Design of Digital Systems II

Combinational Logic Design Practices (3)

Moslem Amiri, Václav Přenosil

Embedded Systems Laboratory
Faculty of Informatics, Masaryk University
Brno, Czech Republic

amiri@mail.muni.cz
prenosil@fi.muni.cz

Fall, 2014

Exclusive-OR and Exclusive-NOR Gates

- An XOR gate is a 2-input gate whose output is 1 if its inputs are different

$$X \oplus Y = X' \cdot Y + X \cdot Y'$$

- An XNOR gate is a 2-input gate whose output is 1 if its inputs are the same

Table 1: Truth table for XOR and XNOR functions.

X	Y	$X \oplus Y$	$(X \oplus Y)'$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Exclusive-OR and Exclusive-NOR Gates

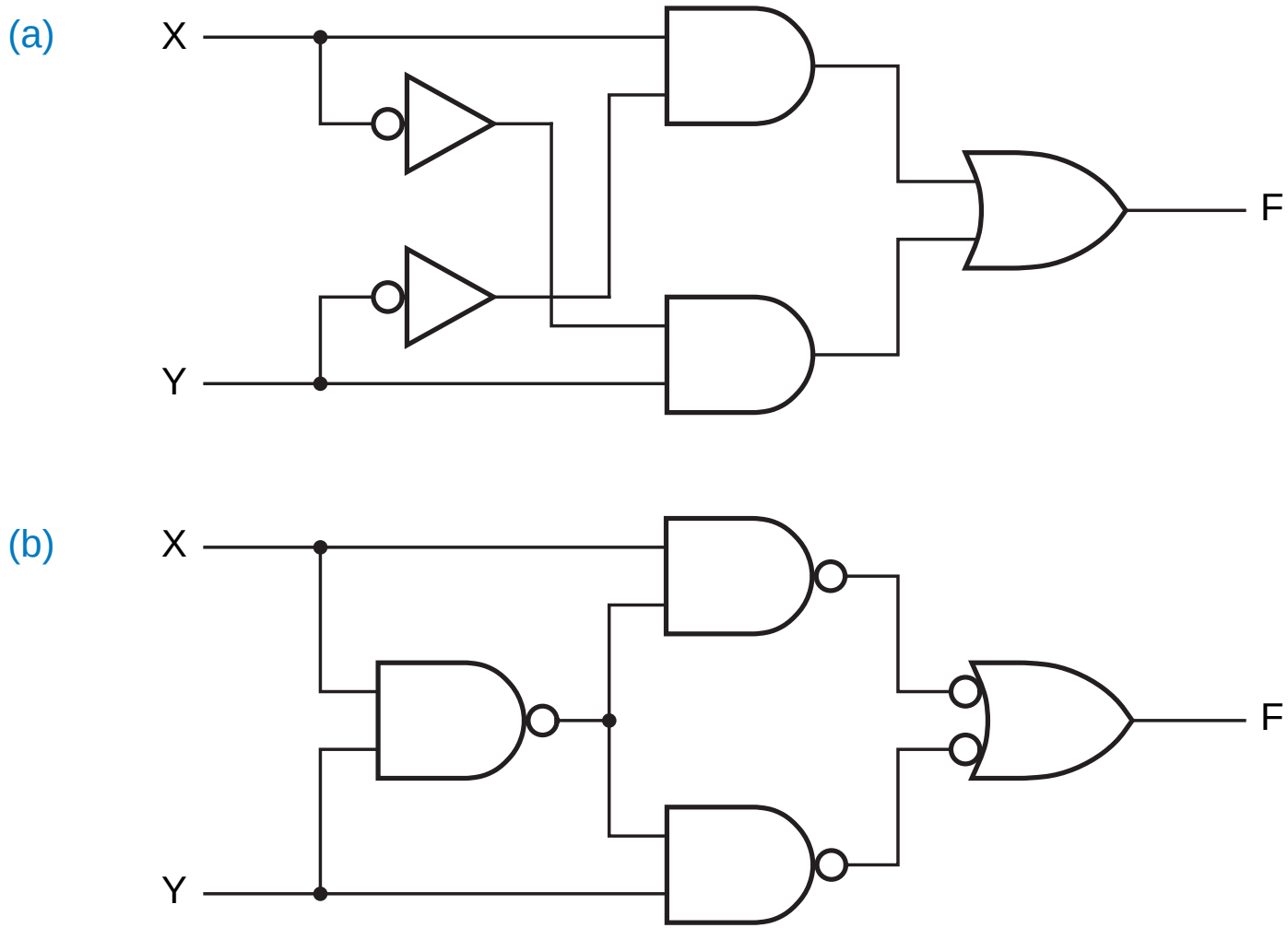


Figure 1: Multigate designs for the 2-input XOR function: (a) AND-OR; (b) three-level NAND.

