CSC 3210 Computer organization and programming

Chapter 3

Chunlan Gao

Review

- The **Basics** of Logic Design
- Truth table
- Logic Equations
- Gates
- Multiplexors
- ROMs
- ALU
- Finite-State Machines

Finite-State Machines

Sequential systems contain state stored in memory elements internal to the system. Their behavior depends both on the set of inputs supplied and on the contents of the internal memory, or state of the system.

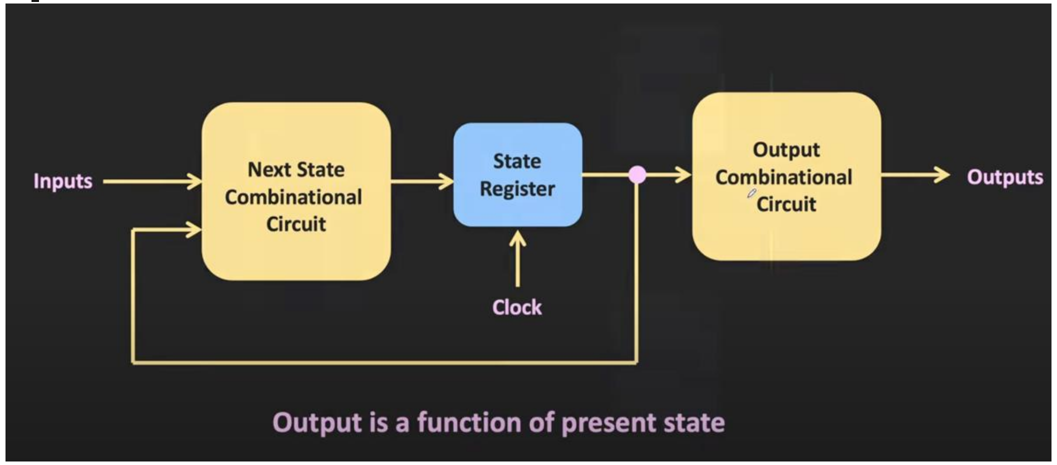
Thus, a sequential system cannot be described with a truth table.

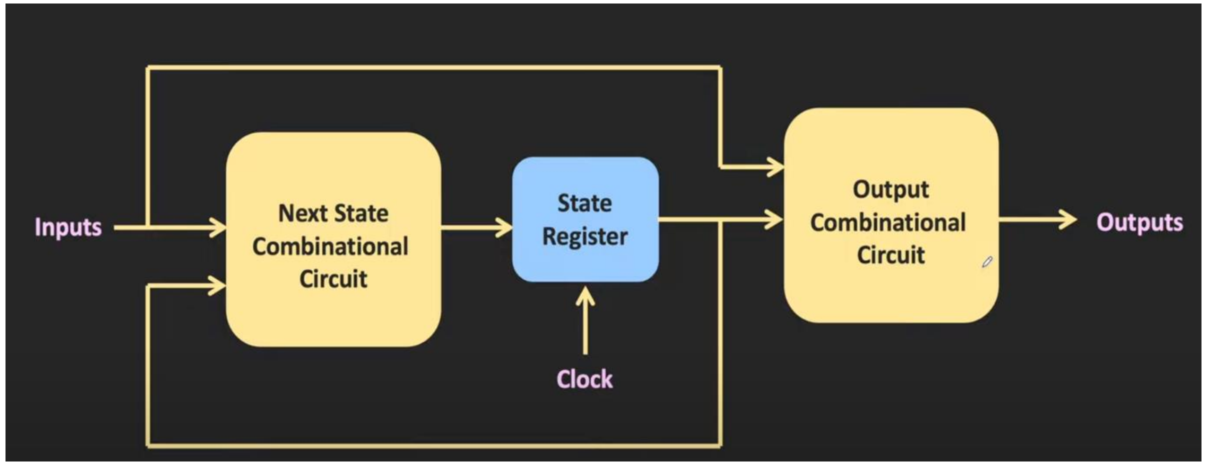
Instead, a sequential system is described as a finite-state machine (or often just state machine).

