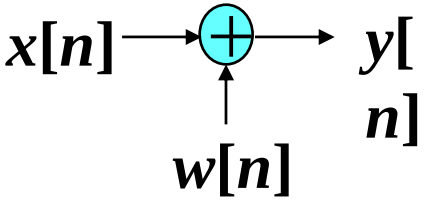
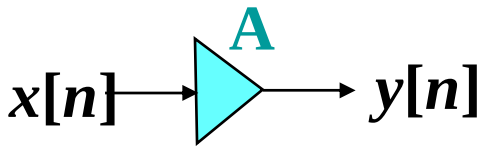
Chapter 10 Architectures for Arithmetic Processors

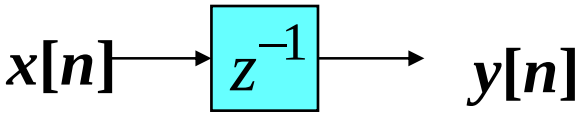
- Many algorithms require repeated execution of arithmetic operations, so **it is important that they be implemented efficiently**
- Examining **algorithms and architectures** for implementing addition, subtraction, multiplication, and division of **fixed-point numbers**



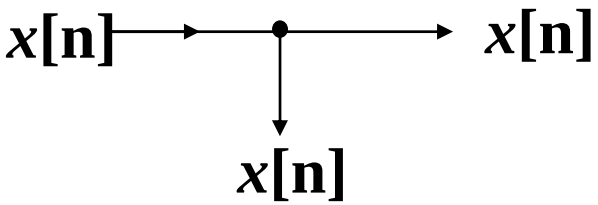
Adder



Multiplier



Unit delay



Pick-off node

10.1 Number Representation

A binary number system has **two symbols and a radix of 2.**

Ex: an n-bit unsigned binary number is represented as B= $b_{n-1}b_{n-2}....b_1b_0$, $b_i \in \{0,1\}$

- A binary number can be expressed as a weighted sum of ascending and descending powers of 2:

with decimal value $B_{10} = \sum_{i=-m}^{i=n-1} b_i 2^i$

Signed numbers

There are three common formats for signed numbers:

- Signed magnitude
- 1s complement
- 2s complement

10.1.1 Singed Magnitude Representation of Negative Integers

- The MSB of a word is the encoded sign bit:

  0 representing a positive value;

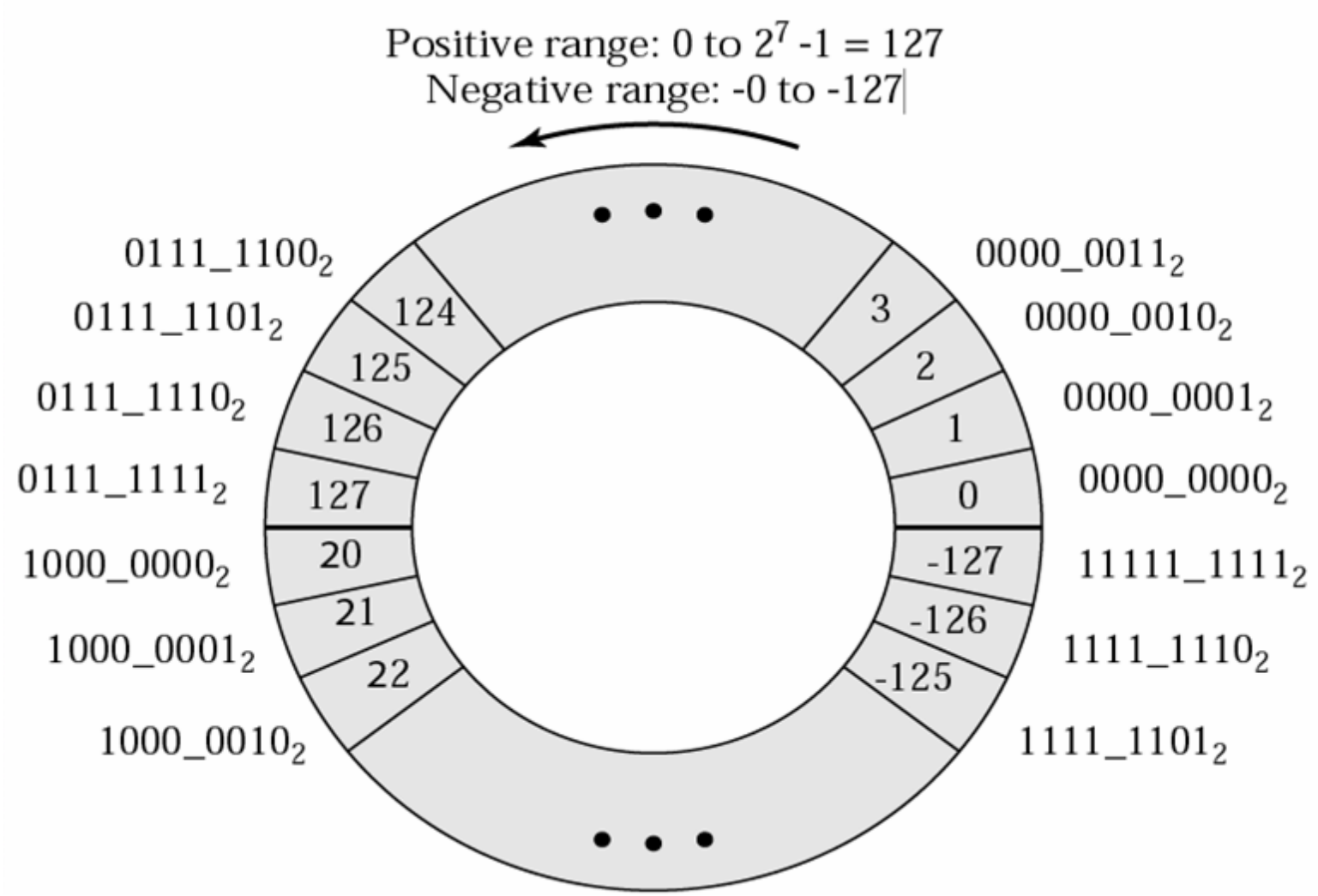
  1 representing a negative value.

- 0 has two representations:

  +0: 0000_0000

  -0: 1000_0000

Fig. 10-1 The numbers wheel of 8 bit signed-magnitude numbers



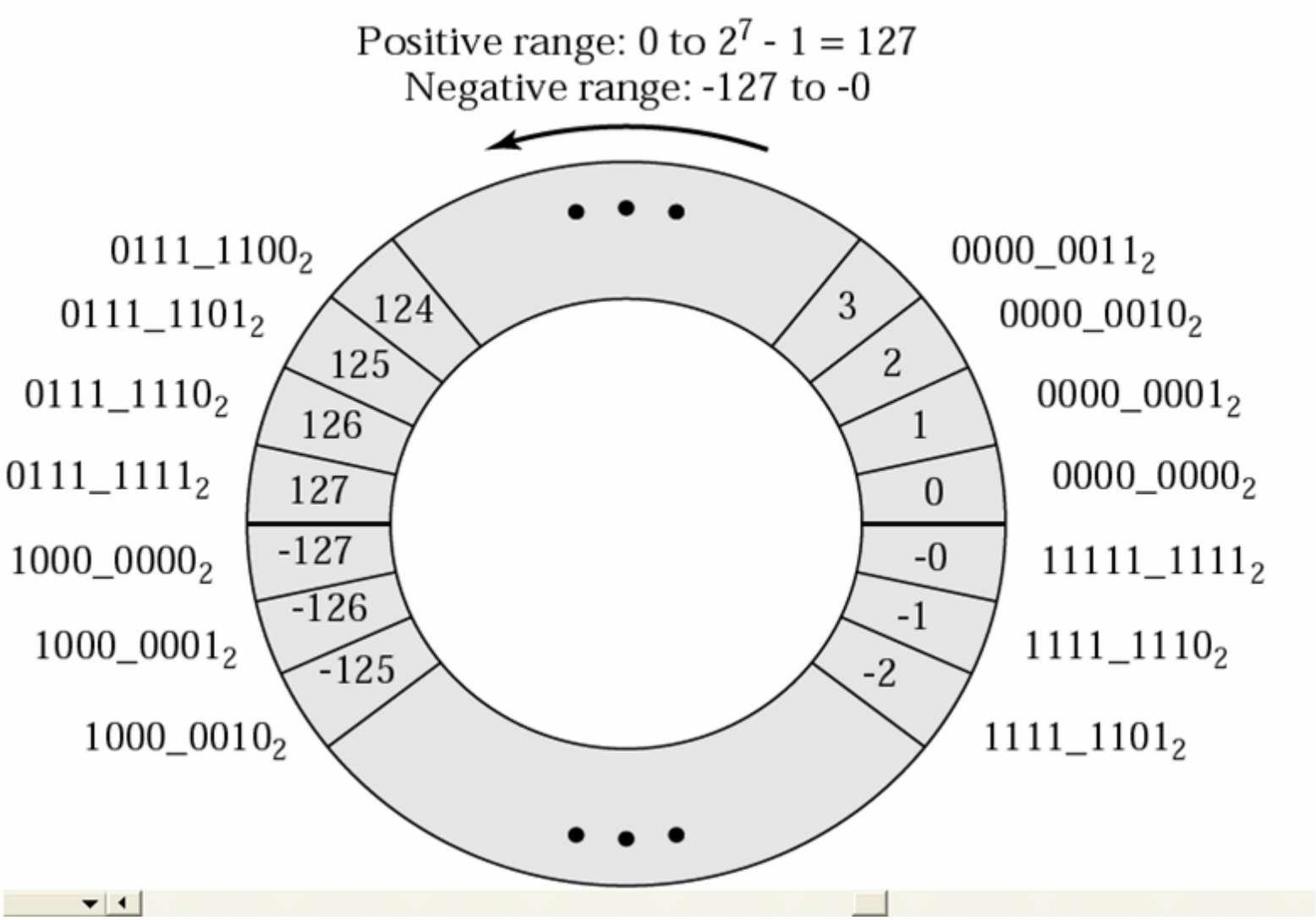
10.1.2 Ones Complement Representation of Negative Integers

- -B denotes the 1s complement of an n-bit binary number B:

$$B + (-B) = 2^n - 1$$

- Positive numbers are represented in a 1s complement system in the same way they do in a signed-magnitude system
- But negative 1s complement numbers are represented differently, in a 1s complement format

1s complement numbers



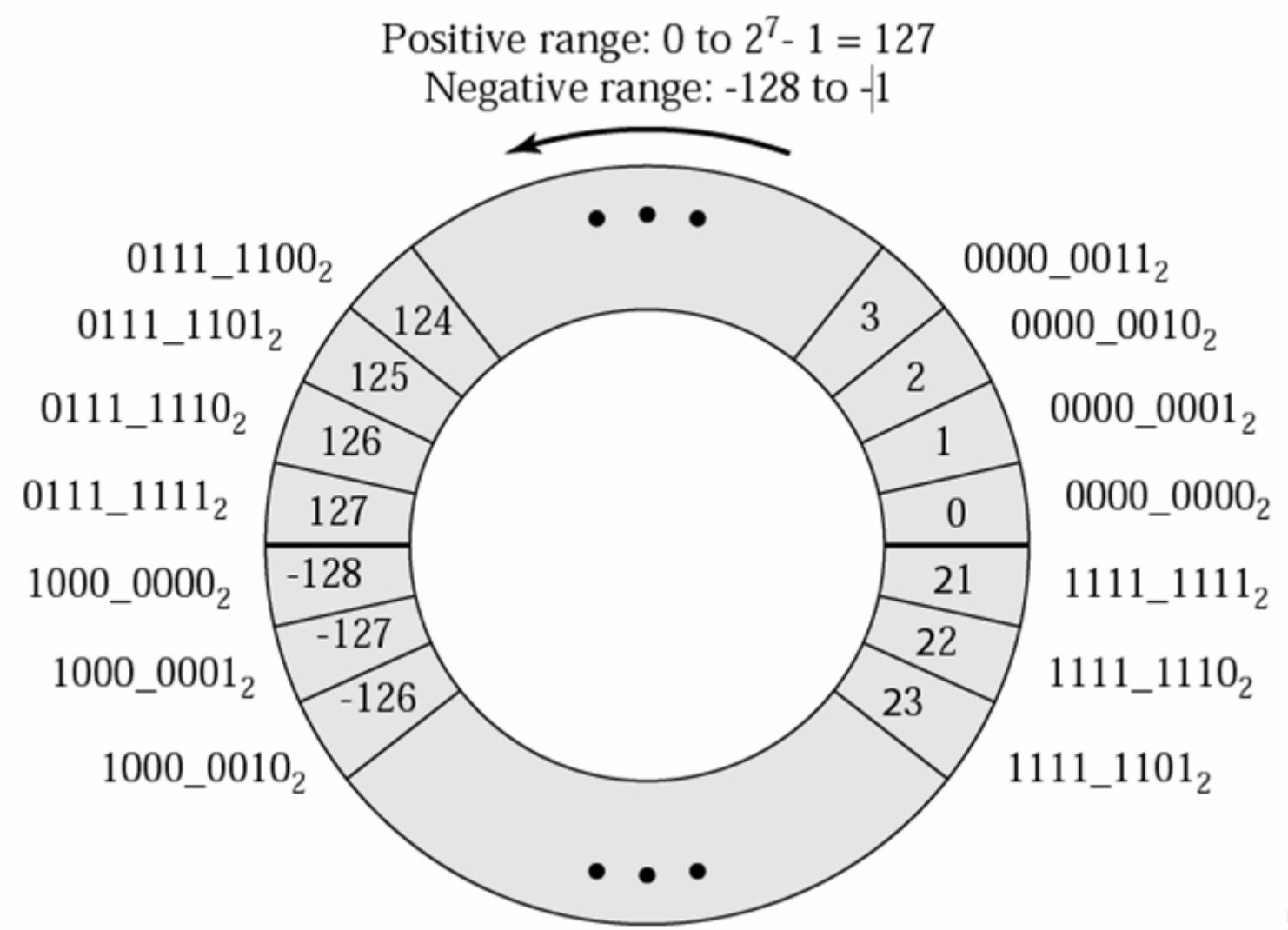
10.1.3 Twos Complement Representation of Positive and Negative Integers