Exclusive-OR and Exclusive-NOR Gates

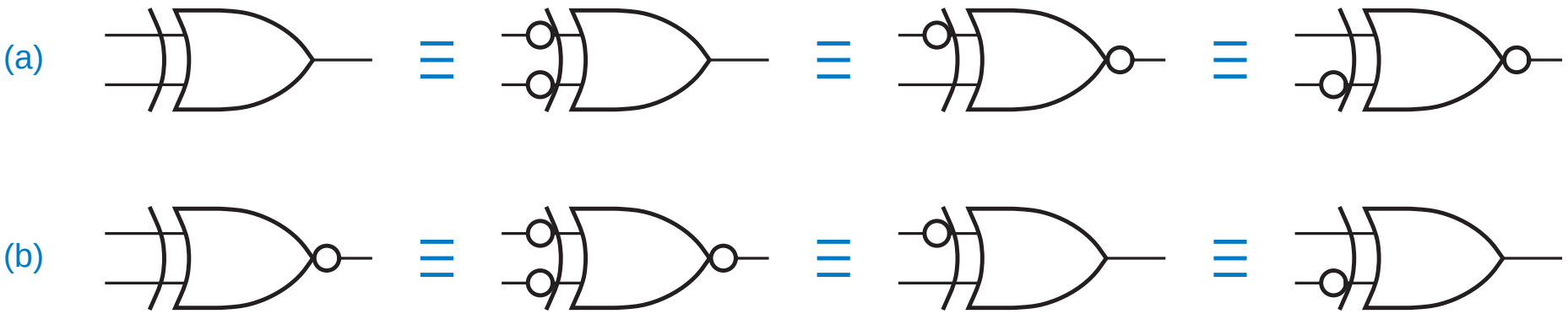


Figure 2: Equivalent symbols for (a) XOR gates; (b) XNOR gates.

- As seen in Fig. 2, any two signals (inputs or output) of an XOR or XNOR gate may be complemented without changing resulting logic function