- A finite-state machine has a set of states and **two functions**, called the next-state function and the output function.

Type of Finite State Machine

.


Inputs
Next State Combinational Circuit
State Register
Clock
Output Combinational Circuit
Outputs
Output is a function of present state


Inputs
Next State Combinational Circuit
State Register
Clock
Output Combinational Circuit
Outputs

When a finite-state machine is used as a controller, the output function is often restricted to depend on just the current state. Such a finite-state machine is called **a Moore machine**.

If the output function can depend on both the current state and the current input, the machine is called a Mealy machine.

Finite-State Machine

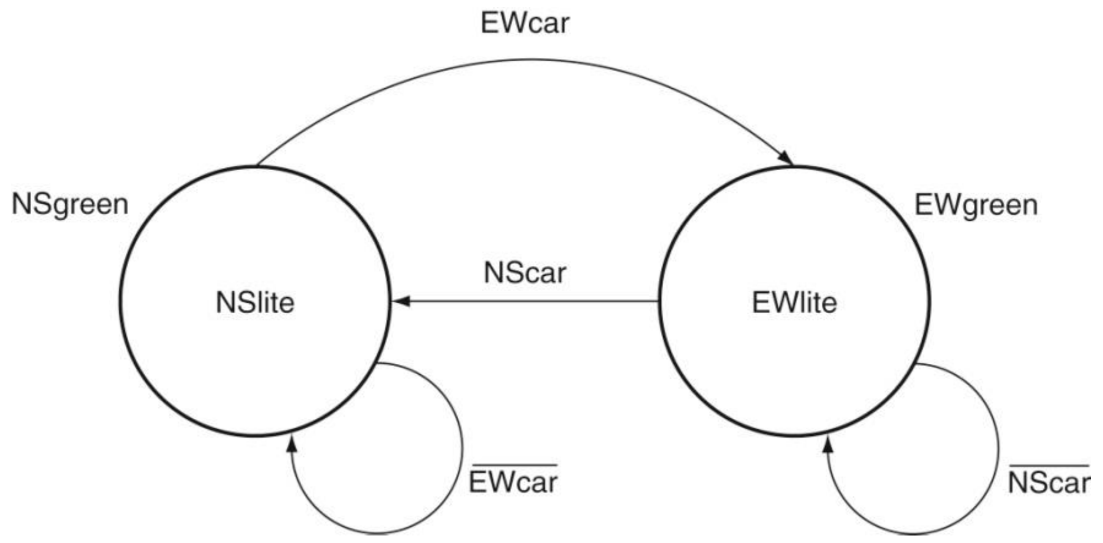


Figure A.10.2 The graphical representation of the two-state traffic light controller. We simplified the logic functions on the state transitions. For example, the transition from NSgreen to Ewgreen in the next-state table is (NScar EWcar) (NScar EWcar), which is equivalent to EWcar.

Chapter 3

Arithmetic for Computers

Arithmetic for Computers

Chapter 2 shows that integers addition and subtraction.