- The 2s complement of an n-bit binary integer is defined as:

$$B^* = 2^n - B$$

or: $$B + B^* = 2^n = 0 \bmod n$$

2s complement numbers



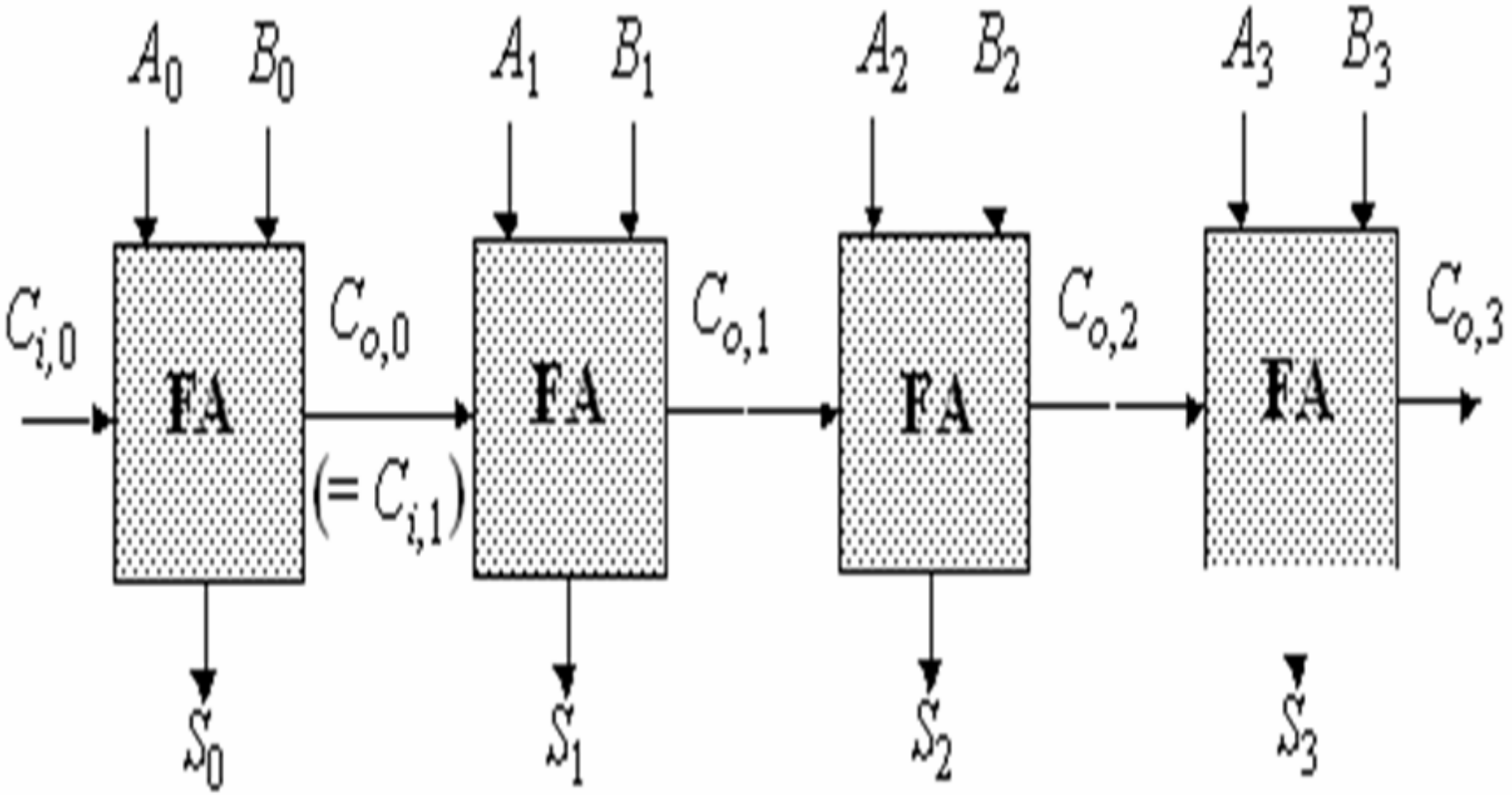
10.2 Functional Units for Addition and Subtraction

- Addition and subtraction are implemented in all arithmetic processors
- There are several alternatives that provide a tradeoff between hardware cost and performance

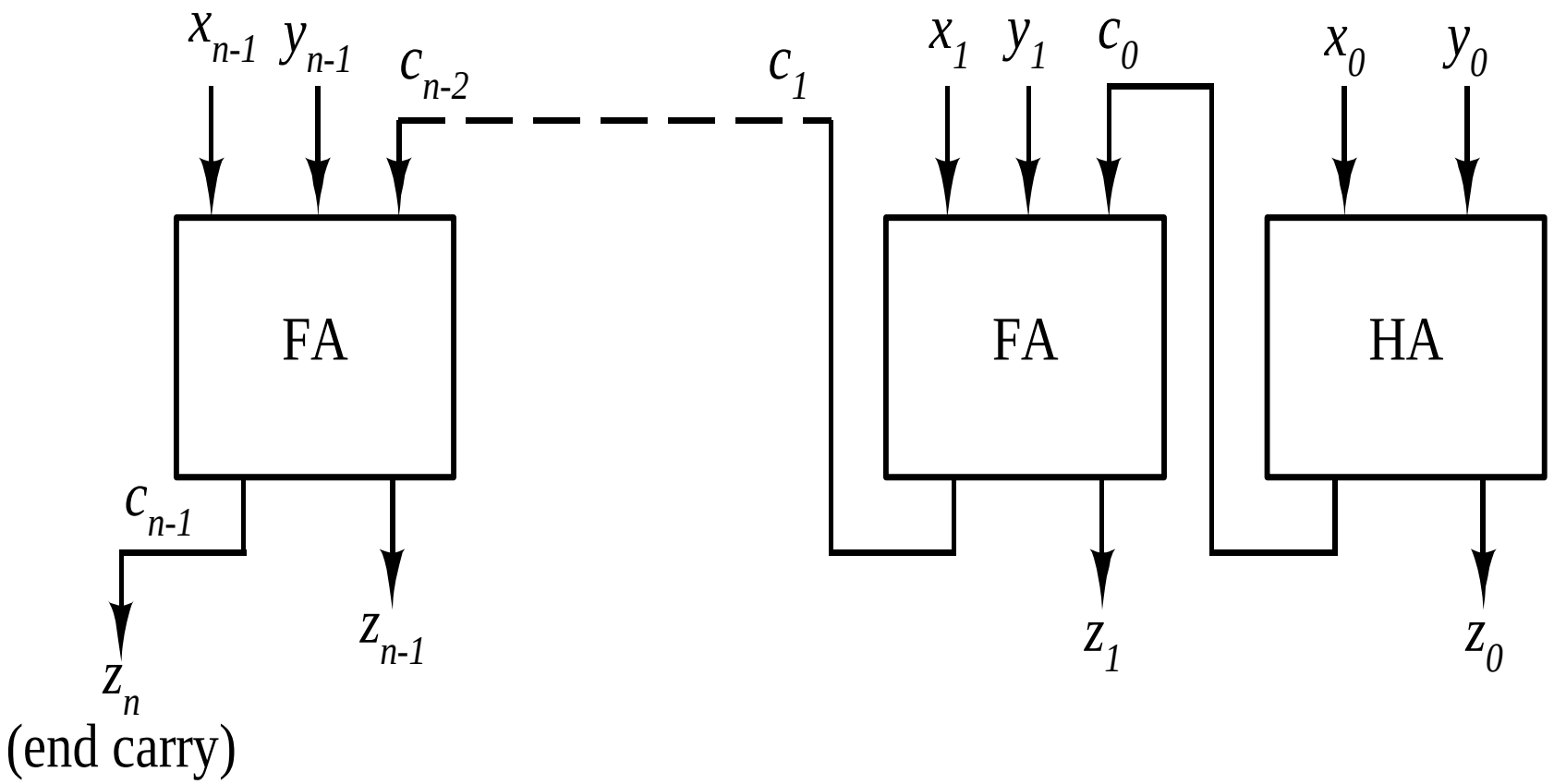
10.2.1 Ripple-Carry Adder

- A 16-bit ripple-carry adder can be formed by cascading four 4-bit ripple-carry adders in a chain
- The carry generated by a unit is passed to the carry input port of its neighbor, beginning with the least significant bit

Ripple-Carry Adder



Ripple Carry Adder



Alternative architecture to form the outputs quickly

- If the word length of the processor using the adder is large, it might be necessary to use an alternative architecture to form the outputs quickly enough to satisfy timing constraints
- Pipelining the dataflow is one alternative, but it introduces latency and requires hardware to implement the pipeline registers
- Another alternative is to consider other algorithms for addition， among those that are used are :

  the carry look-ahead algorithm

  the carry select algorithm

  the carry-save algorithm

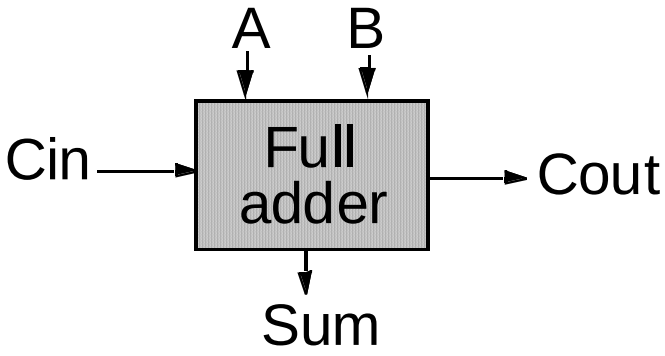
10.2.2 Carry Look-Ahead Adder

- The algorithm is based on the observation that the value of the carry into any stage of a multicell adder depends on only the data bits of the previous stages and the carry into the first stage
- This relationship can be exploited to improve the speed of the adder by using additional logic to implement the carry, rather than waiting for the value to propagate through the cells of the adder

Generate a Carry & Propagate a carry

- A given cell is said to generate a carry if both of the cell's data bits are 1
- A cell is said to propagate a carry if either of the cell's data bits could combine with the carry into the cell to cause a carry out to the next stage of the adder

Full-Adder



A	B	C_i	S	C_o	*Carry status*
0	0	0	0	0	delete
0	0	1	1	0	delete
0	1	0	1	0	propagate
0	1	1	0	1	propagate
1	0	0	1	0	propagate
1	0	1	0	1	propagate
1	1	0	0	1	generate
1	1	1	1	1	generate

Carry Look-Ahead Adders -- Basic Idea

Recall that $c_i = a_i b_i + a_i c_{i-1} + b_i c_{i-1}$

$= a_i b_i + a_i b_i c_{i-1} + a_i b_i' c_{i-1} + a_i b_i c_{i-1} + a_i' b_i c_{i-1}$

$= a_i b_i + a_i b_i' c_{i-1} + a_i' b_i c_{i-1}$

$= a_i b_i + (a_i b_i' + a_i' b_i) c_{i-1}$

$= a_i b_i + (a_i \oplus b_i) c_{i-1}$

Let $g_i = a_i b_i$ [carry generate]

$p_i = a_i \oplus b_i$ [carry propagate]

Then $\boldsymbol{c_i = g_i + p_i c_{i-1}}$

$\boldsymbol{s_i = p_i \oplus c_{i-1}}$

Carry Look-Ahead Adders
-- Three-Bit Example

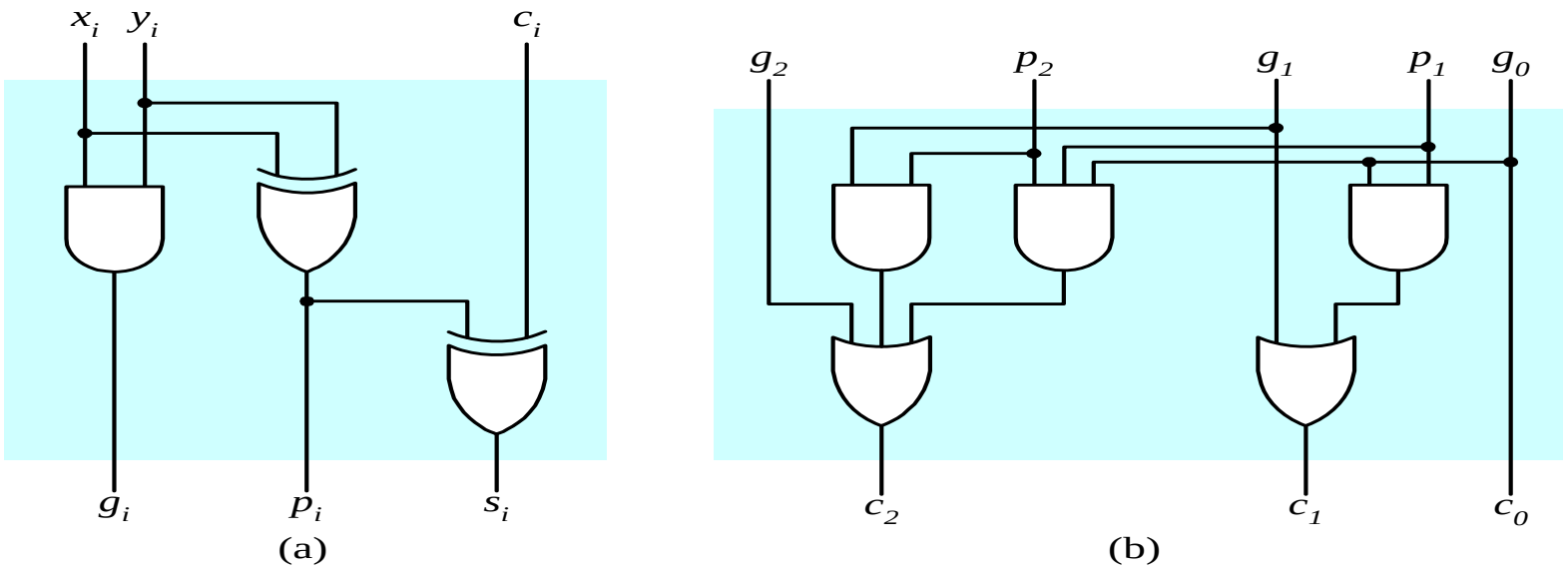
$$c_0 = g_0$$
$$s_0 = p_0$$

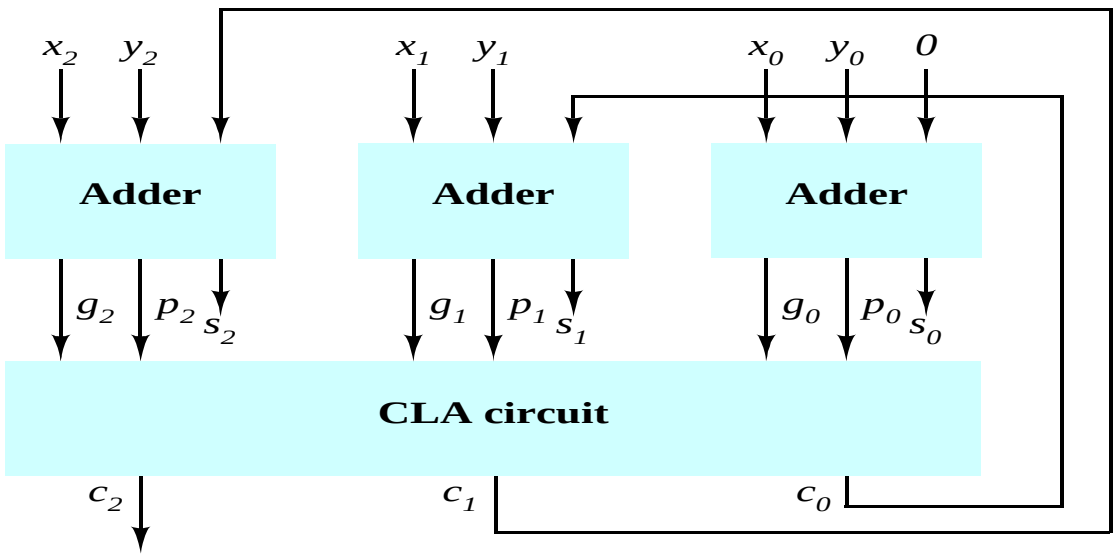
$$c_1 = g_1 + p_1 c_0$$
$$= g_1 + p_1 g_0$$
$$s_1 = p_1 \oplus c_0$$

$$c_2 = g_2 + p_2 c_1$$
$$= g_2 + p_2(g_1 + p_1 g_0)$$
$$= g_2 + p_2 g_1 + p_2 p_1 g_0$$
$$s_2 = p_2 \oplus c_1$$

$gi = ai bi,$
$pi = ai \oplus bi$

Carry Look-Ahead Adder Design





Carry Look-Ahead Adder

$c_0=0$, $c_{i+1} = a_ib_i+a_ic_i+b_ic_i = a_ib_i+(a_i+b_i)c_i$

Suppose :$g_i = a_ib_i$, $q_i = a_i+b_i$

Then:

$c_{i+1}=g_i+p_ic_i$

$=g_i+p_i(g_{i-1}+p_{i-1}c_{i-1}))$

$=g_i+p_i(g_{i-1}+(p_{i-1}(g_{i-2}+p_{i-2}c_{i-2}))$

...