Parity Circuits

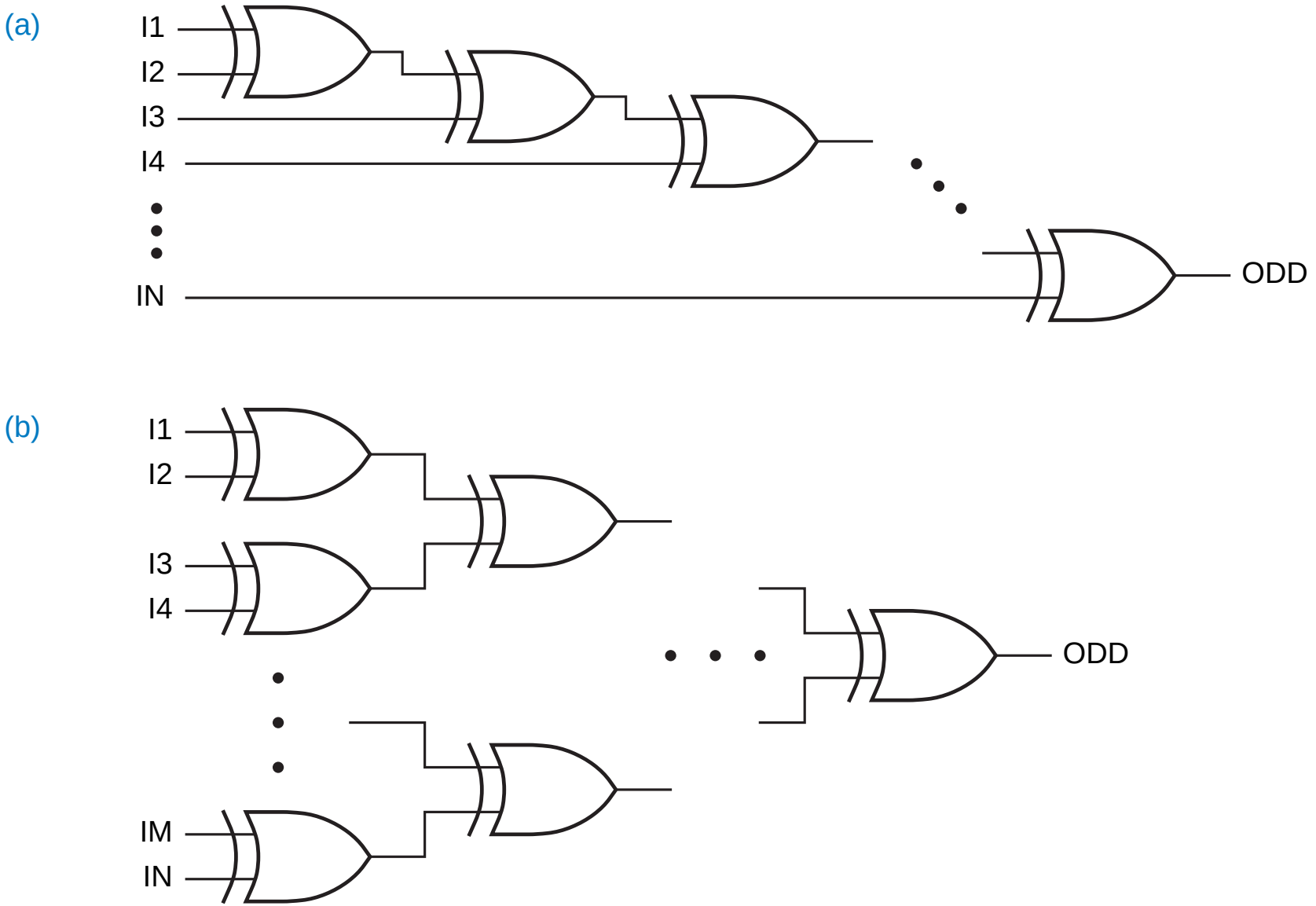


Figure 3: Cascading XOR gates: (a) daisy-chain connection; (b) tree structure.

Parity Circuits

- Fig. 3
  - (a) is an **odd-parity circuit**
    - Its output is 1 if an odd number of its inputs are 1
  - (b) is also an odd-parity circuit, but it is faster
  - If output of either circuit is inverted, we get an **even-parity circuit**