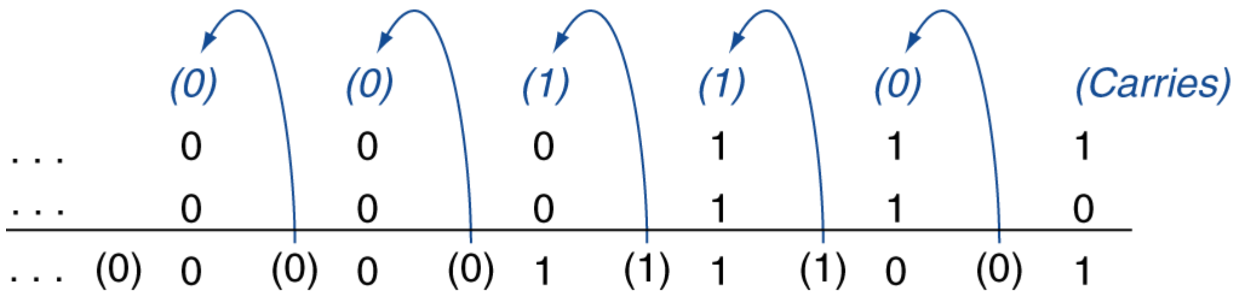
- What about multiplication and division ?
- What about fractions and other real numbers?
- What happens if an operation creates a number bigger than can be represented?
- And underlying these questions is a mystery: How does hardware really multiply or divide numbers?

Operation on integers

- Addition and subtraction
- Multiplication and division
- Dealing with overflow

Integer Addition

Example 7+6=13



Overflow if result out of range(sign number changed) :

1100	1111	1101	0001	1000	1010	0101	0010
1010	0000	1000	1000	1000	1010	0101	1111

0100	1111	1101	0001	1000	1010	0101	0010
0111	0000	1000	1000	1000	1010	0101	1111

For addition, the two operands have same sign bit have the change to overflow

Integer Subtraction

- Add negation of second operand

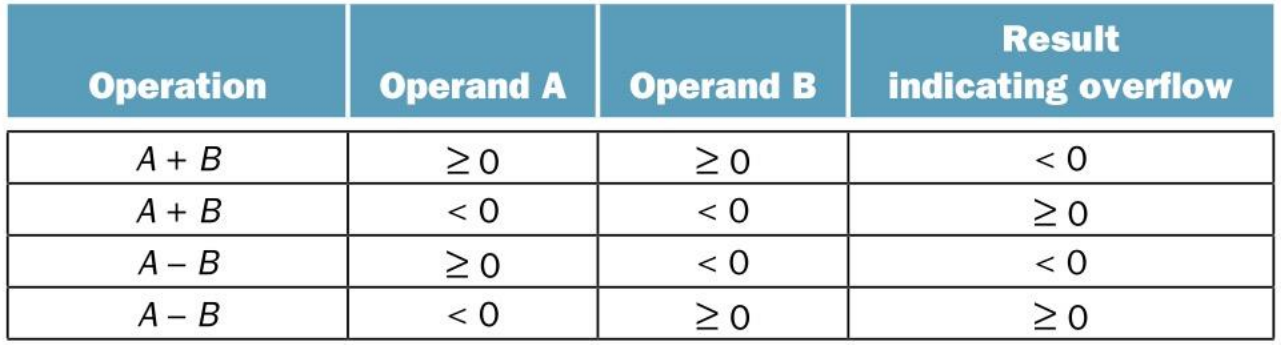
  Example: 8 – 6 = 8 + (–6)

  ```
  +8: 0000 0000 … 0000 1000
  –6: 1111 1111 … 1111 1010
  +1: 0000 0000 … 0000 0010
  ```

