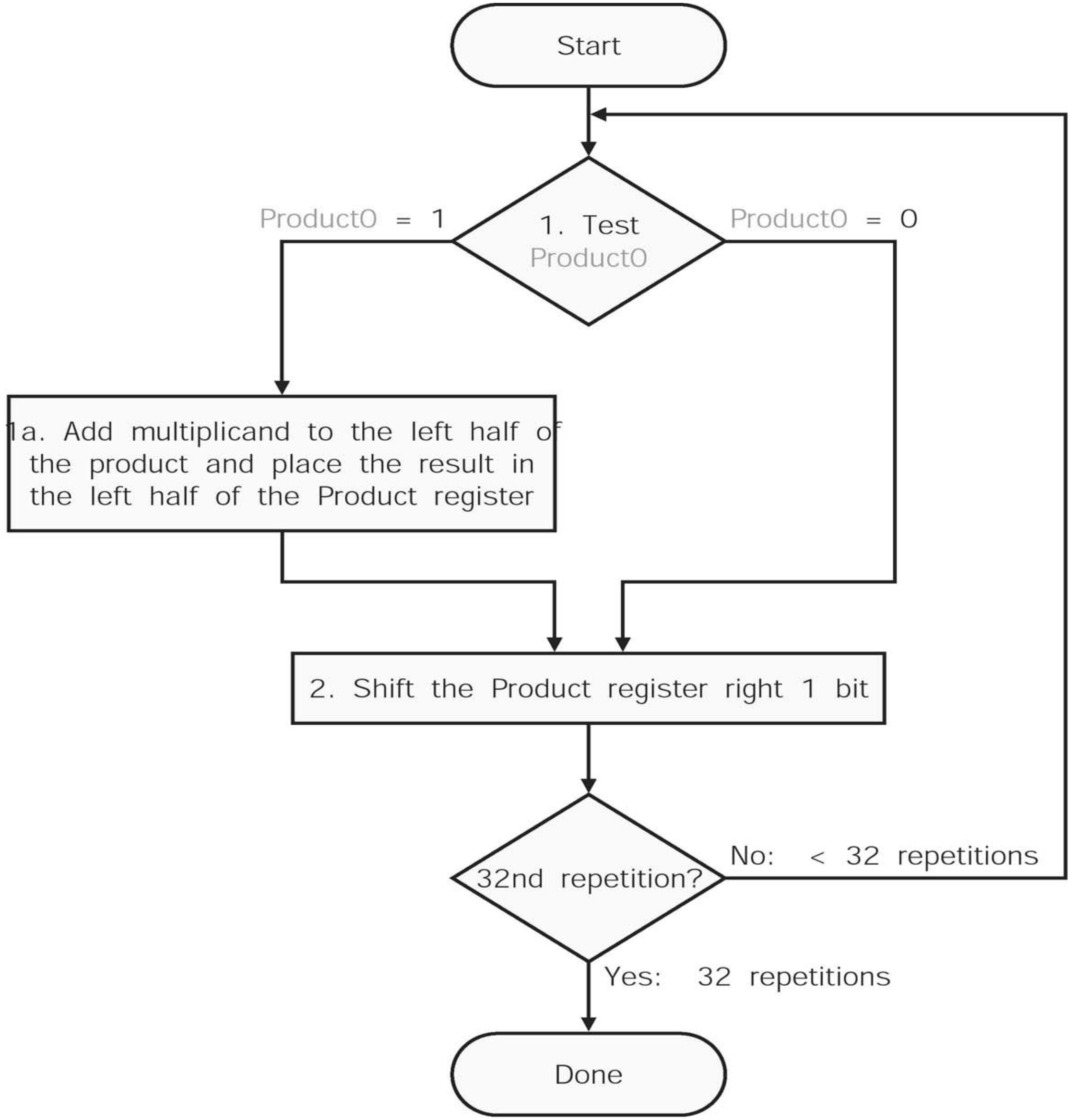


Figure 4.32 in 2ed.