$=g_i+p_i(g_{i-1}+p_{i-1}(g_{i-2}+p_{i-2}(\ldots(g_0+p_0c_0)\ldots)))$

$=g_i+p_ig_{i-1}+p_ip_{i-1}g_{i-2}+\ldots+p_ip_{i-1}\ldots p_1p_0+p_1g_0+p_ip_{i-1}\ldots p_1p_0c_0$

g_i,p_i depend on: a_i,b_i

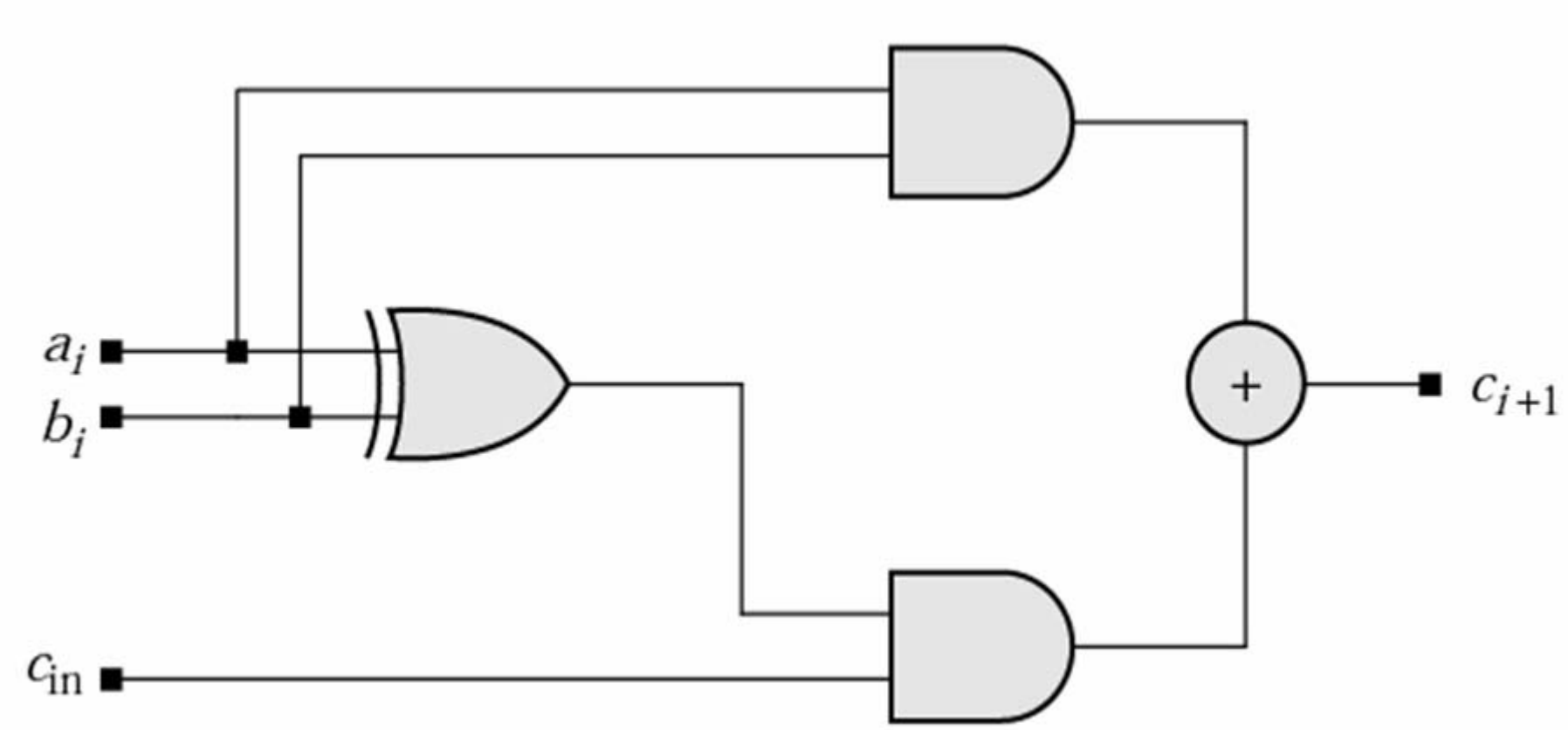
c_{i+1} depends on :$a_i,a_{i-1}..a_0,b_i,b_{i-1}\ldots b_0$, c_0

Carry Look-Ahead Adder

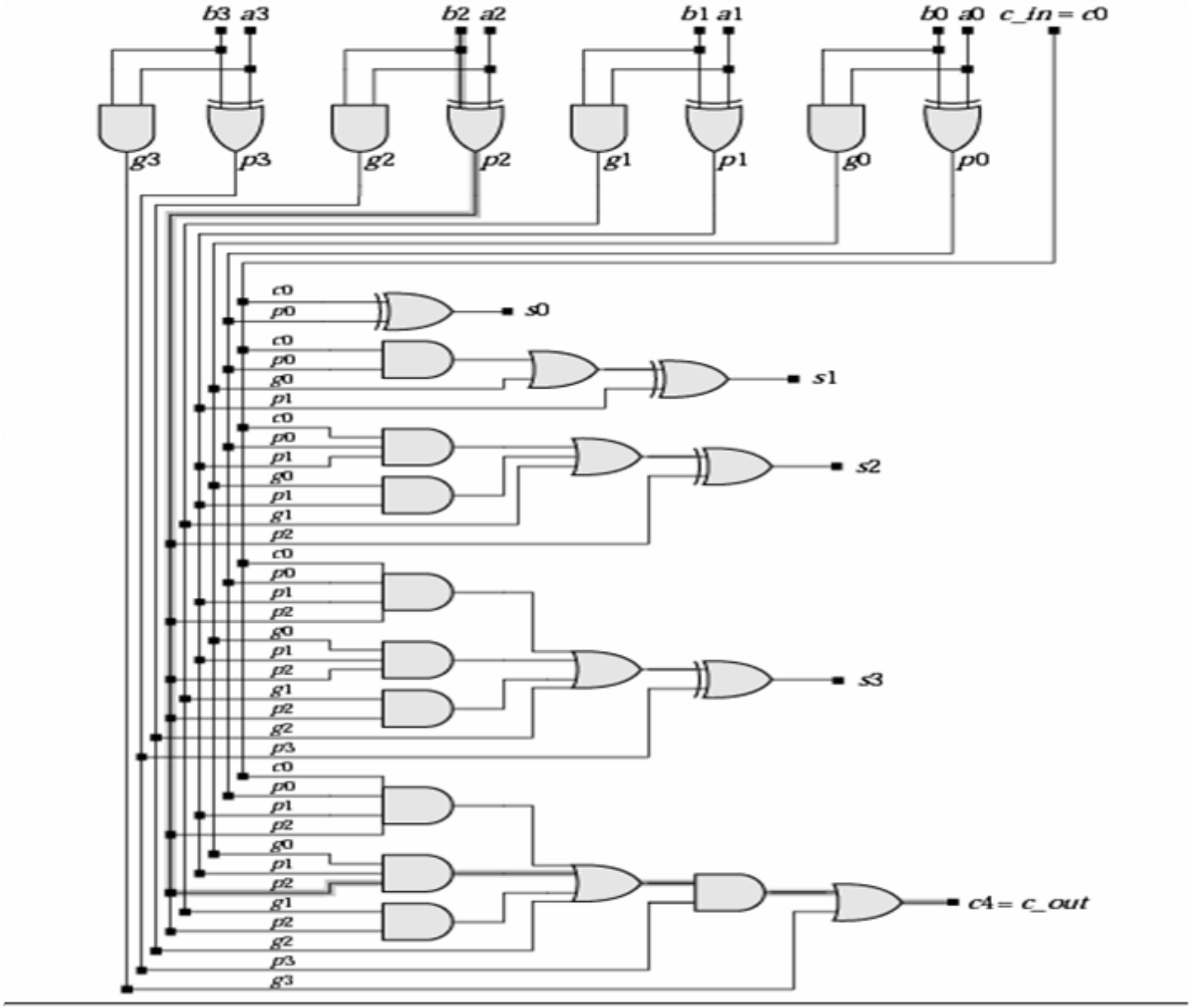
- The sum and the carry-out bits of each cell can be expressed in terms of the data bits of that cell and of the previous cells,and **the carry into only the first cell of the adder chain**.
- So there is ***no need*** **to wait for a carry bit to propagate through the adder to a particular cell**

Fig 10-6 Schematic for arithmetic implementation of

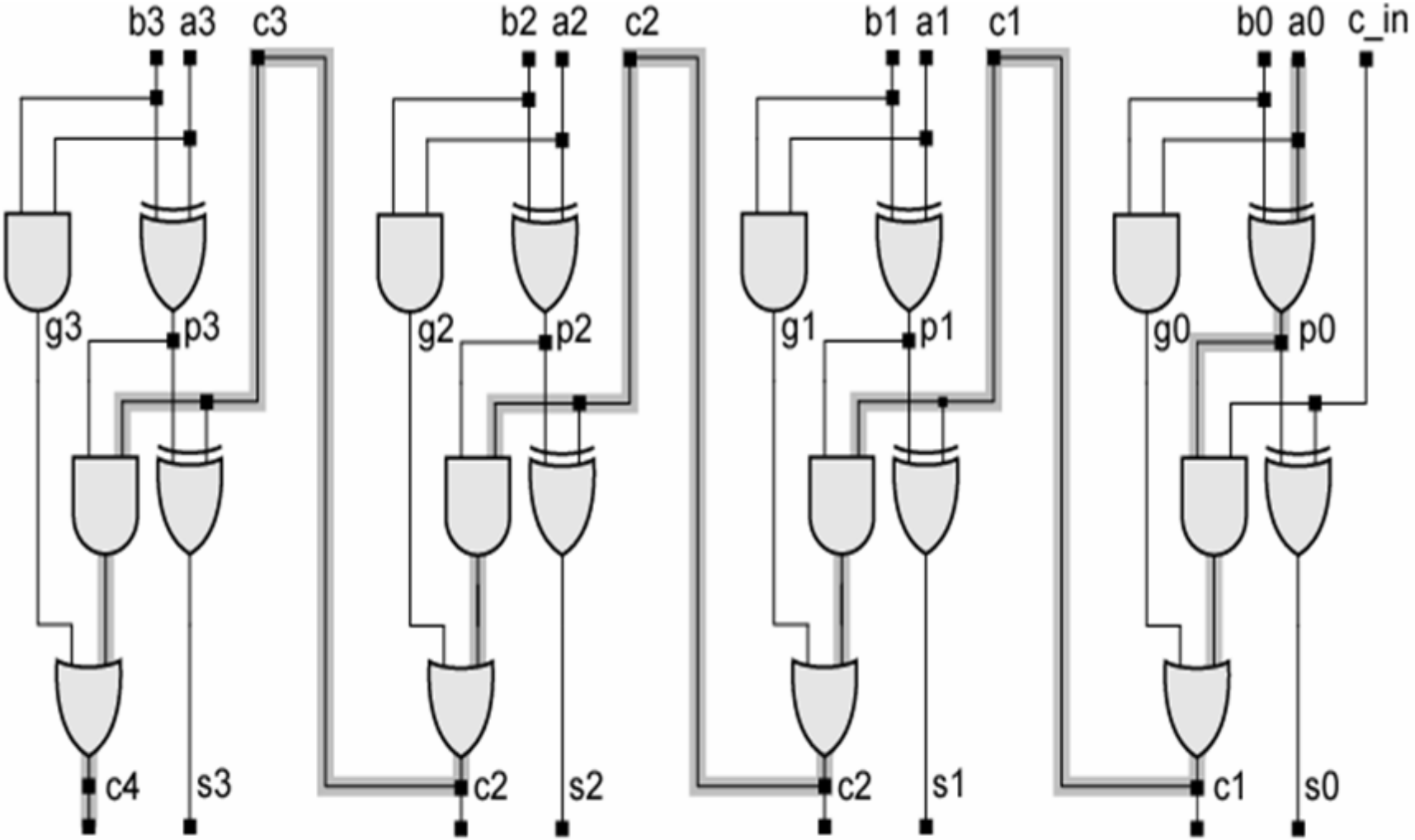
$$C_{i+1} = a_i \,\&\, b_i + (a_i \oplus b_i) \,\&\, c_i$$



Gate level circuit-carry look adder



Gate level circuit-ripple-carry adder



(b)

The propagate and generate recursive algorithm for a four-bit adder

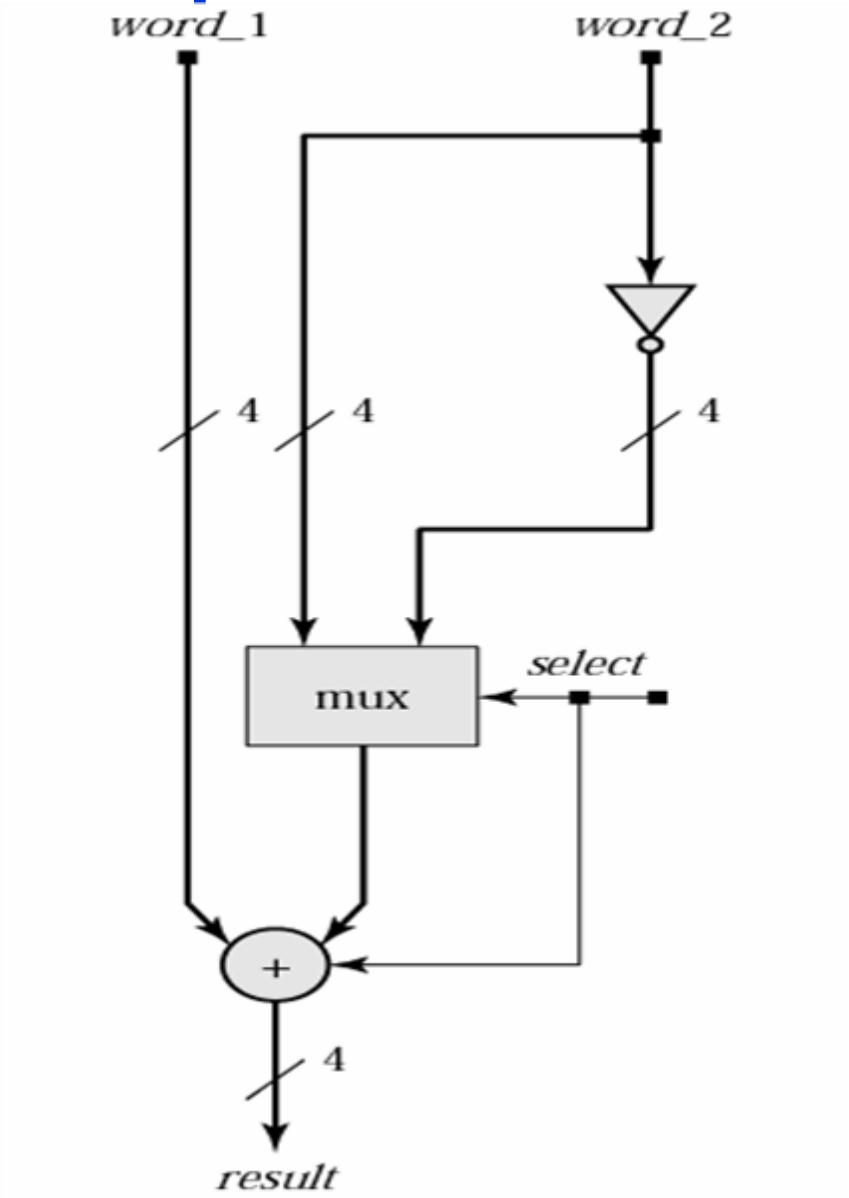
```
module Add_prop_gen (sum, c_out, a, b, c_in);
    output       [3: 0]   sum;
    output       c_out;
    input        [3: 0]   a, b;
    input        c_in;
    reg          [3: 0]   carrychain;
    integer      i;
    wire         [3: 0]   g = a & b;
      //carry generate, continuous assignment, bitwise and
    wire         [3: 0]   p = a ^ b;
      //carry propagate, continuous assignment, bitwise xor
```

```
always @ (a or b or c_in or p or g)
  begin: carry_generation
   integer i;
   carrychain[0] = g[0] + (p[0] & c_in);
   for(i = 1;i<= 3;i = i + 1)
    begin // Ci+1 =  ai & bi + (ai ⊕ bi) & ci
     carrychain[i] = g[i] I (p[i] & carrychain[i-1]);
    end
  end
  wire [4:0] shiftedcarry = {carrychain, c_in};
        //concatenation
  wire [3:0] sum = p ^ shiftedcarry;
       //summation
  wire c_out = shiftedcarry[4];
endmodule
```

Hardware subtraction units

- Hardware units usually implement subtraction by adding the 1s complement of the subtrahend to the minuend, and then adding 1 to the result
- This can be implemented with the architecture shown in Figure 10-9
- One adder unit can be used for addition or subtraction, depending on the value of the signal select

Fig 10-9 Architecture for a combined 4-bit datapath adder and subtractor unit



10.2.3 Overflow and Underflow

- **Occurs when assigning a value that is too large (overflow) or too small (underflow) to be held in a variable**
- Overflow occurs under two conditions:
  - (1) when adding two positive numbers produces a sum that exceeds the largest positive number that can be represented in the word length of the unit
  - (2) when adding two negative numbers produces a sum that is positive
- Arithmetic units include logic for underflow and overflow detection

10.3 Functional Units for Multiplication

- Multiplication can be **implemented with a combinational circuit or by a sequential circuit.**
- A combinational circuit require **more silicon area,but operate faster**.
- Sequential circuit require **less silicon area,but takes several clock cycles.**
- To discuss the differences between them and their own characters

10.3.1 Combinational (Parallel) Binary Multiplier

$$A \times B = \sum_{i=0}^{m-1} A_i 2^i \sum_{j=0}^{n-1} B_j 2^j$$

And we know:

$$(a_1 + a_2 + \cdots + a_m)(b_1 + b_2 + \cdots b_n)$$
$$= a_1 b_1 + a_1 b_2 + \cdots + a_1 b_n$$
$$+ a_2 b_1 + a_2 b_2 + \cdots + a_2 b_n + \cdots + a_m b_n$$

So:

$$A \times B = \sum_{i=0}^{m-1} \sum_{j-0}^{n-1} A_i B_j 2^{i+j}$$

The result can be written as a sum of weighted powers of 2:

$$A \times B = \sum_{k=0}^{m+n-1} P_k \, 2^k$$

Generally

we complement multiplier by adding shifted copies of the multiplicand.