Overflow if result out of range

The two operands have different sign bit have the change to overflow

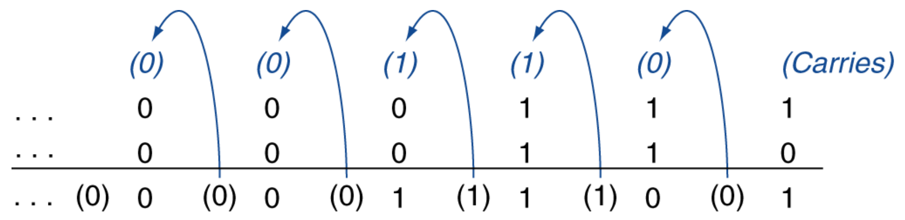
Overflow Conditions



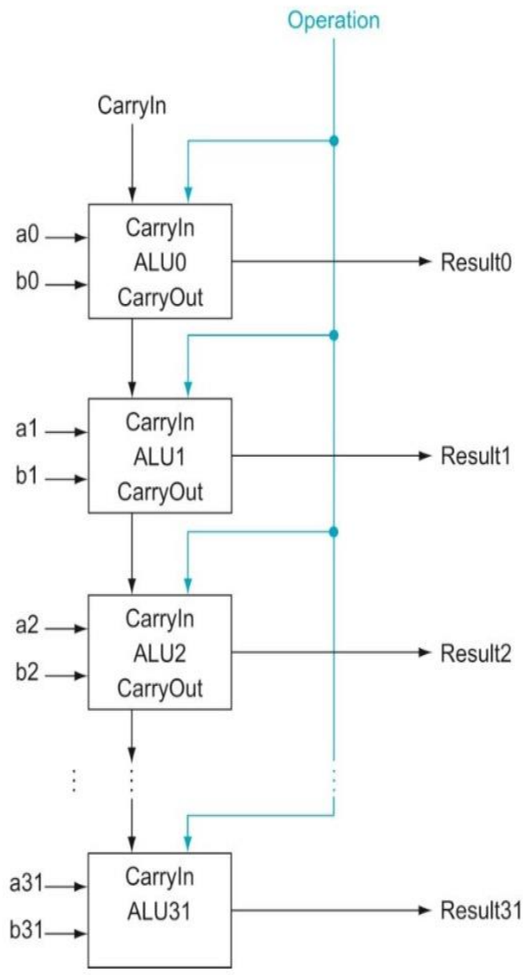
Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

FIGURE 3.2 Overflow conditions for addition and subtraction.

In hardware



(0) (0) (1) (1) (0) (Carries)
. . . 0 0 0 1 1 1
. . . 0 0 0 1 1 0
. . . (0) 0 (0) 0 (0) 1 (1) 1 (1) 0 (0) 1



Operation
CarryIn
a0
b0
CarryIn
ALU0
CarryOut
Result0
a1
b1
CarryIn
ALU1
CarryOut
Result1
a2
b2
CarryIn
ALU2
CarryOut
Result2
a31
b31
CarryIn
ALU31
Result31

Multiplication

- Start with long-multiplication approach，

```
multiplicand  →     1000ten
multiplier    →   × 1001ten
                  ---------
                    1000
                   0000
                  0000
                 1000
                 ---------
product       →  1001000ten
```

With only two choices, each step of the multiplication is simple:
1. Just place a copy of the multiplicand (1 × multiplicand) in the proper place if the multiplier digit is a 1, or
2. Place 0 (0 × multiplicand) in the proper place if the digit is 0.

Length of product is the sum of operand lengths(n + m represent all possible product)