Example 1: Refined Version

- **Multiply 0010$_{two}$ X 0011$_{two}$.**

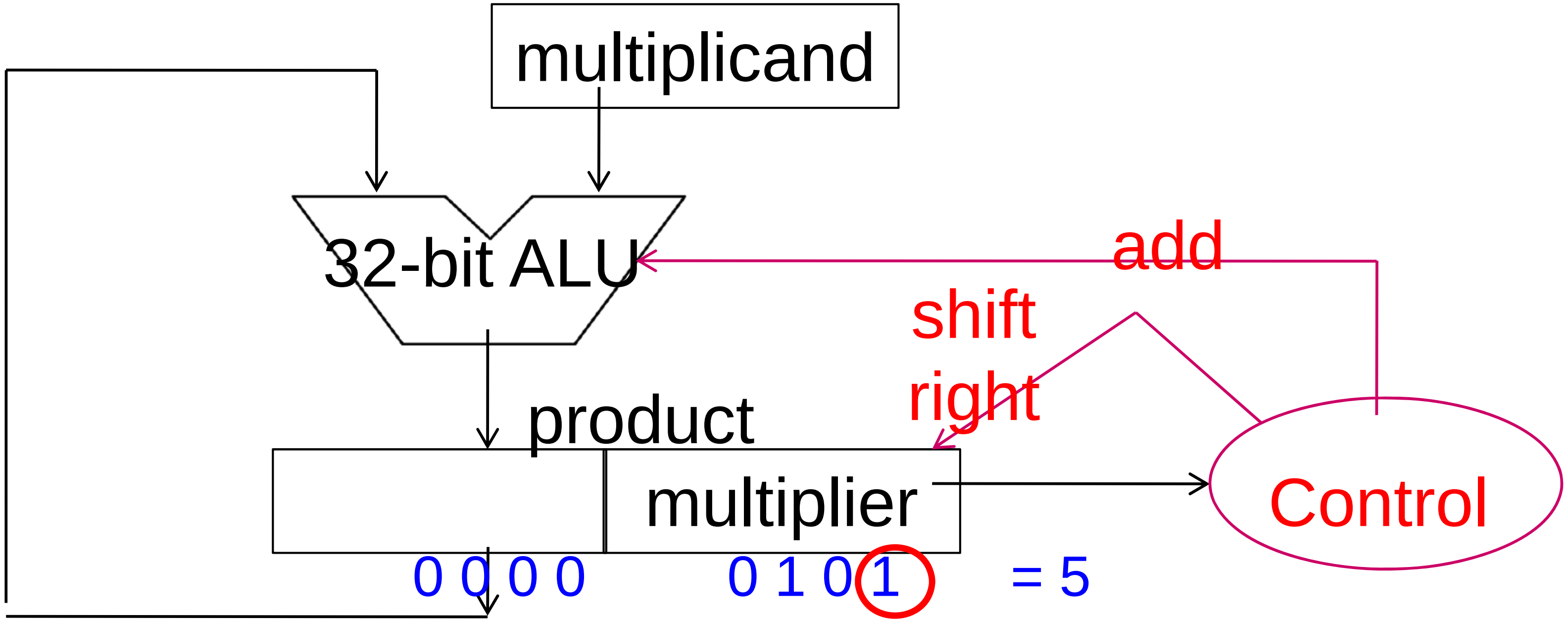
[Answer]

Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 0011
1	1a: 1 ⇒ Product += Multiplicand		0010 0011
	2: Shift Product right		0001 0001
2	1a: 1 ⇒ Product += Multiplicand		0011 0001
	2: Shift Product right		0001 1000
3	1: 0 ⇒ no operation		0001 1000
	2: Shift Product right		0000 1100
4	1: 0 ⇒ no operation		0000 1100
	2: Shift Product right		0000 0110

Figure 4.33 of 2ed.

Example 2: Refined Version

0 1 1 0 = 6



	0 0 0 0	0 1 0 1	= 5
add	0 1 1 0	0 1 0 1	
	0 0 1 1 →	0 0 1 0	
no add	0 0 1 1	0 0 1 0	
	0 0 0 1 →	1 0 0 1	
add	0 1 1 1	1 0 0 1	
	0 0 1 1 →	1 1 0 0	
no add	0 0 1 1	1 1 0 0	
	0 0 0 1 →	1 1 1 0	= 30

Signed Multiplication

- **Booth's Algorithm**
  - Multiplication algorithm for signed 2's complement numbers
- **Key idea**
  - $30_{ten} = 16_{ten} + 8_{ten} + 4_{ten} + 2_{ten}$
    $= 32_{ten} - 2_{ten}$
  - 011110_{two}
    $= 010000_{two} + 001000_{two} + 000100_{two} + 000010_{two}$
    $= 100000_{two} - 000010_{two}$