The Binary Multiplication

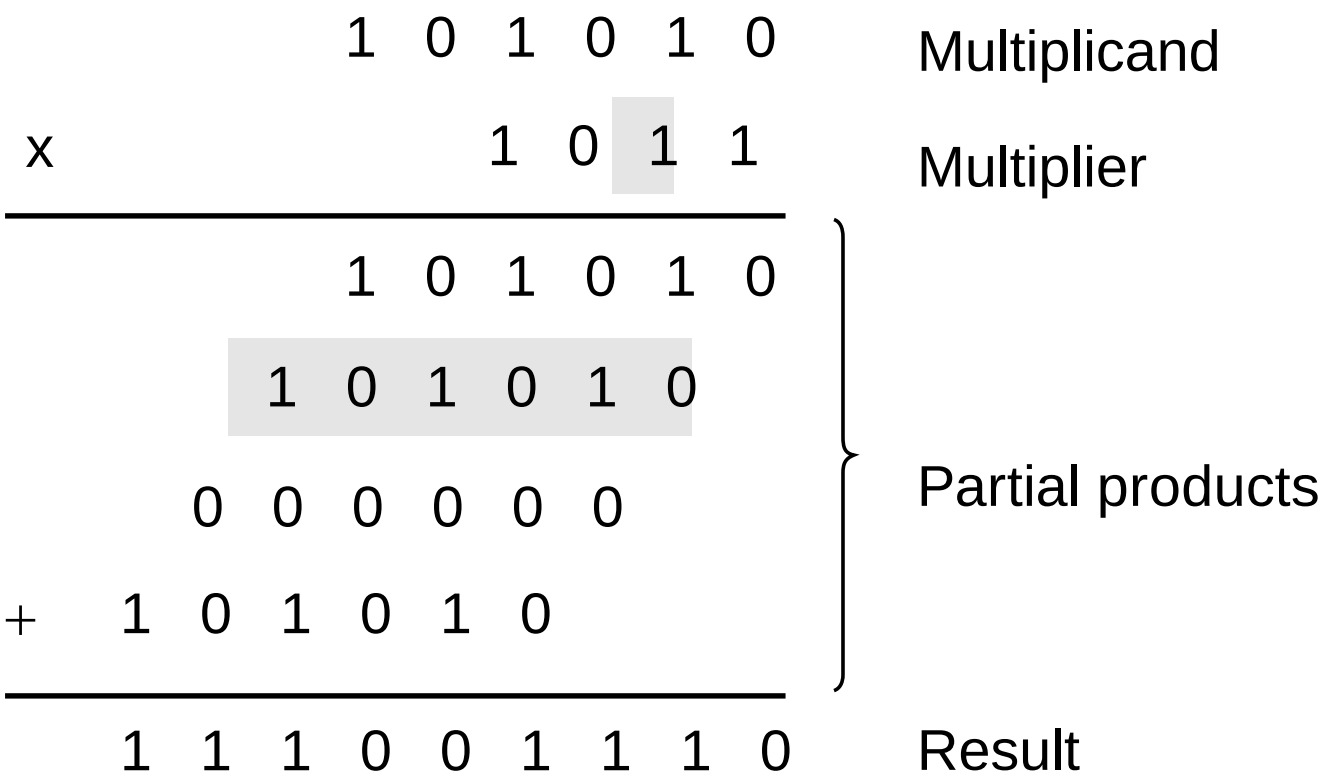
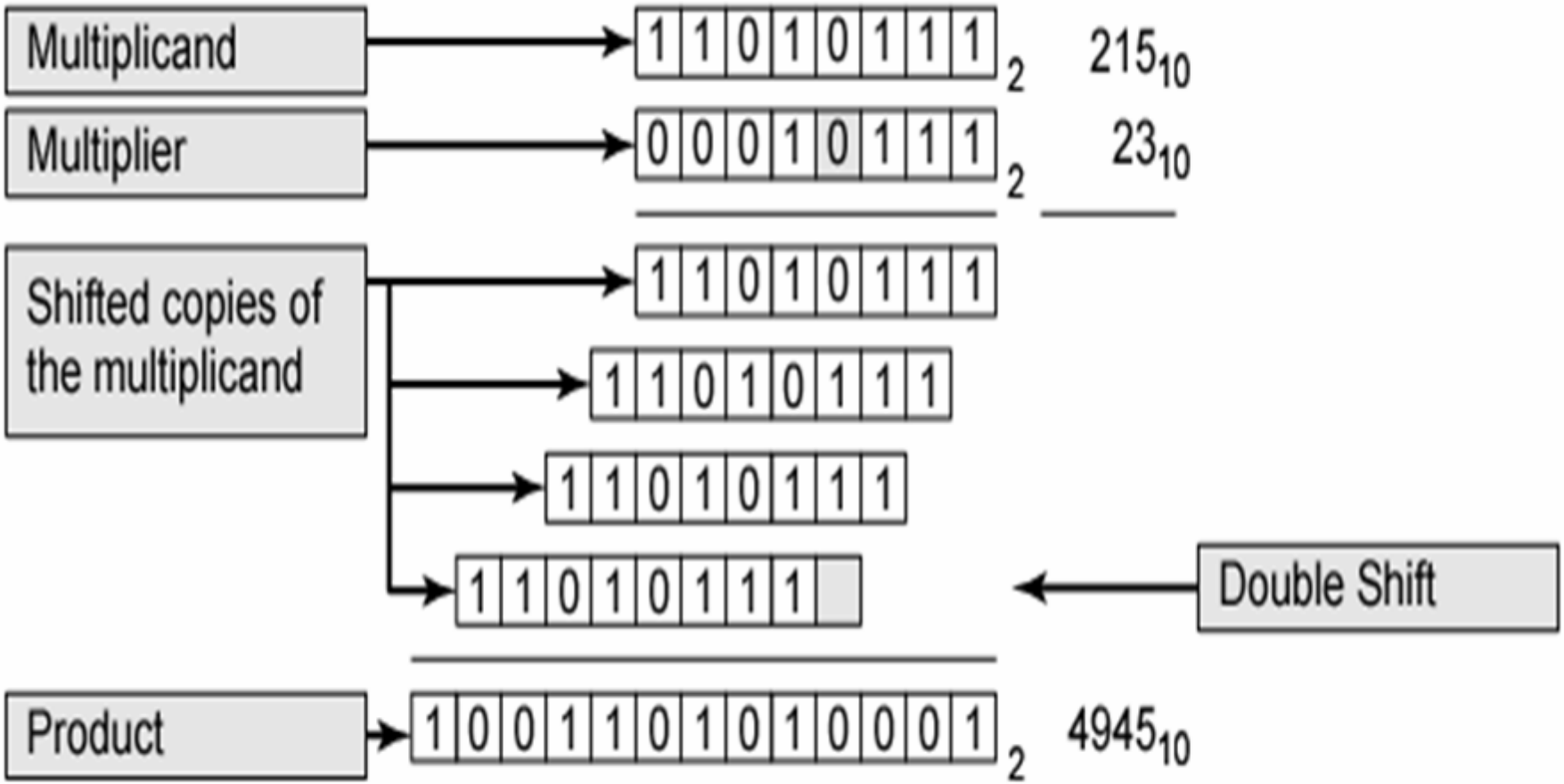


Fig 9-10 Formation of a product by columnwise adding shifted copies of the multiplicand



The process works manually

- The process shown in Figure 10-10 works manually, and a combinational circuit can be developed to implement the product, **with binary multiplication formed by an AND gate**
- **An ordinary adder operates on only two words at a time**, so a more attractive scheme that forms a sequence of row sums by adding a single copy of the multiplicand to an accumulated product will be presented here

Fig 9-11 Alternative sequential process forming the product of a pair of binary words by accumulating partial sums

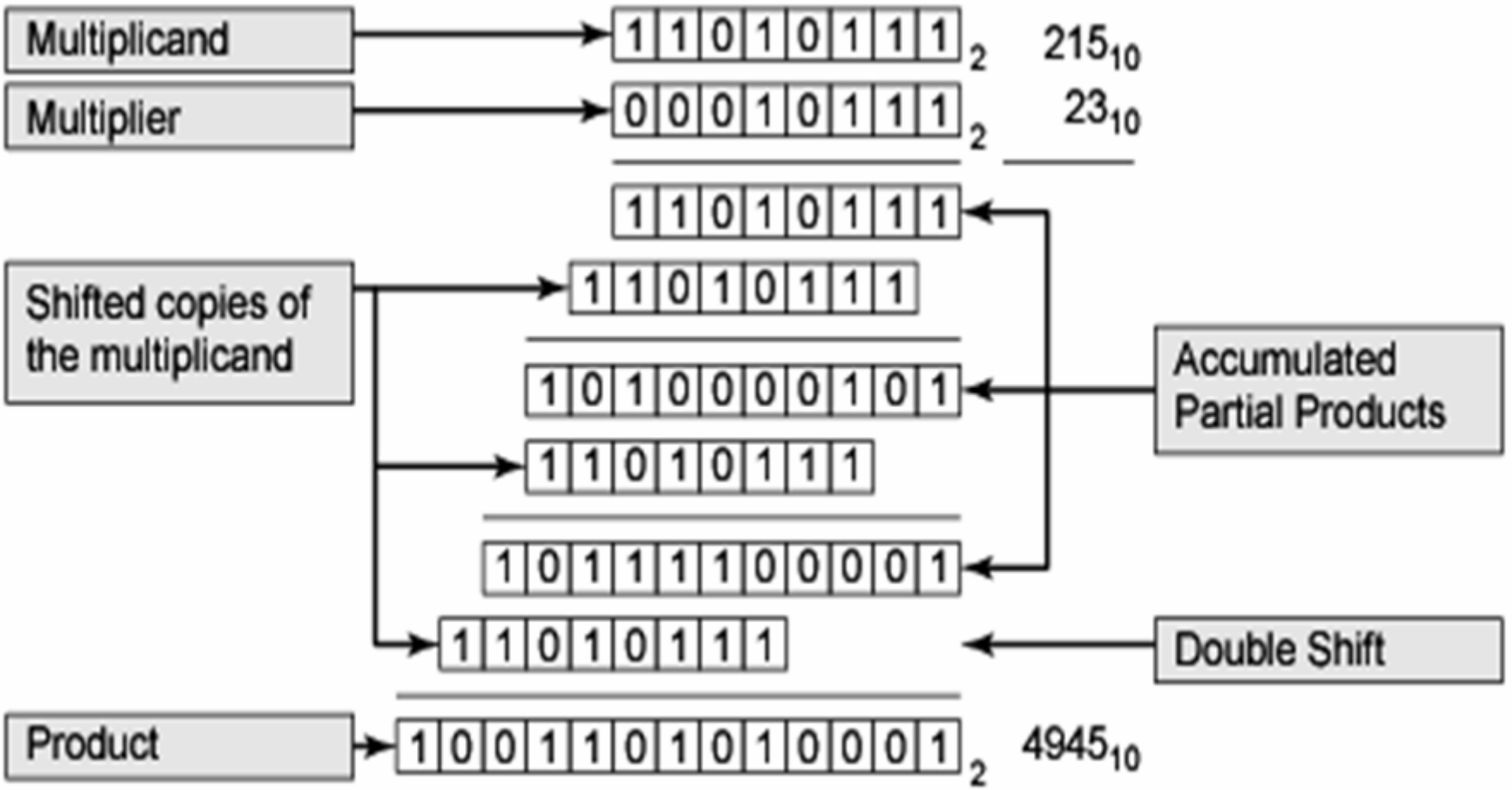
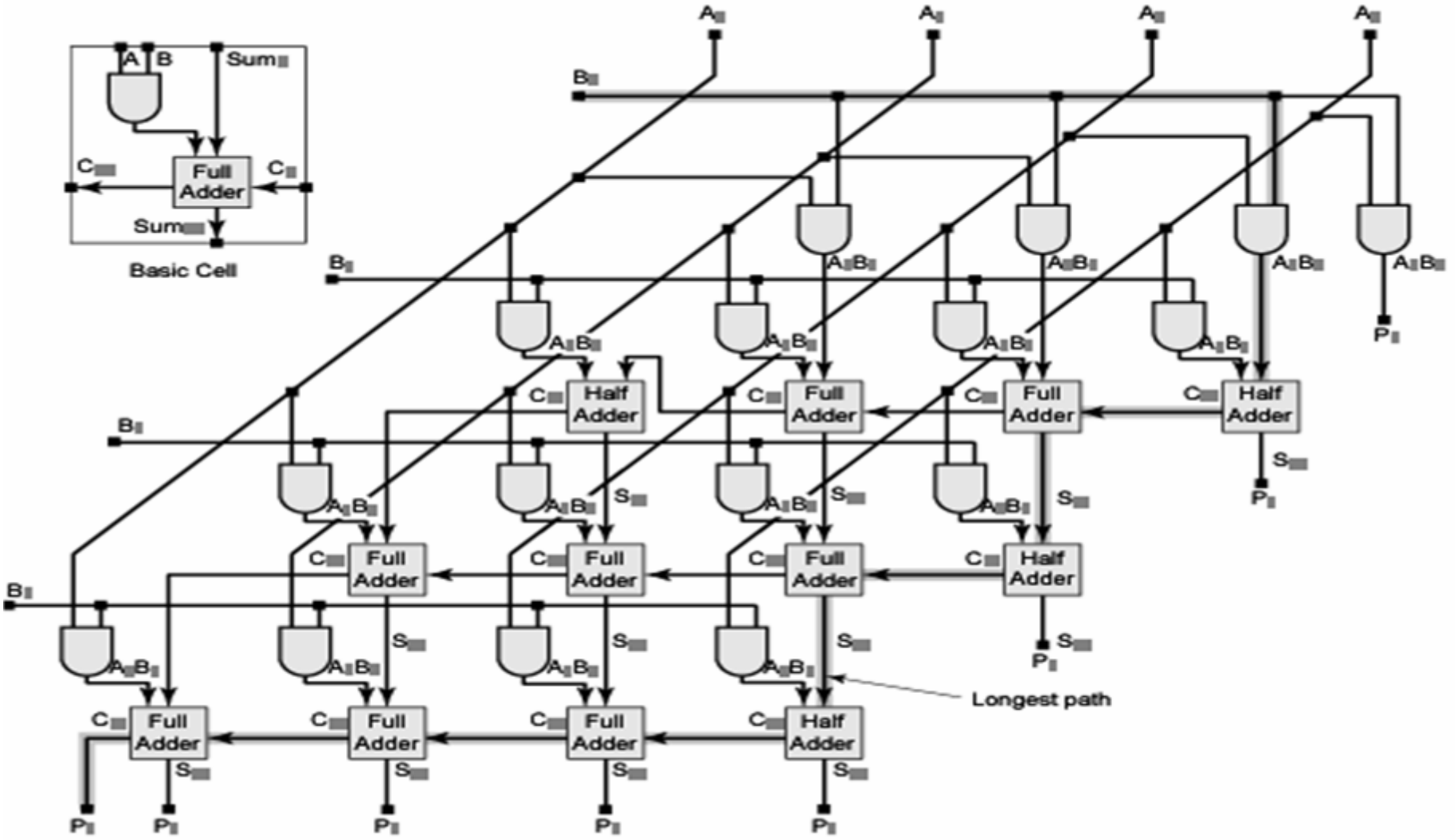