Multiplication Hardware

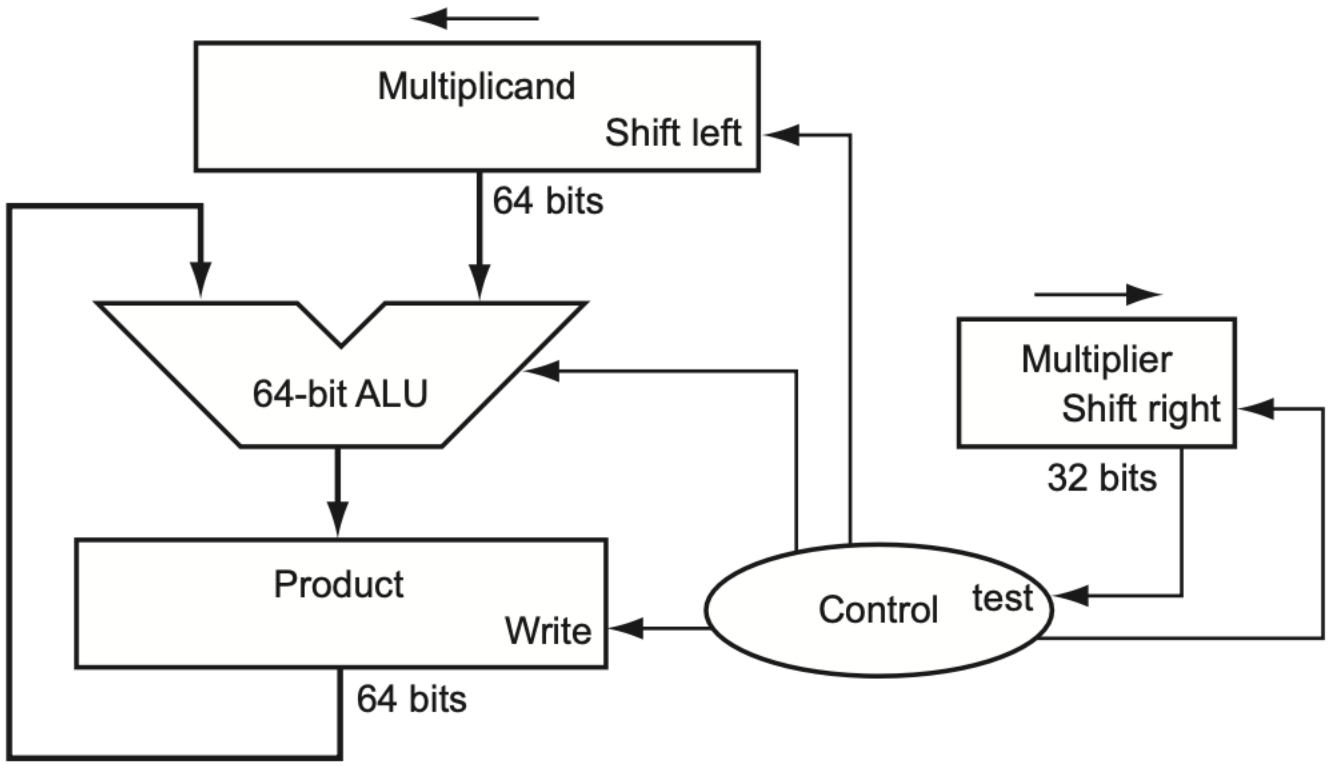
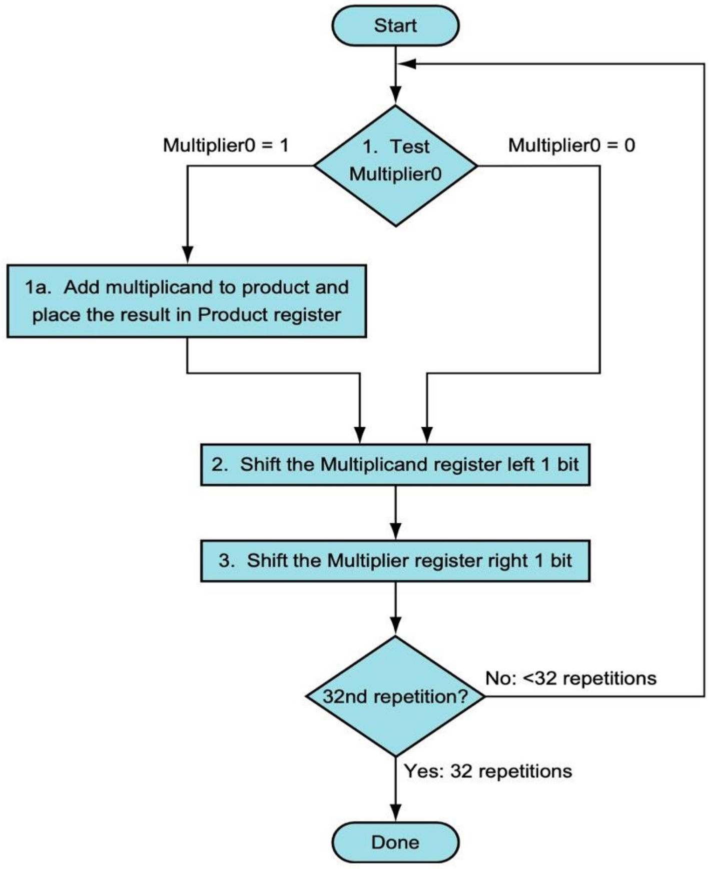