Multiplication by Booth's Algorithm

1. Depending on the current and previous bits

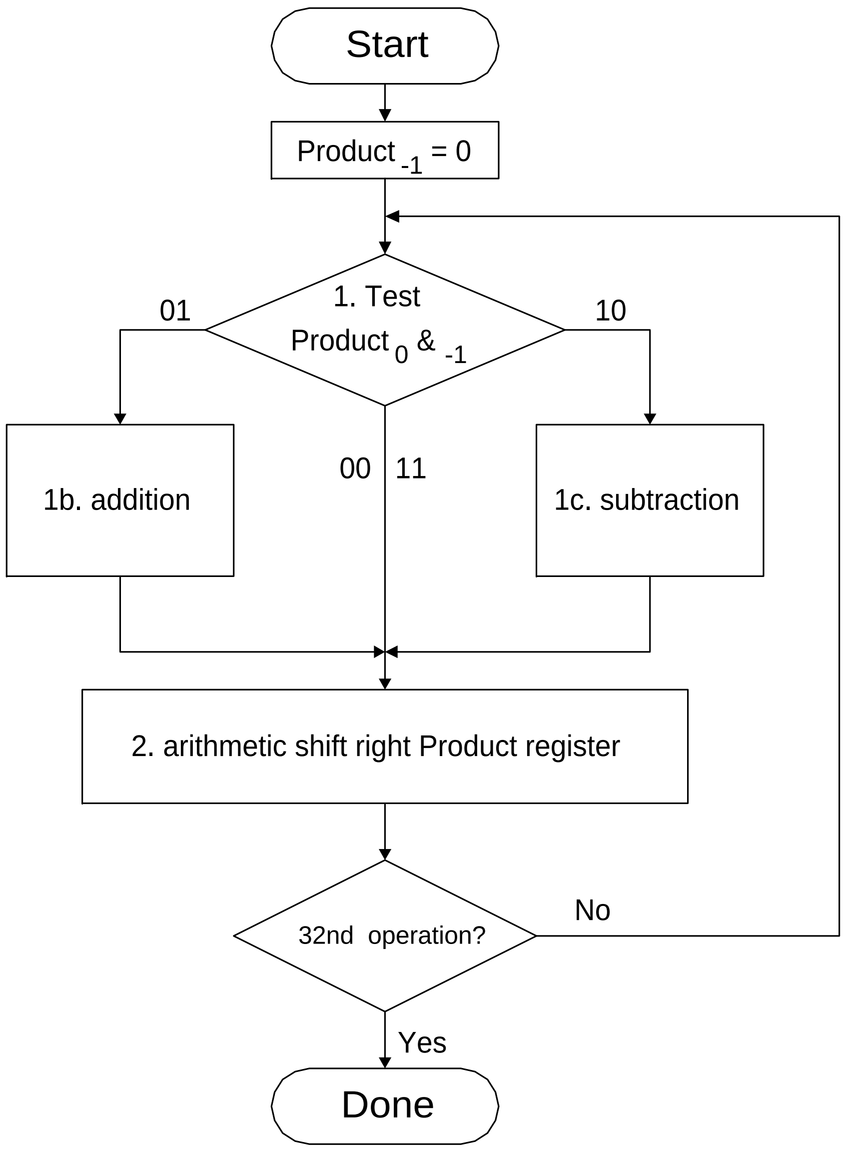
00, 11: No arithmetic

01: Addition (end of a string of 1s)

10: Subtraction (beginning of a string of 1s)

2. Arithmetic shift right the product register.

Booth's Algorithm - Flowchart



Example: Booth's Algorithm

- **Multiply 0010_{two} X 1101_{two}**
  [Answer]

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

Comparison

Iteration	Multiplicand	Original algorithm		Booth's algorithm	
		Step	Product	Step	Product
0	0010	Initial values	0000 0110	Initial values	0000 0110 0
1	0010	1: 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
2	0010	1a: 1 => Prod=Prod+Mcand	0010 0011	1c: 10 => Prod=Prod-Mcand	1110 0011 0
	0010	2: Shift right Product	0001 0001	2: Shift right Product	1111 0001 1
3	0010	1a: 1 => Prod=Prod+Mcand	0011 0001	1d: 11 => no operation	1111 0001 1
	0010	2: Shift right Product	0001 1000	2: Shift right Product	1111 1000 1
4	0010	1: 0 => no operation	0001 1000	1b: 01 => Prod=Prod+Mcand	0001 1000 1
	0010	2: Shift right Product	0000 1100	2: Shift right Product	0000 1100 0

Figure 4.34 in 2ed.