Fig 10-12 steps in the multiplication of unsigned 4-bit binary words

				A3 B3	A2 B2	A1 B1	A0 B0	Multiplicand Multiplier
			A_3B_1 C_{12}	A_3B_0 A_2B_1 C_{11}	A_2B_0 A_1B_1 C_{10}	A_1B_0 A_0B_1	A_0B_0 S_{00}	Partial Product 0 Partial Product 1 1st Row Carries
		C_{13} A_3B_2 C_{22}	S_{13} A_2B_2 C_{21}	S_{12} A_1B_2 C_{20}	S_{11} A_0B_2	S_{10}		1st Row Sums Partial Product 2 2nd Row Carries
	C_{23} A_3B_3 C_{32}	S_{23} A_2B_3 C_{31}	S_{22} A_1B_3 C_{30}	S_{21} A_0B_3	S_{20}			2nd Row Sums Partial Product 3 3rd Row Carries
C_{33}	S_{33}	S_{32}	S_{31}	S_{30}				3rd Row Sums
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	Final Product
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Weight

An array structure of glue logic ,half adder ,and full adder for a 4-bit binary



The Array Multiplier

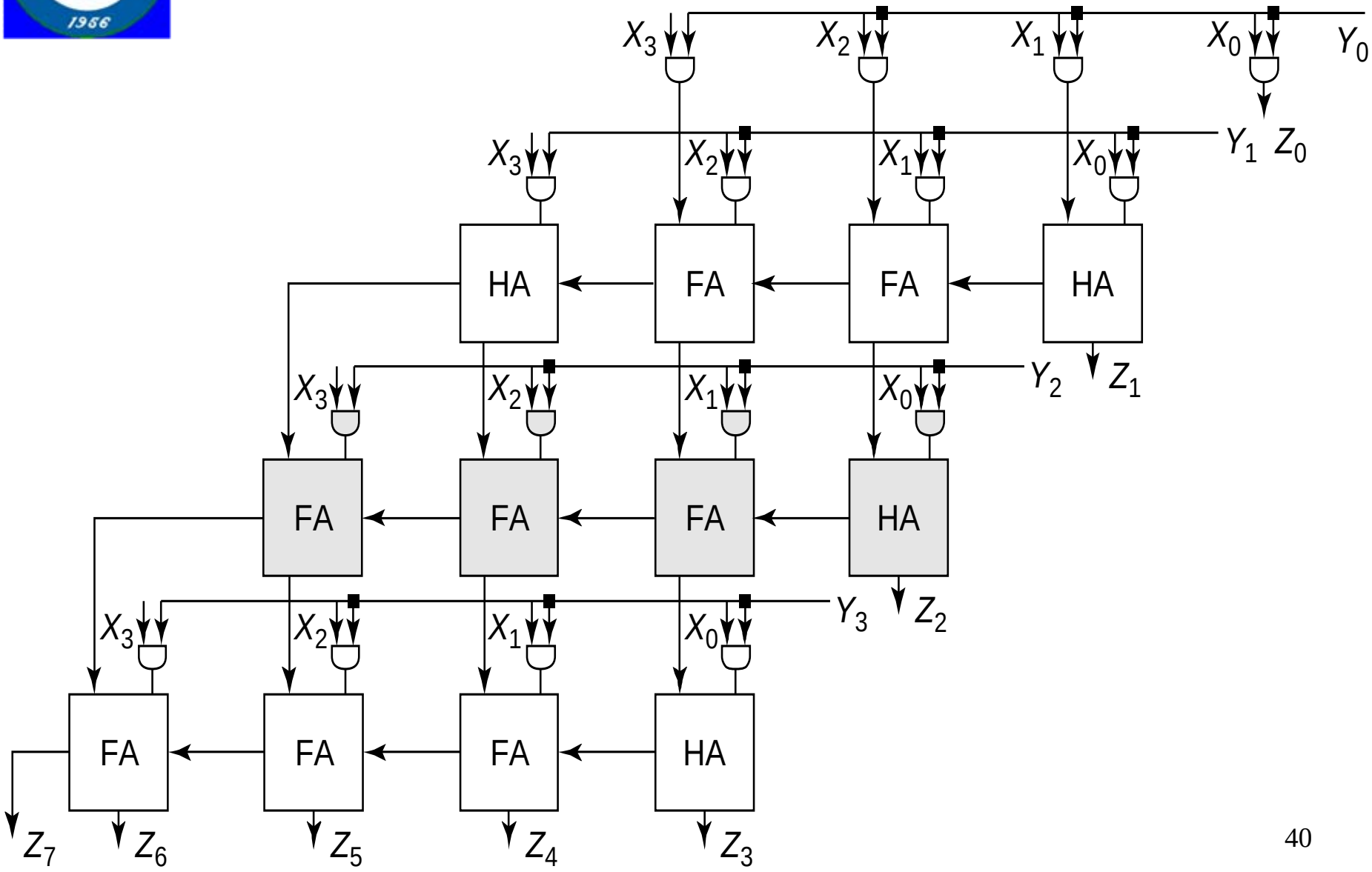
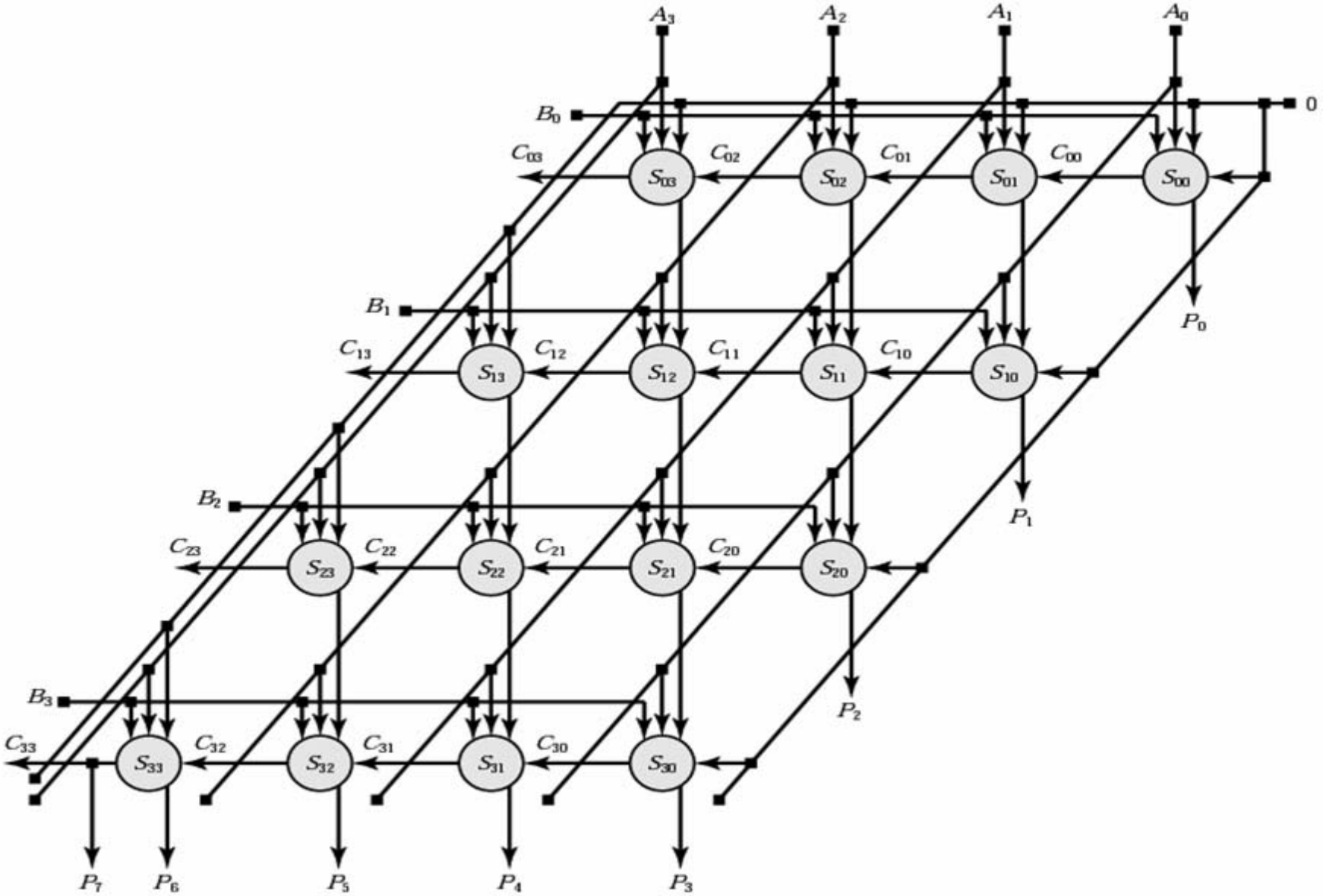


Fig 1-14 A systolic array structure for a 4-bit binary multiplier



The tradeoff for area and performance

- Both the carry and sum paths of the adders affect the longest path, and balanced delays through them are desirable
- The area of the device is relatively large, compared to other realizations, such as the sequential multipliers that will be considered
- But the extra area is the price of the superior performance of the combinational multiplier.

10.3.2 Sequential Binary Multiplier

- **If area is an important consideration, it can be reduced at the expense of performance by scheduling the sub-operations of the multiplier to execute in successive clock cycles**
- **The advantages of sequential multiplier**:

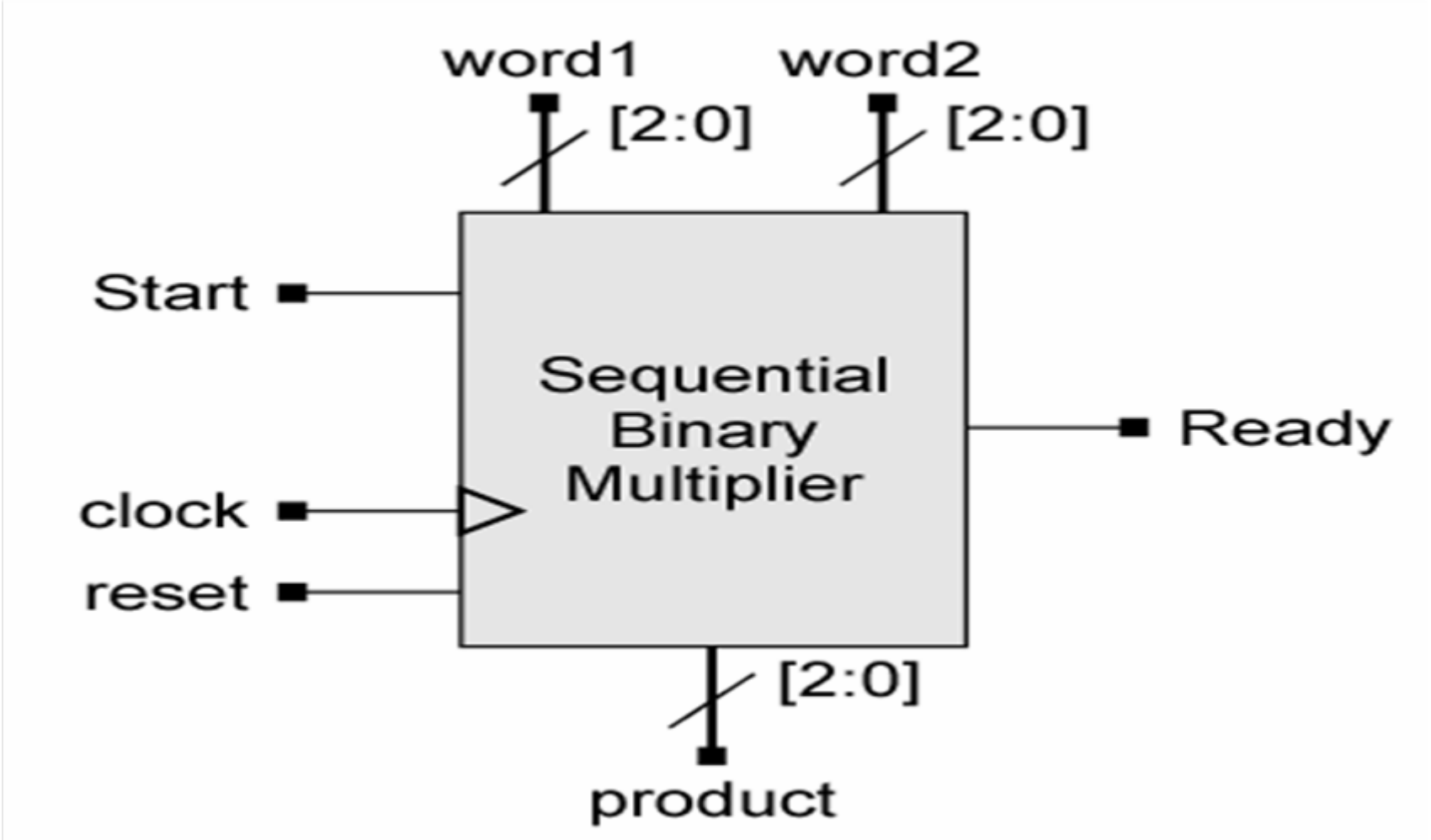
① Compact, need fewer adders

② Amenable to pipelining

③ The silicon area dose not grow significantly with the word length

④ The number of clock cycles grow in a linear manner, rather than exponentially

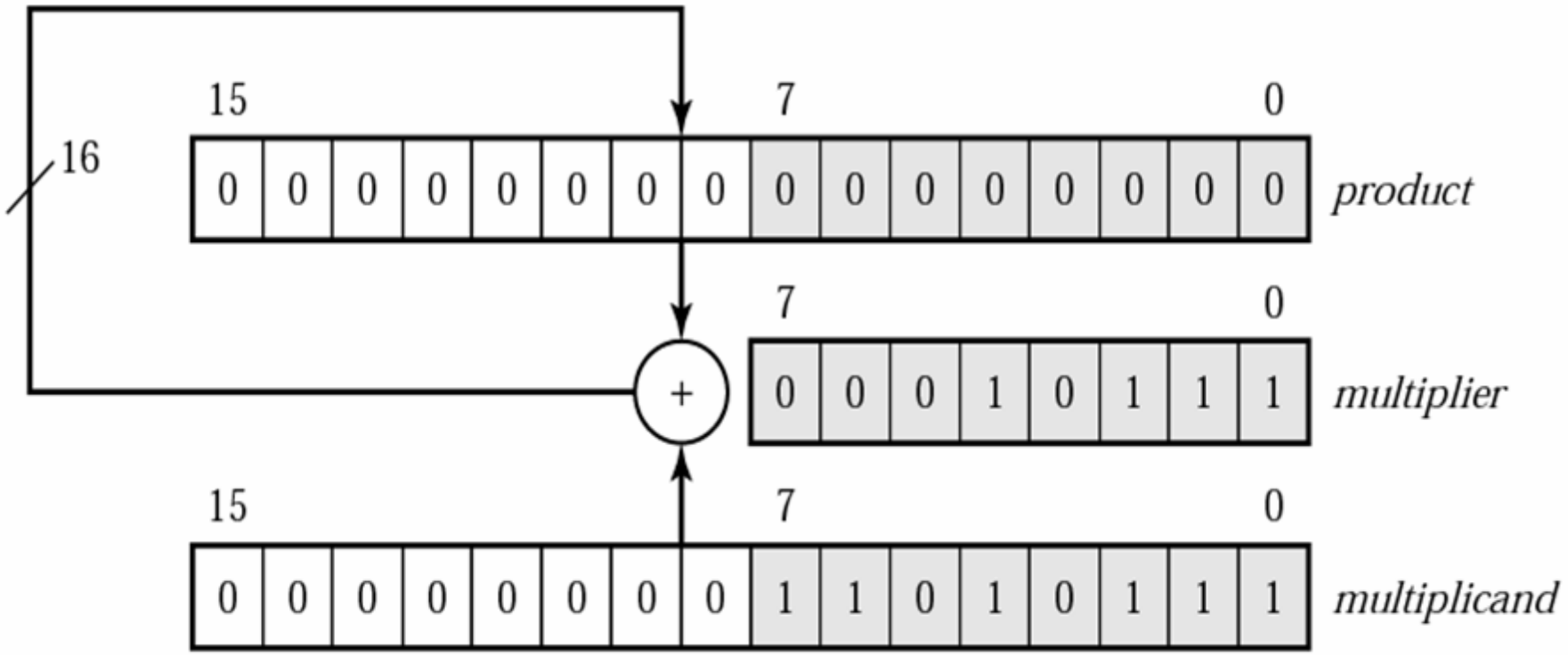
Fig 10-15 Interface signal of a sequential 8-bit binary multiplier