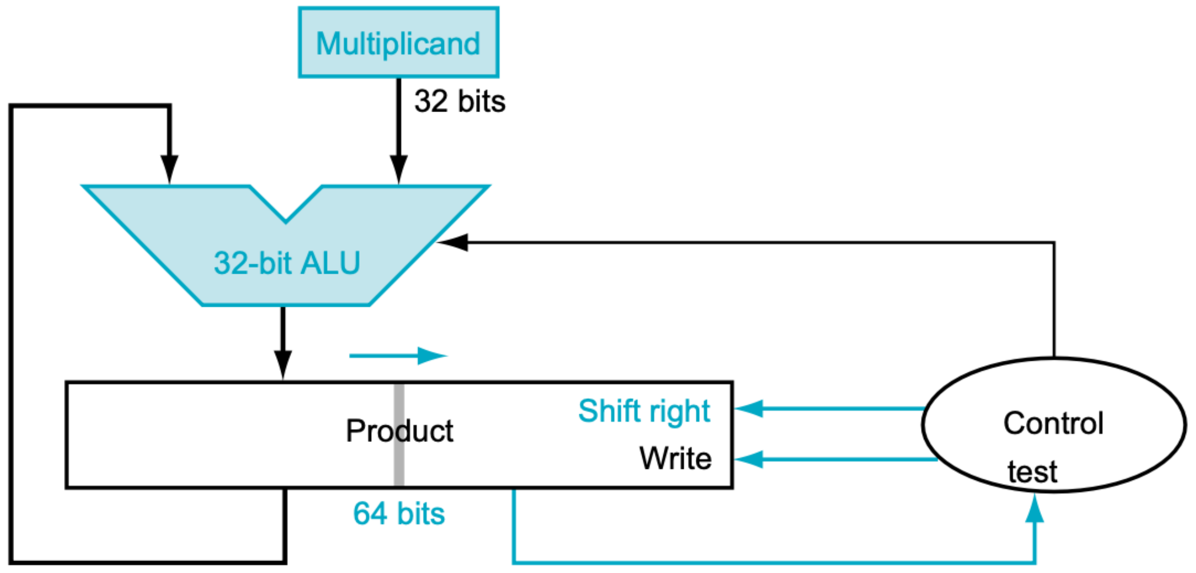
FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix A describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.



Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: <32 repetitions
Yes: 32 repetitions
Done

FIGURE 3.4 The first multiplication algorithm, using the hardware shown in Figure 3.3. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