Parity Circuits: The 74x280 9-Bit Parity Generator

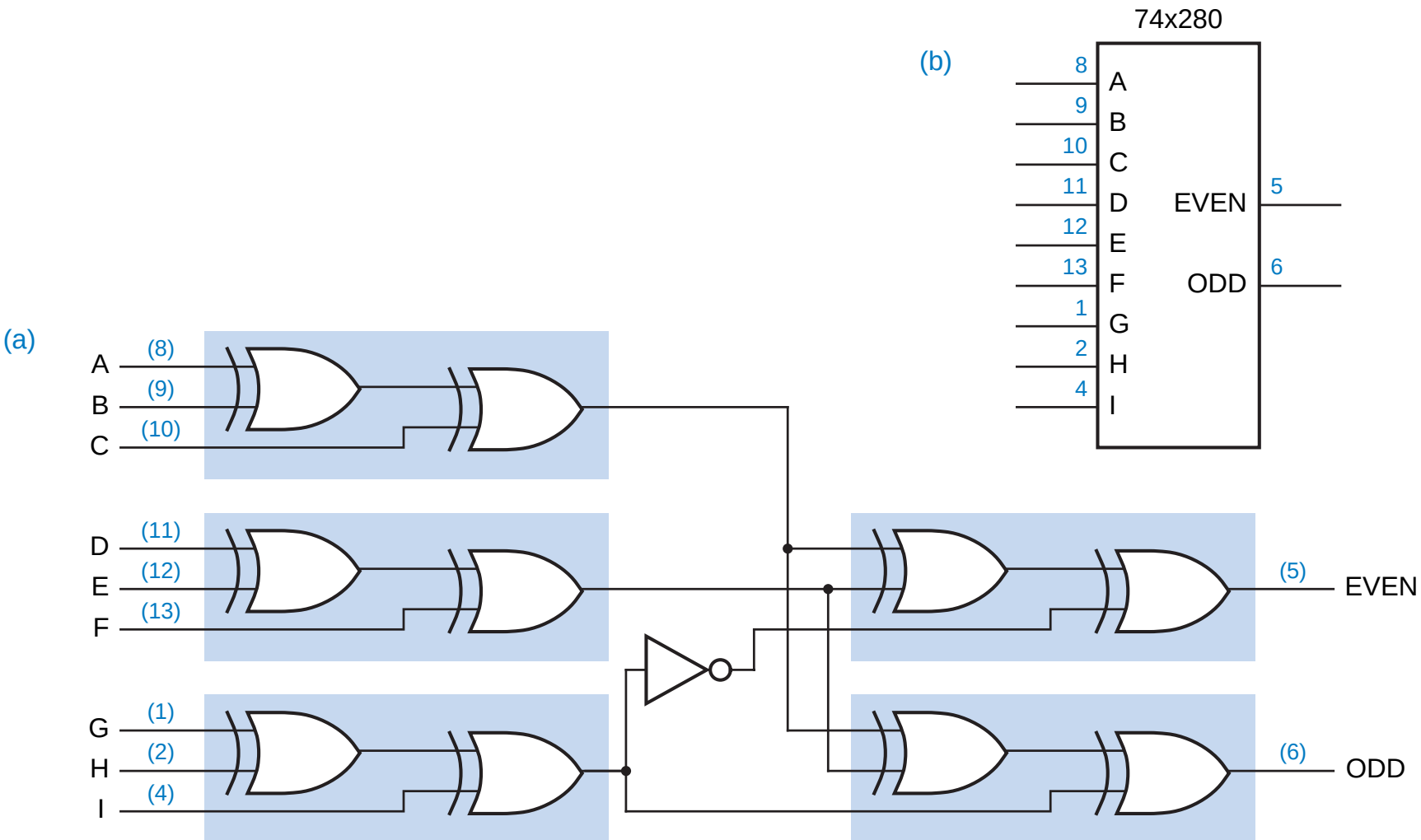


Figure 4: The 74x280 9-bit odd/even parity generator: (a) logic diagram, including pin numbers for a standard 16-pin dual in-line package; (b) traditional logic symbol.

Parity Circuits: Parity-Checking Applications

- A parity bit is used in error-detecting codes to detect errors in transmission and storage of data
  - In an even-parity code, parity bit is chosen so that total number of 1 bits in a code word is even
- Parity circuits like 74x280 are used both to generate correct value of parity bit when a code word is stored or transmitted, and to check parity bit when a code word is retrieved or received