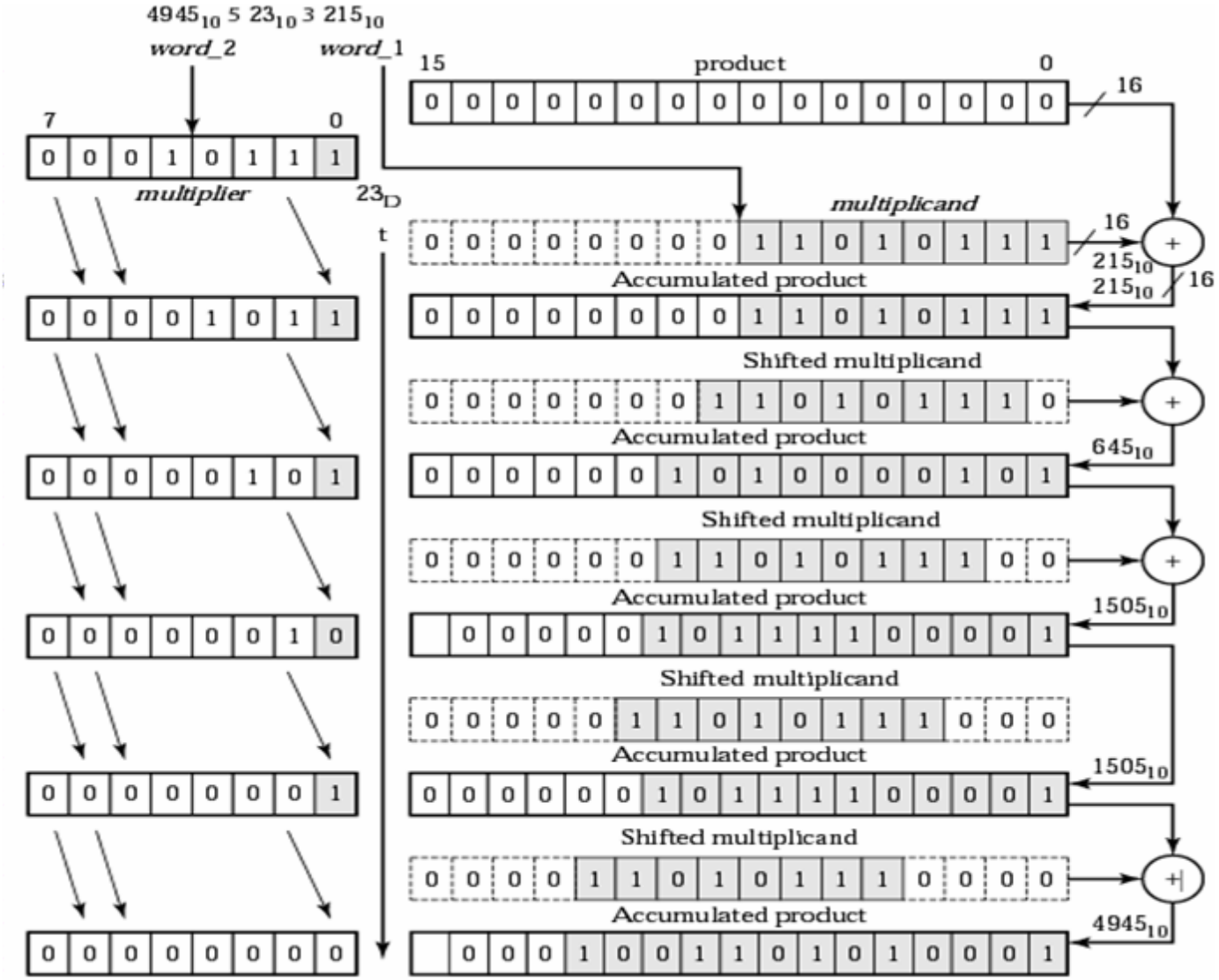
10.3.3 Sequential Multiplier Design: Hierarchical Decomposition

- The method for designing a sequential binary multiplier has two main steps:
  (1) choose a datapath architecture and
  (2) design a state machine to control the datapath
- For a given datapath architecture, the state machine must generate the appropriate sequence of control signals to direct the movement of data to produce the desired product

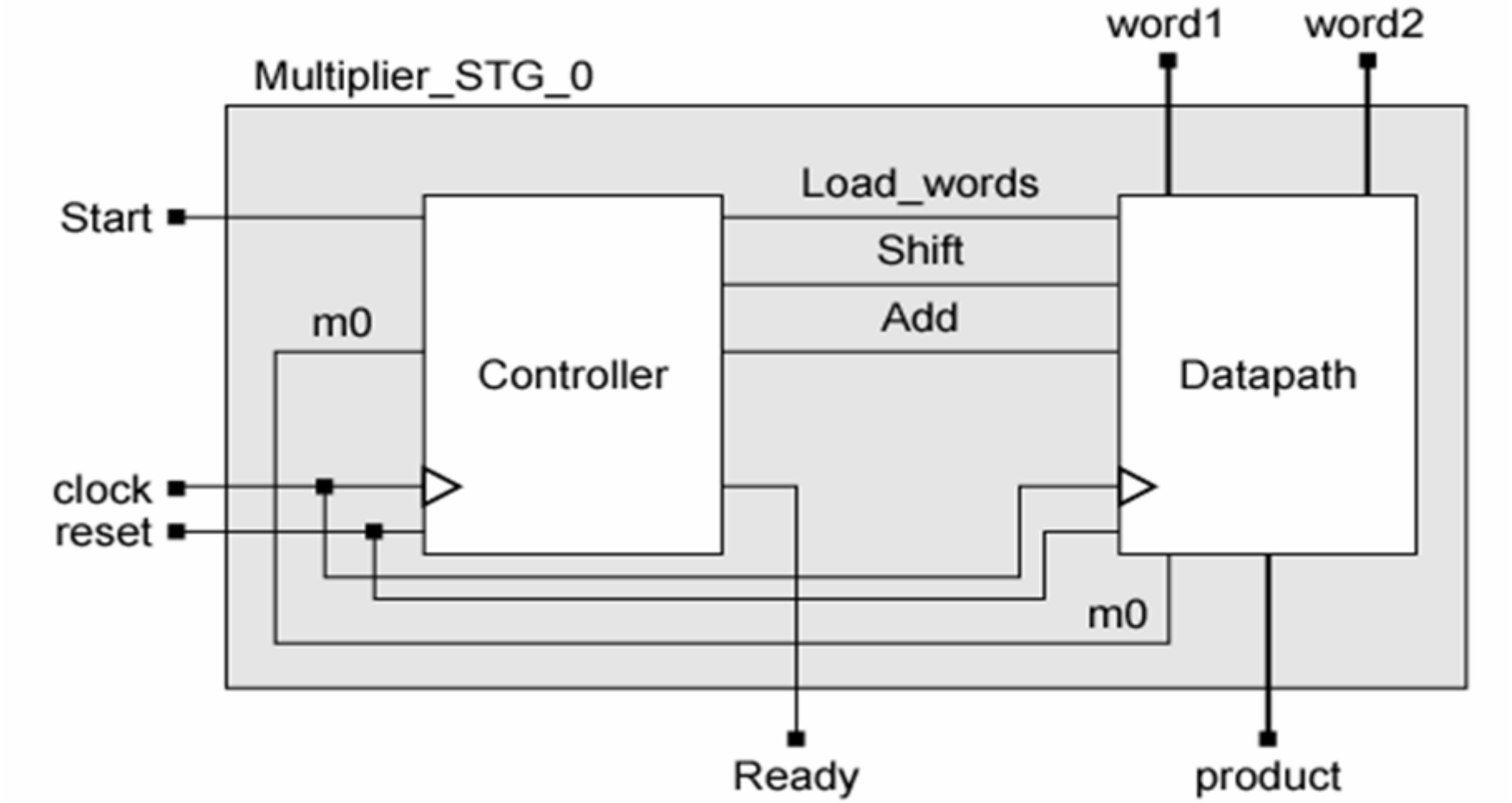
Datapath architecture of a sequential 8-bit binary multiplier



Register transfers in an 8-bit sequential binary multiplier



Structure units of a partitioned sequential binary multiplier：The datapath and controller of the Verilog structural



10.3.4 STG-Based Controller Design

- Use state-transition graphs (STGs) to specify the state transitions of the controller
- Two kinds of design methods for controllers:
- STG-Based Controller Design
- ASMD-Based Sequential Binary Multiplier

Fig 10-19 Alternative STGs for a 4-bit sequential binary multiplier：

two versions of a STG for the controller, with different behavior in S_8

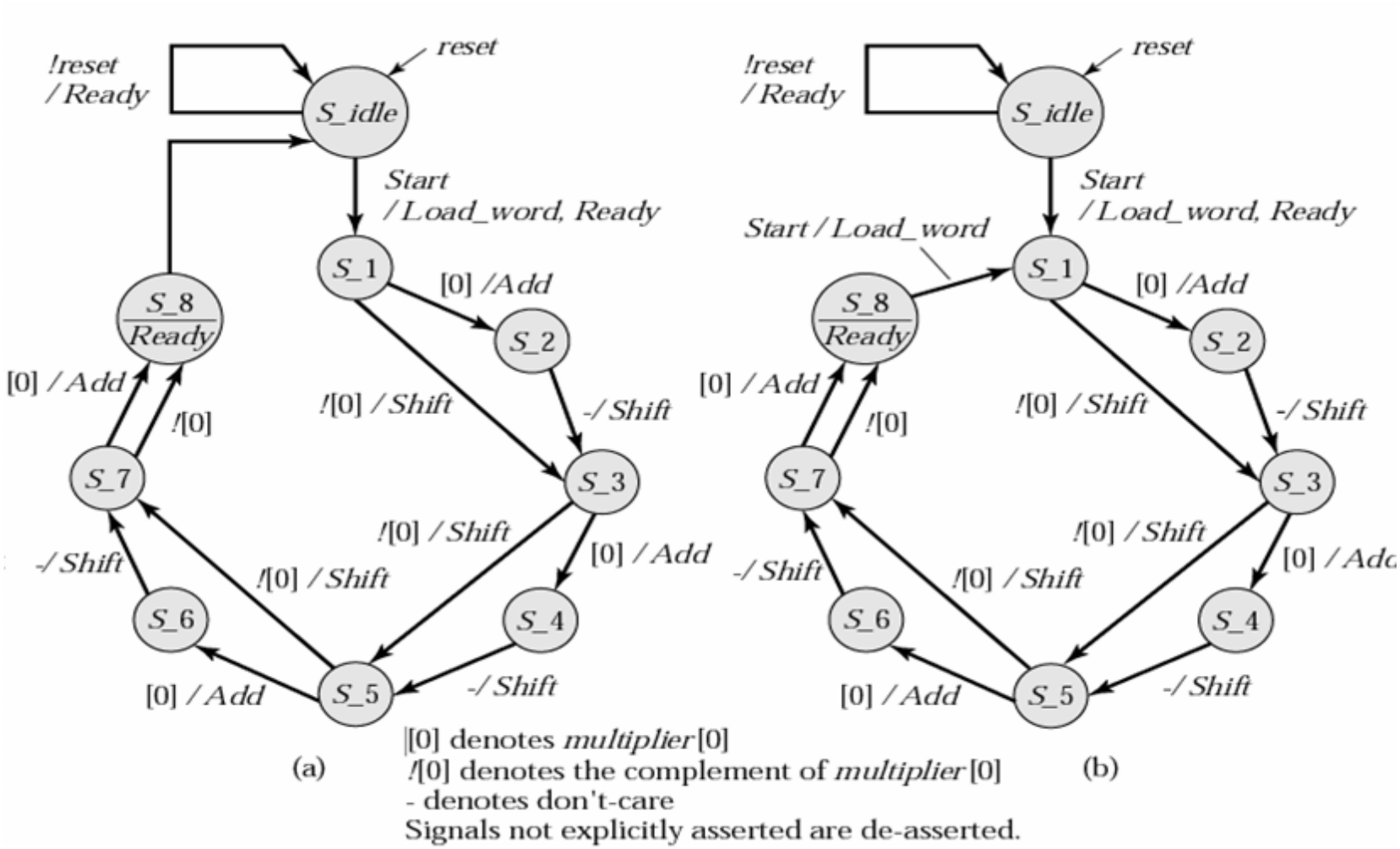
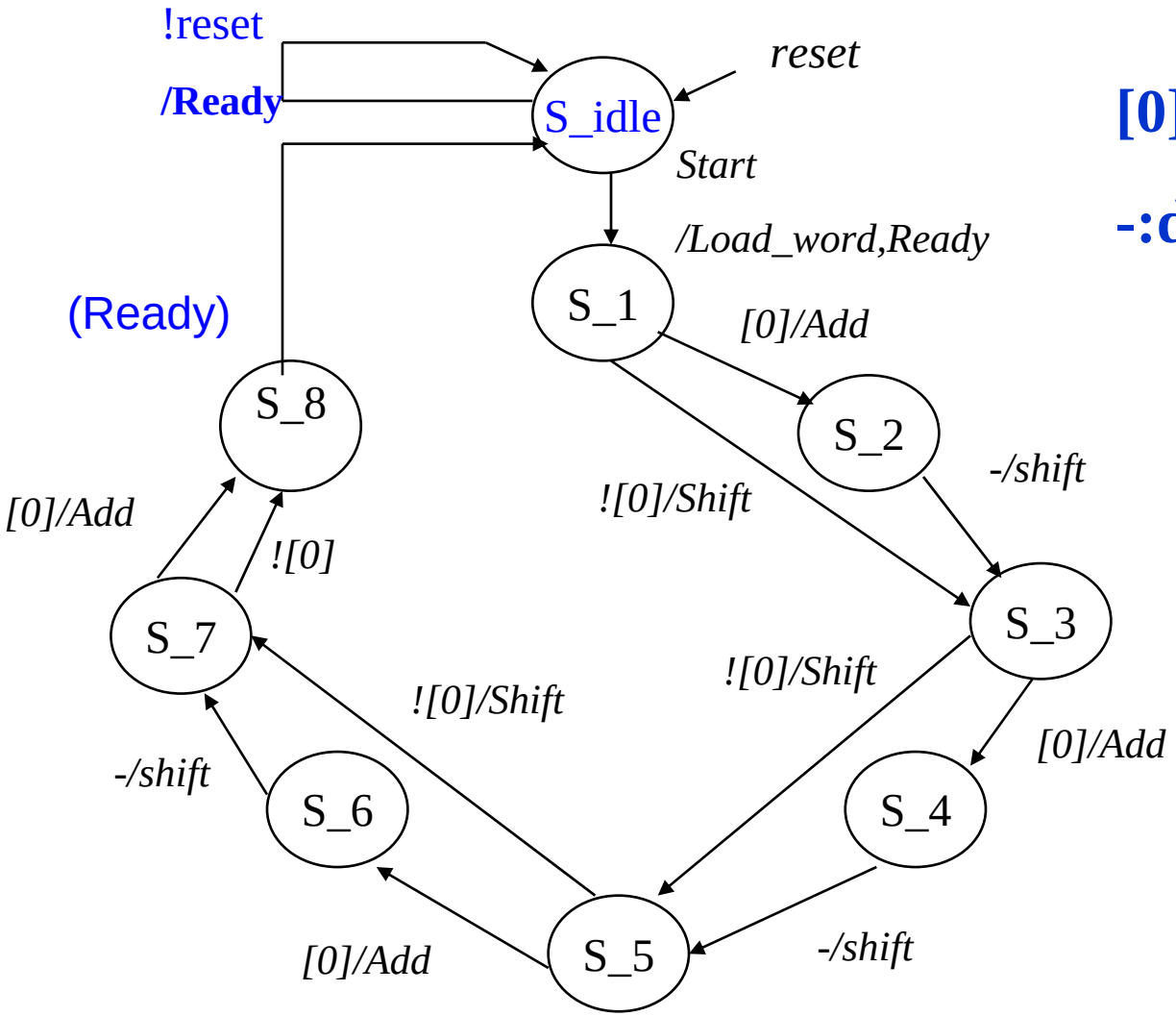