Multiplication Hardware



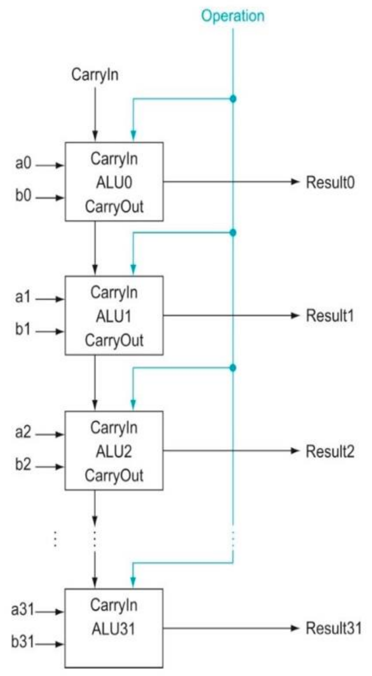
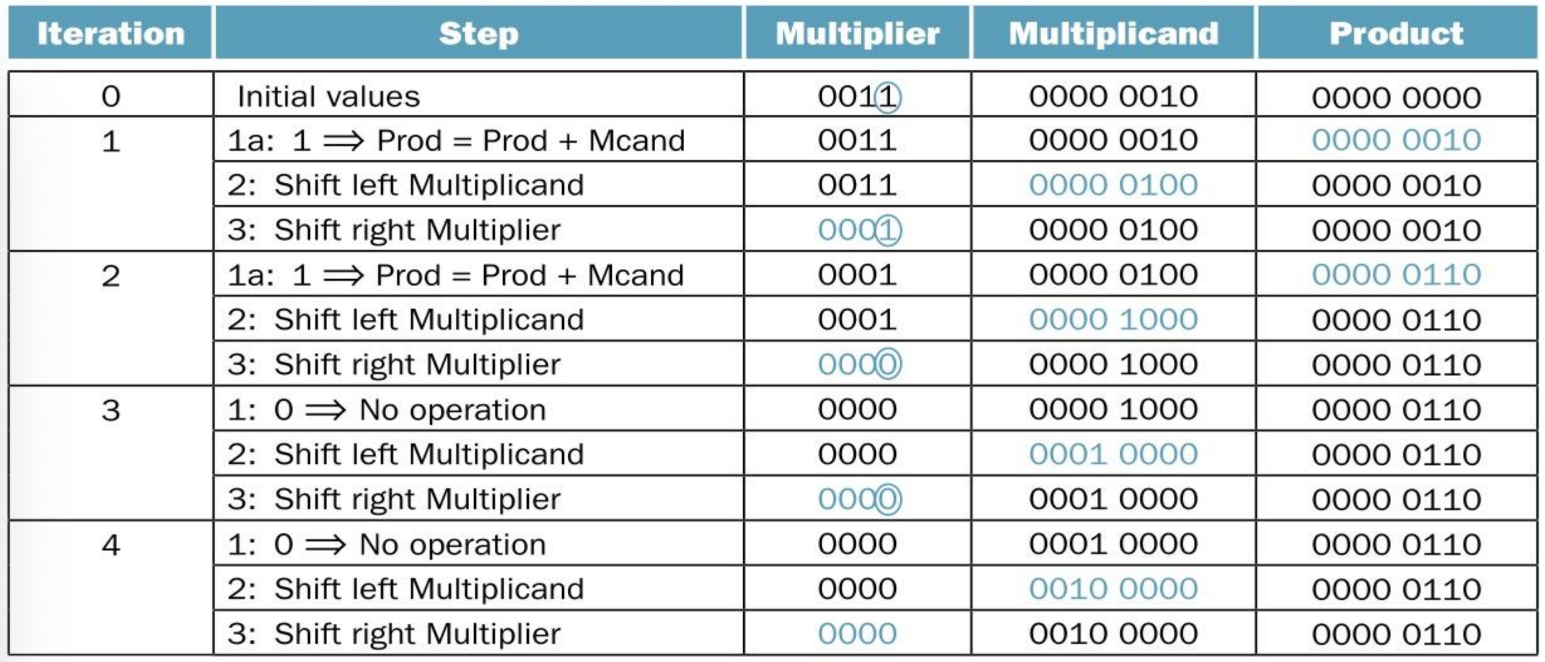


FIGURE 3.5 Refined version of the multiplication hardware. Compare with the first version in Figure 3.3. The Multiplicand register and ALU have been reduced to 32 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register, which has grown by one bit to 65 bits to hold the carry-out of the adder. These changes are highlighted in color