Parity Circuits: Parity-Checking Applications

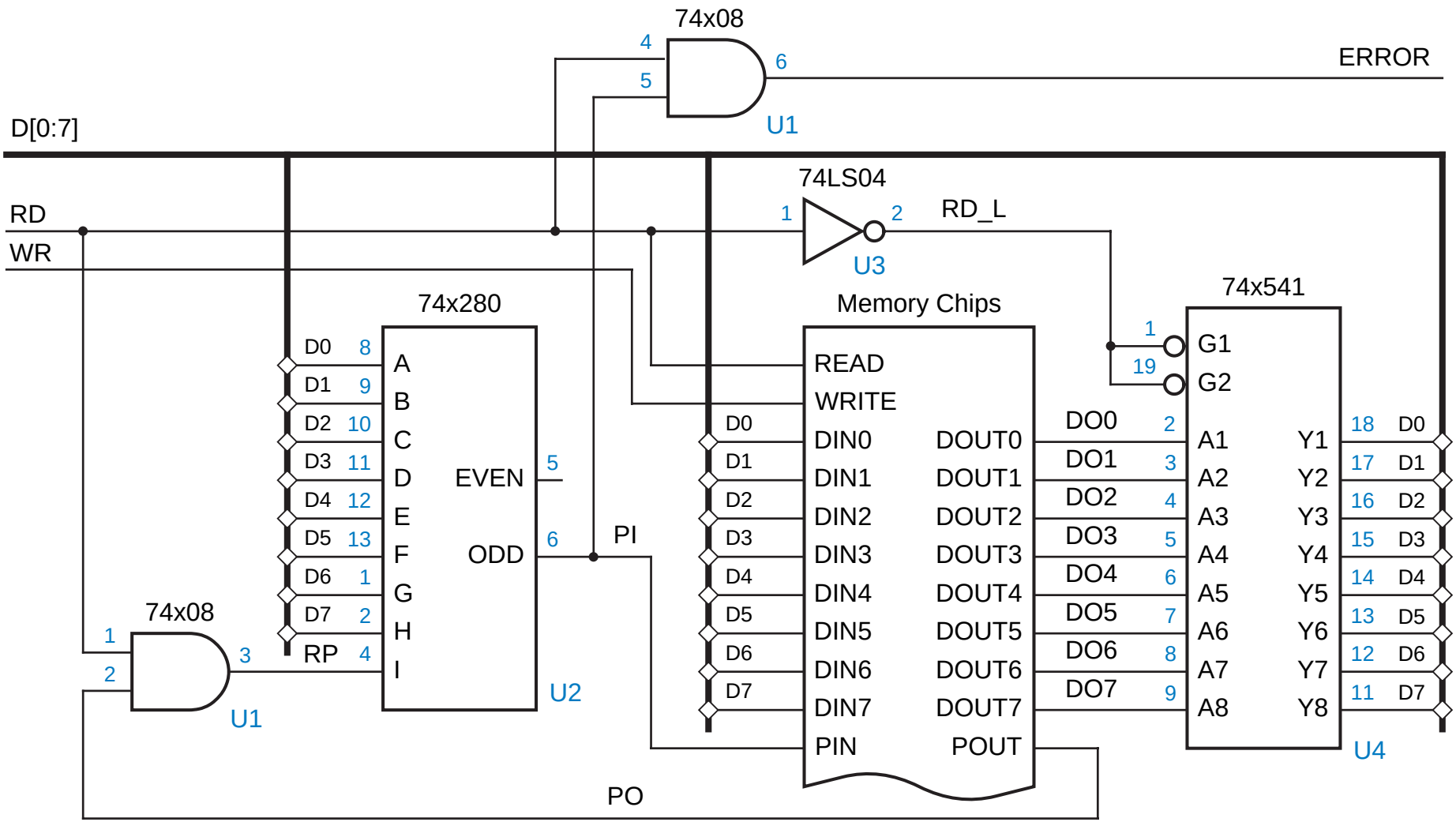


Figure 5: Parity generation and checking for an 8-bit-wide memory.

Parity Circuits: Parity-Checking Applications

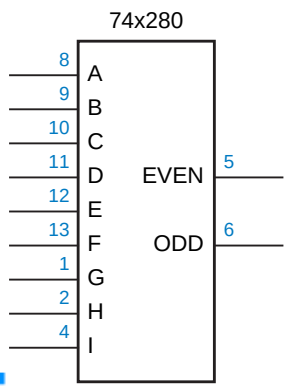
- In Fig. 5, to store a byte into memory
  - Specify an address
  - Place byte on D[0–7]
  - Generate its parity bit on PIN
  - Assert WR
  - '280's ODD output is connected to PIN, so that total number of 1s stored is even
- In Fig. 5, to retrieve a byte
  - Specify an address
  - Assert RD
  - A 74x541 drives byte onto D bus, and '280 checks its parity
  - If parity of 9-bit word is odd during a read, ERROR signal is asserted

Table 2: Dataflow-style Verilog module for a 3-input XOR device.

```
module Vrxor3(A, B, C, Y);
  input A, B, C;
  output Y;

  assign Y = A ^ B ^ C;
endmodule
```

Table 3: Behavioral Verilog program for a 9-input parity checker.

```
module Vrparity9(I, EVEN, ODD);
  input [1:9] I;
  output EVEN, ODD;
  reg p, EVEN, ODD;
  integer j;

  always @ (I) begin
    p = 1'b0;
    for (j =1; j <= 9; j = j+1)
      if (I[j]) p = ~p;
    else p = p;
    ODD = p;
    EVEN = ~p;
  end
endmodule
```