Fig 10-19(a) Machine return to S_idel upon completion



[0]: multiplier[0]

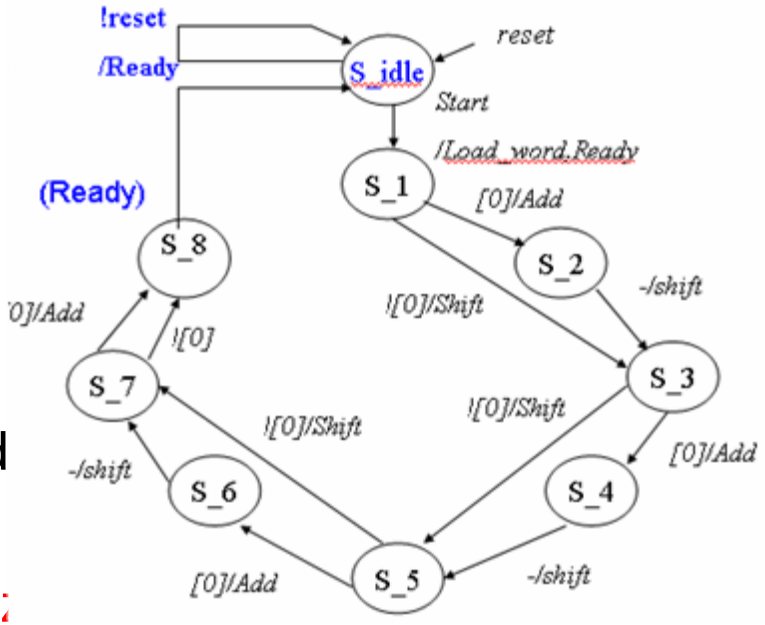
-:don't care

Example 10.1

- The Verilog behavioral description of Multiplier_STG_0 is given, along with a testbench that contains an exhaustive pattern generator for word1 and word2, and a comparator that samples product and checks, while Done is asserted, whether product matches the expected value

```
module Multiplier_STG_0 ( product, Ready, word1,
word2, Start, clock, reset);
   parameter                        L_word = 4;  // Datapath size
   output     [2*L_word-1:0]        product;
   output                           Ready;
   input      [L_word-1:0]          word1,word2;
   input                            Start,clock,reset;
   wire                             m0,load_words,Shift;

Datapath
     M1(product,m0,word1,word2,Load_words,Shift,Add,
          clock,reset);
Controller
   2(Load_words,Shift,Add,Ready,m0,Start,clock,reset);
endmodule
```




```
module Controller ( Load_words, Shift, Add
    Start, clock, reset );

Parameter           L_word=4; // Datapath si
Parameter           L_state=4;
output              Load_words,Shift,Add,Ready;
Input               m0,Start,clock,reset;
reg      [L_state-1:0]      state,next_state;
parameter           S_idle=0,S_1=1,S_2=2;
parameter           S_3=3,S_4=4,S_5=5,S_6=6;
parameter           S_7=7,S_8=8;
reg                 Load_words,Shift,Add;
wire
    Ready=((state==S_idle)&&!reset)||(state==S_8);
```

```
always @ (posedge clock or posedge reset)
     if (reset) state <= S_idle; else state <= next_state;
always@ (state or Start or m0)
begin
   Load_words=0;Shift=0;Add=0;
   case(state)
    S_idle:   if(Start)
                 begin Load_words=1; next_state=S_1;end
                   else  next_state=S_idle;
    S_1:       if(m0) //m0=multiplier[0]
                 begin Add=1; next_state=S_2;end
                  else
                     begin  Shift=1;next_state=S_3; end
    S_2:       begin   Shift=1;  next_state=S_3;  end
    S_3:       if(m0)   begin   Add=1;  next_state=S_4;  end
                  else   begin   Shift=1;  next_state=S_5;   end
```

```
        S_5:  if(m0)  //m0=multiplier[0]
                  begin    Add=1;  next_state=S_6;  end
                else
                  begin     Shift=1;  next_state=S_7;  end
        S_6:     begin   Shift=1;   next_state=S_7;   end
        S_7:   if(m0)
                  begin    Add=1;  next_state=S_8;   end
                 else
                  begin    Shift=1;  next_state=S_8;  end
        S_8:     if(Start)
                    begin Load_words=1;
                       next_state=S_1;  end
                   else     next_state=S_8;
        default:  next_state=S_idle;
        endcase
    end
endmodule
```

```
module Datapath (  product, m0,
   word1,word2, Load_words, Shift, Add, clock,
   reset);
parameter      L_word=4;
output    [2*L_word-1:0]       product;
output                         m0;
input  [L_word-1:0]     word1,word2;
Input    Load_words,Shift,Add,clock,reset;
reg [2*L_word-1:0]   product,multiplicand;
reg [L_word-1:0]      multiplier;
wire       m0=multiplier[0];
```

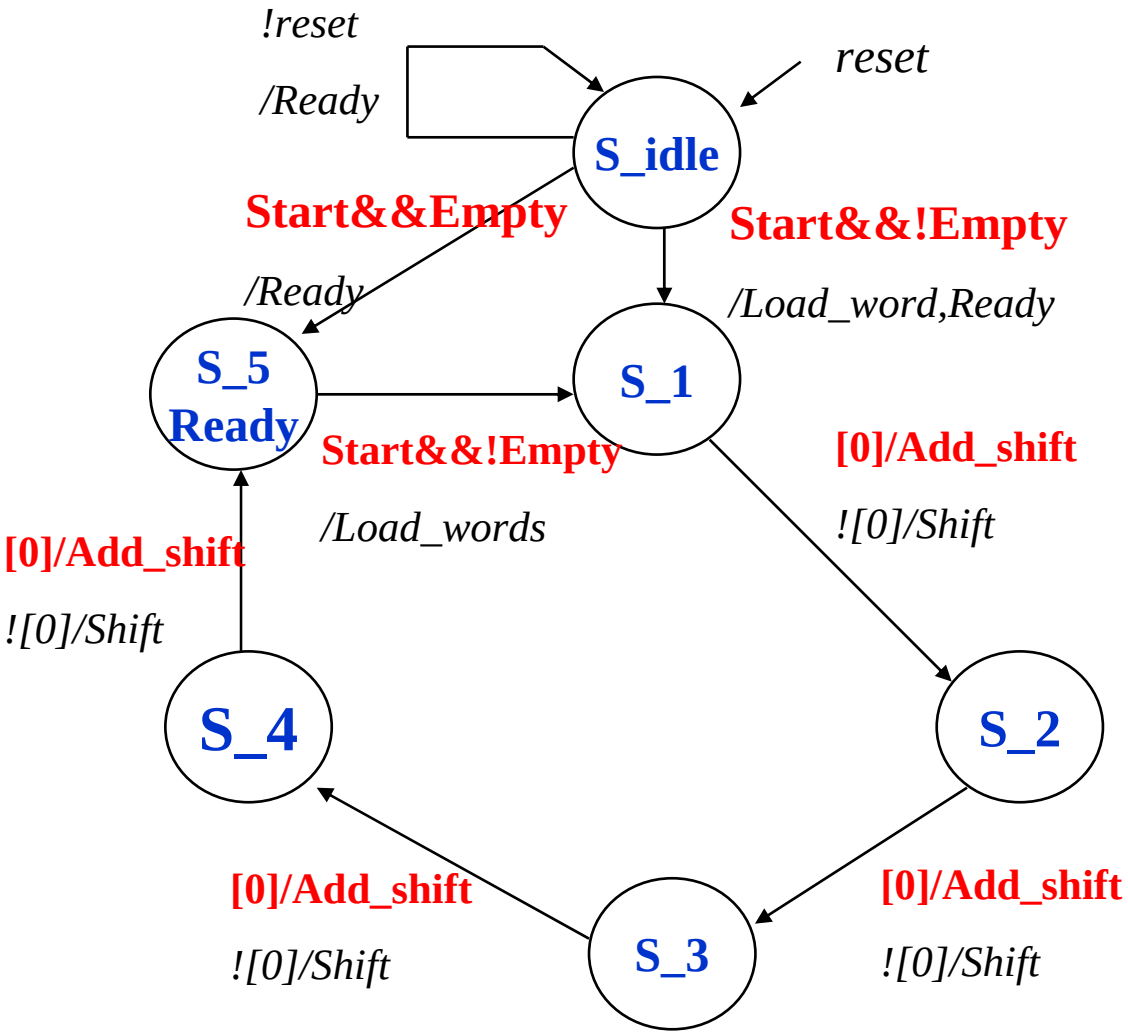
```
always@(posedge clock or posedge reset)
begin
        if(reset)
          begin  multiplier<=0; multiplicand<=0;
            product<=0; end
        else if (Load_words)
          begin  multiplicand<={4'b0,word1};
            multiplier<=word2; product<=0;end
        else if (Shift)
          begin multiplier<=multiplier>>1;
            multiplicand<=multiplicand<<1; end
         else if(Add)
               product<=product+multiplicand;
end
endmodule
```

10.3.5 Efficient STG-Based Sequential Binary Multiplier

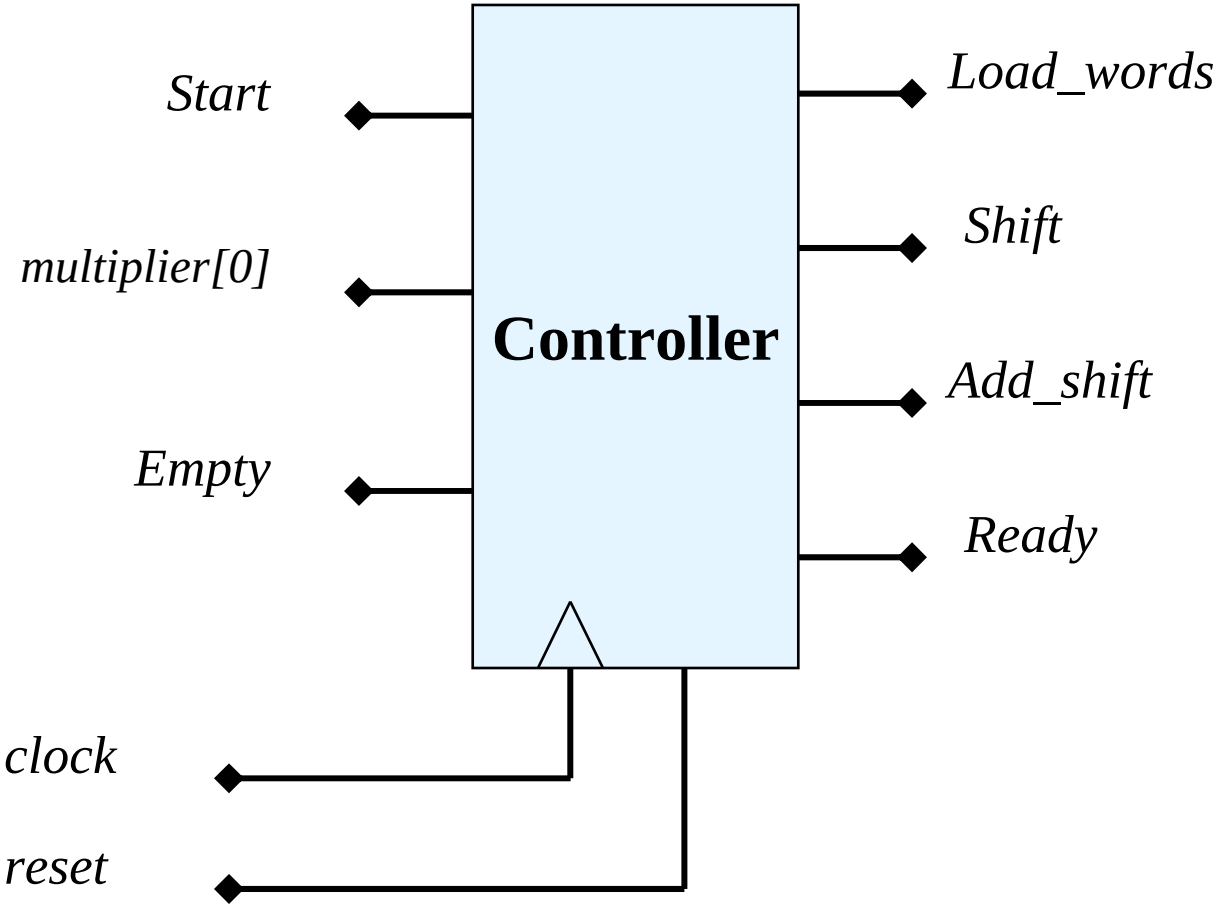
- The sequential multiplier, Multiplier_STG_0, is inefficient,

  because it executes the add and shift operations in separate clock cycles
- If the architecture is modified to direct the output of the adder to the appropriate bits of the product register, the operations can execute in the same cycle

Figure 10-22 STG for the controller of an efficient sequential multiplier



STG and block diagram symbol for the controller of an efficient sequential multiplier



10.3.6 ASMD-Based Sequential Binary Multiplier

- STGs are convenient tools for designs that have only a few states, but are cumbersome when the number of states is large
- In STG-based design ,for longer words, additional code must be inserted for each new bit to handle additional state transitions.
- **ASMD charts facilitate scalable, portable, and re-usable designs.**