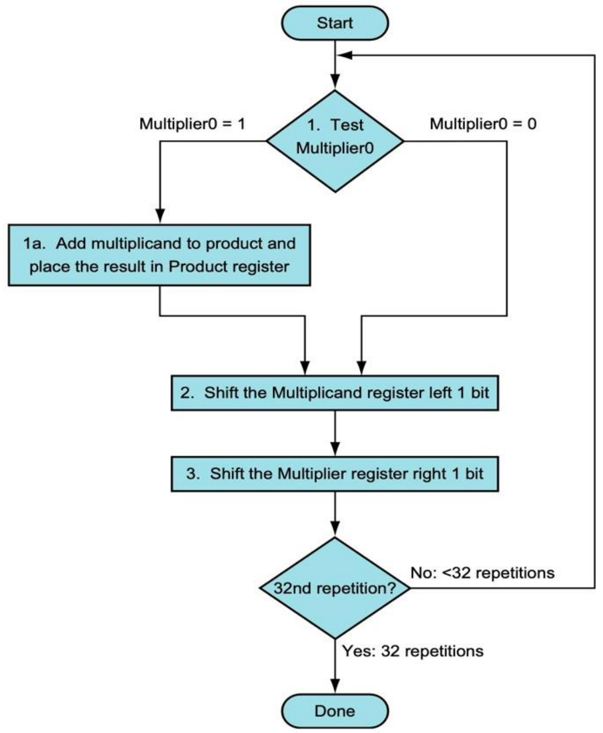
Example

Using 4-bit numbers to save space, multiply 2ten × 3ten, or 0010two × 0011two



Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⟹ 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 ⟹ 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	1: 0 ⟹ 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	1: 0 ⟹ 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

FIGURE 3.6 Multiply example using algorithm in Figure 3.4. The bit examined to determine the next step is circled in color.



Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: <32 repetitions
Yes: 32 repetitions
Done

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff

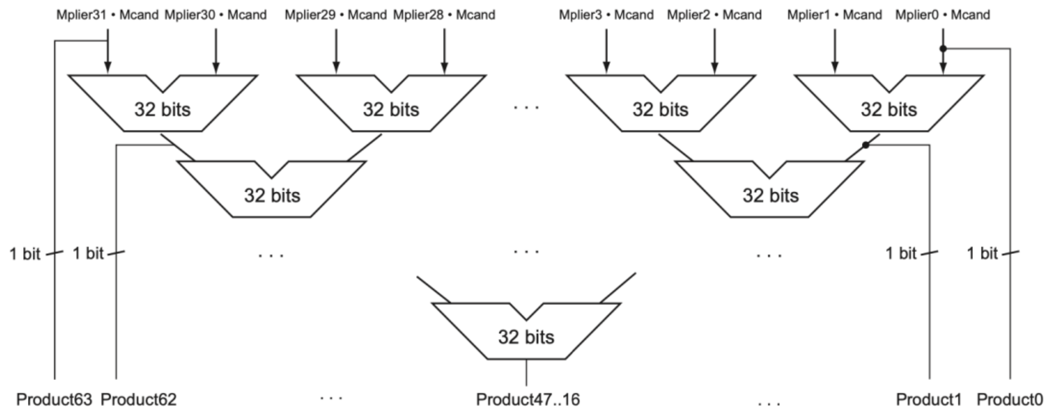


FIGURE 3.7 Fast multiplication hardware. Rather than use a single 32-bit adder 31 times, this hardware "unrolls the loop" to use 31 adders and then organizes them to minimize delay.

Can be pipelined Several multiplication performed in parallel

RISC-V Multiplication

- Four multiply instructions:
  - mul: multiply
    - Gives the lower 64 bits of the product
  - mulh: multiply high
    - Gives the upper 64 bits of the product, assuming the operands are signed
  - mulhu: multiply high unsigned
    - Gives the upper 64 bits of the product, assuming the operands are unsigned
  - mulhsu: multiply high signed/unsigned
    - Gives the upper 64 bits of the product, assuming one operand is signed and the other unsigned
  - Use mulh result to check for 64-bit overflow