Booth's Algorithm – Theoretical Background (1/2)

- **Booth's algorithm with signed 2's complement numbers**
  - If $((a_{i-1} - a_i) == 0)$ do nothing;
    else if $((a_{i-1} - a_i) == (+1))$ add B;
    else if $((a_{i-1} - a_i) == (-1))$ subtract B;
  - Product $= (a_{-1} - a_0) \times B$

$$
\begin{aligned}
&+ (a_0 - a_1) \times B \times 2^1 \\
&+ (a_1 - a_2) \times B \times 2^2 \\
&\quad \cdots \\
&+ (a_{29} - a_{30}) \times B \times 2^{30} \\
&+ (a_{30} - a_{31}) \times B \times 2^{31}
\end{aligned}
$$

Booth's Algorithm – Theoretical Background (2/2)

- Product = $B\times((a_{-1}+(a_0\times 2^0)+(a_1\times 2^1)+\ldots$
  $+(a_{30}\times 2^{30})+(a_{31}\times -2^{31}))$
  $=B\times((a_0\times 2^0)+(a_1\times 2^1)+\ldots$
  $+(a_{30}\times 2^{30})+(a_{31}\times -2^{31}))$
  $=B\times(-a_{31}\times 2^{31}+\sum a_i\times 2^i)$
  $\Rightarrow B\times(A$ in two's complement representation$)$

(cf) Value of $X=x_{n-1}x_{n-2}\cdots x_1x_0$ in signed 2'complement

$$V(X)=-x_{n-1}\cdot 2^{n-1}+\sum_{k=0}^{n-2}x_k\cdot 2^k$$

Supplement

Hardware/Software Interface

- **Replacing arithmetic by shifts**
- **Multiplies by short constants in some compilers**
  - Replacing them with a series of shifts and adds
- **Shift n-bit left = multiply by 2^n**
- **Strength reduction optimization**
  - Substituting a left shift for a multiply by a power of 2
  - Done by almost every compiler

MIPS Multiply Instruction

- Multiply (**`mult`** and **`multu`**) produces a double precision product

```
mult   $s0, $s1  # hi||lo = $s0 * $s1
```

0	16	17	0	0	0x18

  - Low-order word of the product is left in processor register **`lo`** and the high-order word is left in register **`hi`**
  - Instructions **`mfhi rd`** and **`mflo rd`** are provided to move the product to (user accessible) registers in the register file

- Multiplies are usually done by fast, dedicated hardware and are much more complex (and slower) than adders

Multiply in MIPS

- **A separate pair of 32-bit registers**
  - *Hi* and *Lo* registers
  - 64-bit product store
- **Instructions**
  - `mult  $s2,$s3 # Hi,Lo = $s2 x $s3 (signed)`
  - `multu $s2,$s3 # Hi,Lo = $s2 x $s3 (unsigned)`
  - `mul $t0,$s2,$s3 # $t0 = lower 32 bits of $s2 x $s3`
  - `mflo  $s1      # $s1 = Lo (move from lo)`
  - `mfhi  $s1      # $s1 = Hi (move from hi)`
- **Pseudoinstructions**
  - (`mul $t0,$s2,$s3` &) `mulu $t0,$s2,$s3` (no overflow check)
  - `mulo $t0,$s2,$s3` & `mulou $t0,$s2,$s3` (overflow check)

Overflow Check

- **Example C program**

```
integer x, y, z;
x = y * z;
```

  - `Y * z` in `Hi-Lo` register
- **For signed multiply**
  - When (Lo_{31} == 0), if (Hi ≠ 0000…0) then overflow
  - When (Lo_{31} == 1), if (Hi ≠ 1111…1) then overflow
- **For unsigned multiply**
  - If (Hi ≠ 0000…0) then overflow