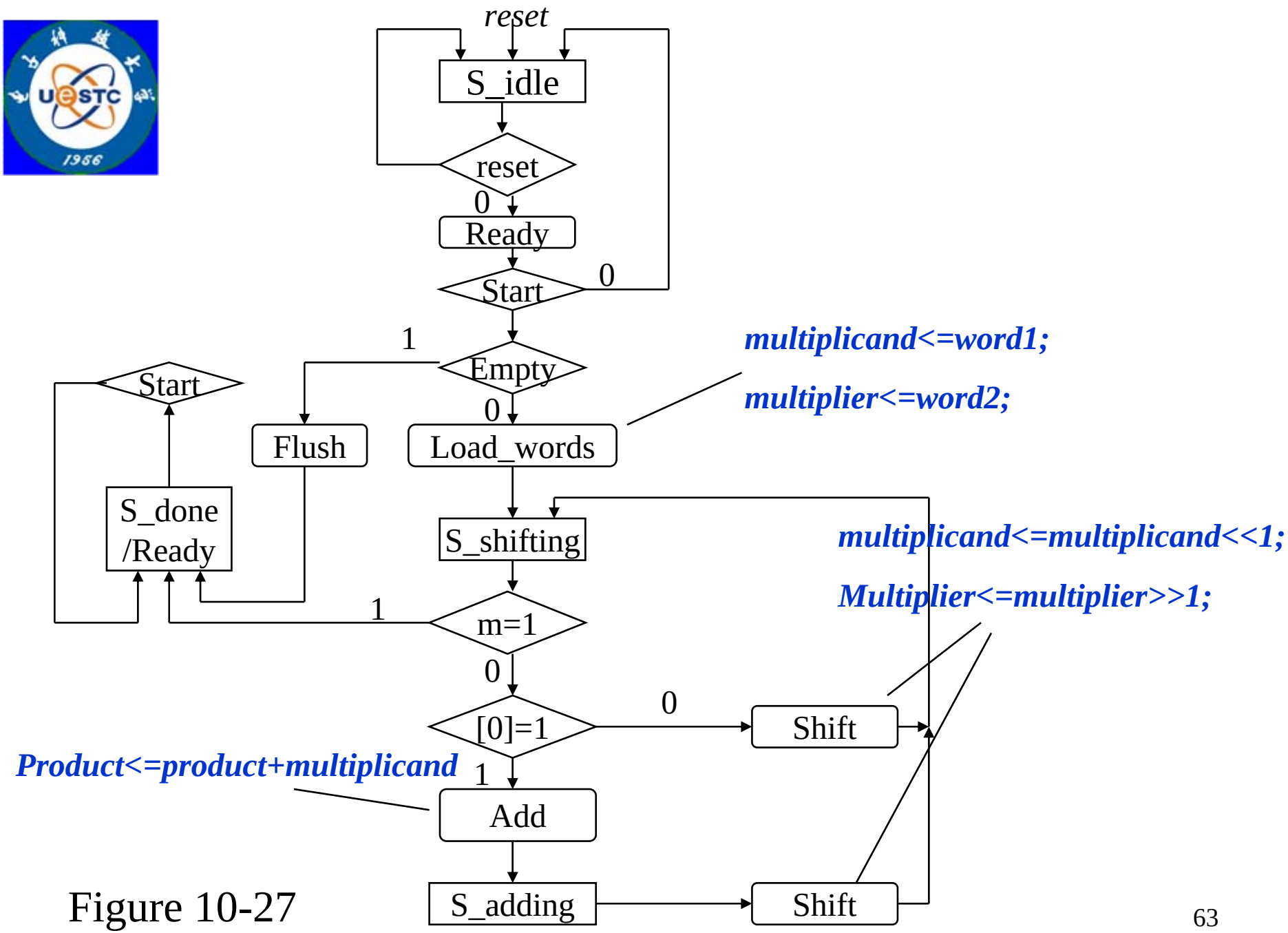
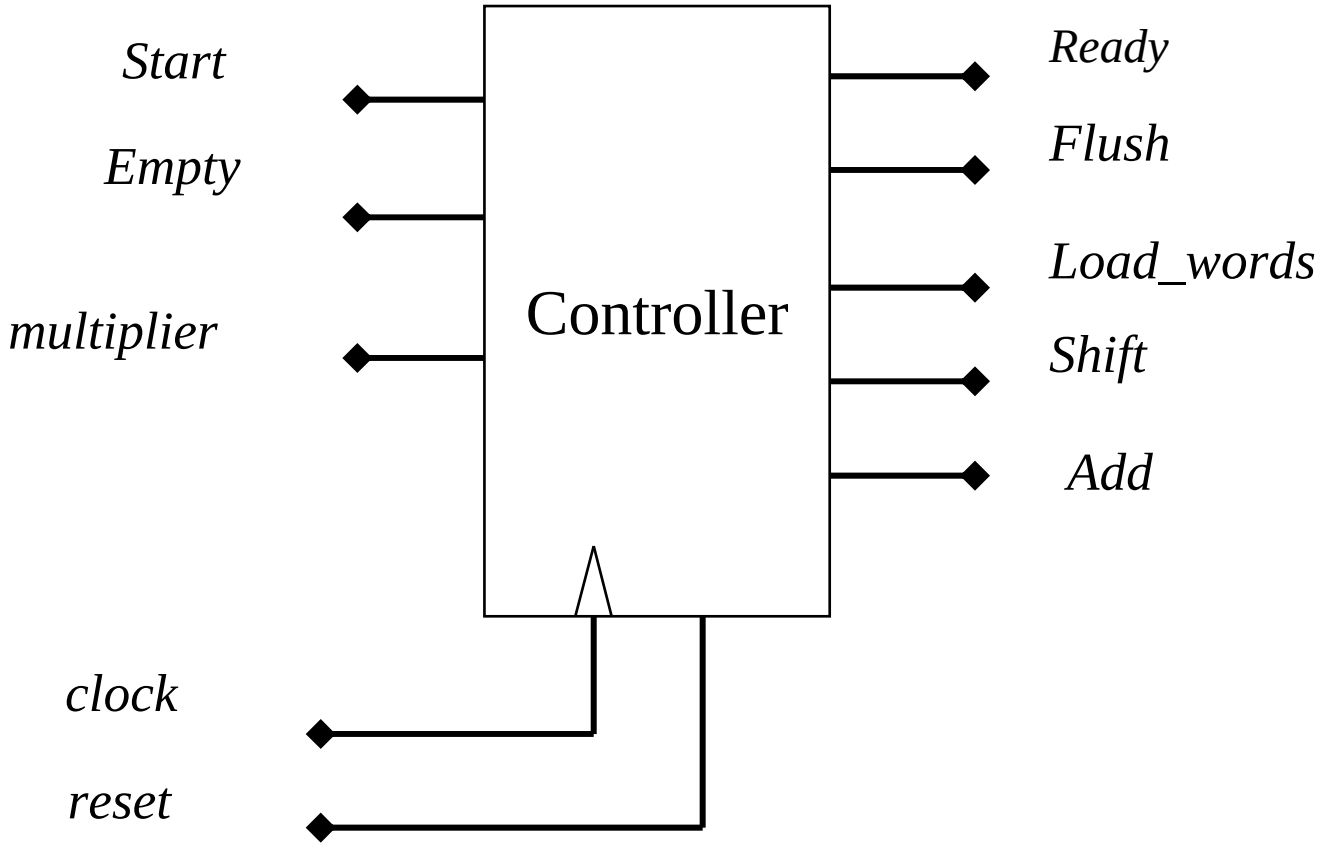

reset
S_idle
reset
0
Ready
Start
0
1
Empty
0
Start
Flush
Load_words
multiplicand<=word1;
multiplier<=word2;
S_done
/Ready
S_shifting
multiplicand<=multiplicand<<1;
Multiplier<=multiplier>>1;
1
m=1
0
[0]=1
0
Shift
Product<=product+multiplicand
1
Add
S_adding
Shift

Figure 10-27

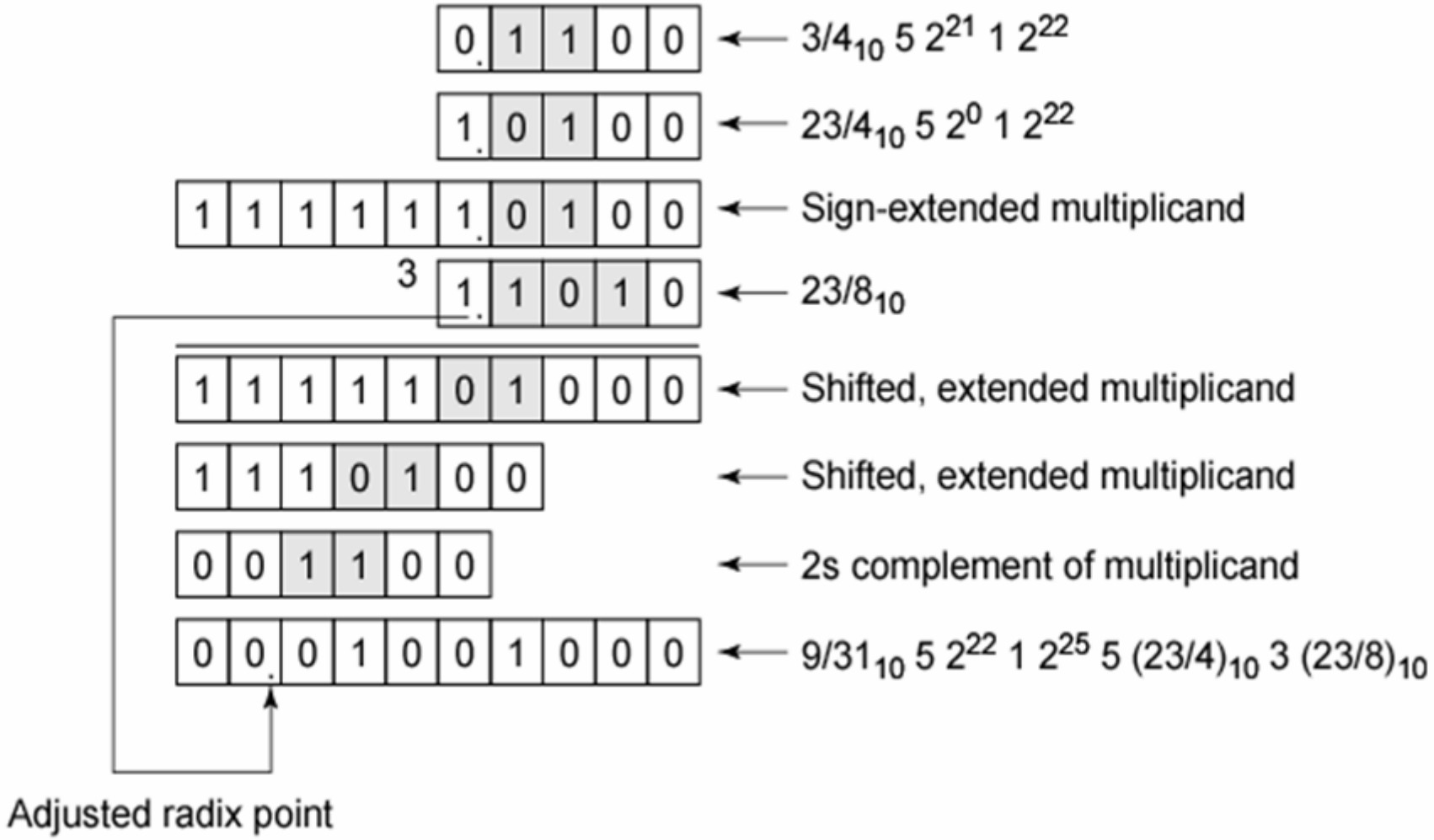
Start is decoded with higher priority than *Empty*



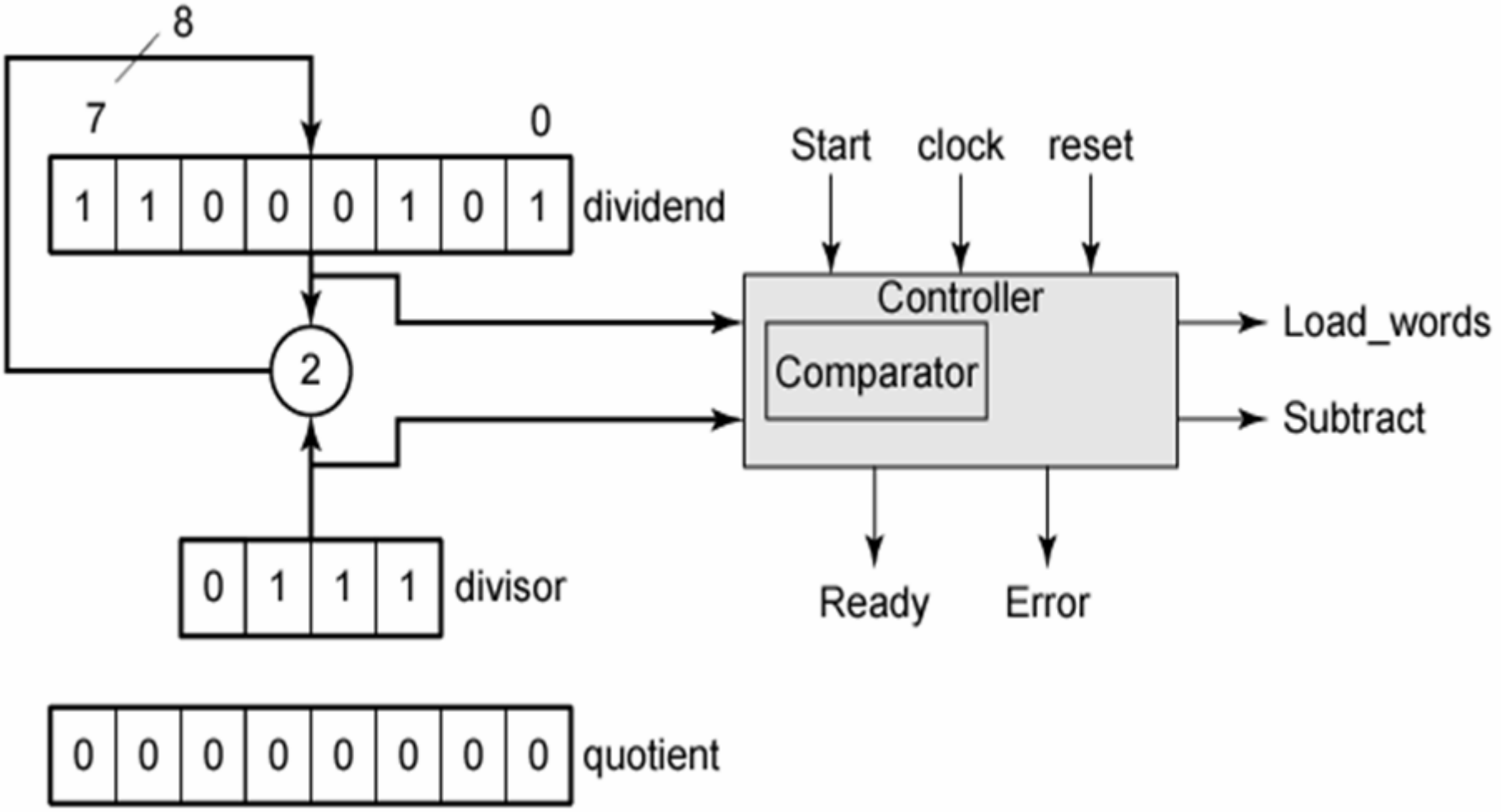
10.6 Functional Units for Division

- Sequential multipliers use an add-and-shift algorithm to form the product of two words
- We will consider various architectures for sequential dividers that use a subtract-and-shift algorithm to form a quotient of two numbers.

A sequential algorithm for dividing two unsigned binary numbers



Architecture of a simple binary divider unit



P10-1.The 4-bit multiplier shown in Figure 10-13 can be modified to exploit a 4-bit carry look-ahead adder instead of a ripple-carry adder. Compare the performance and area of the two models