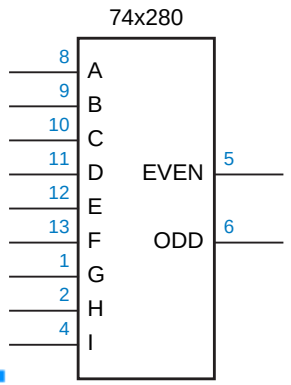
Exclusive-OR Gates and Parity Circuits in Verilog

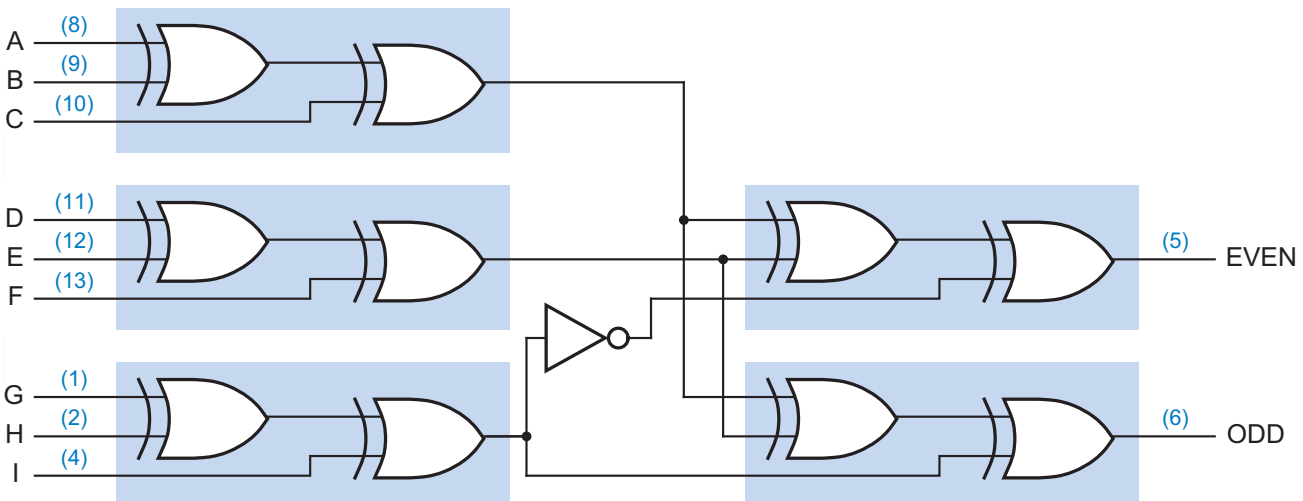
- ASIC and FPGA libraries contain two- and three-input XOR and XNOR functions as primitives
  - In CMOS ASICs, these primitives are realized very efficiently at transistor level using transmission gates
  - Fast and compact XOR trees can be built using these primitives
  - Typical Verilog synthesis tools are not smart enough to create an efficient tree structure from a behavioral program like Tab. 3
  - Instead, we can use a structural program to get exactly what we want
    - Tab. 4

Exclusive-OR Gates and Parity Circuits in Verilog

Table 4: Structural Verilog program for a 74x280-like parity checker.

```
module Vrparity9s(I, EVEN, ODD);
  input [1:9] I;
  output EVEN, ODD;
  wire Y1, Y2, Y3, Y3N;

  Vrxor3 U1 (I[1], I[2], I[3], Y1);
  Vrxor3 U2 (I[4], I[5], I[6], Y2);
  Vrxor3 U3 (I[7], I[8], I[9], Y3);
  assign Y3N = ~Y3;
  Vrxor3 U4 (Y1, Y2, Y3, ODD);
  Vrxor3 U5 (Y1, Y2, Y3N, EVEN);
endmodule
```





Comparators

- A **comparator** is a circuit that compares two binary words and indicates whether they are equal
- **Magnitude comparators** interpret their input words as signed or unsigned numbers and also indicate an arithmetic relationship (greater or less than) between words

Comparators: Comparator Structure

- XOR or XNOR gates may be viewed as 1-bit comparators

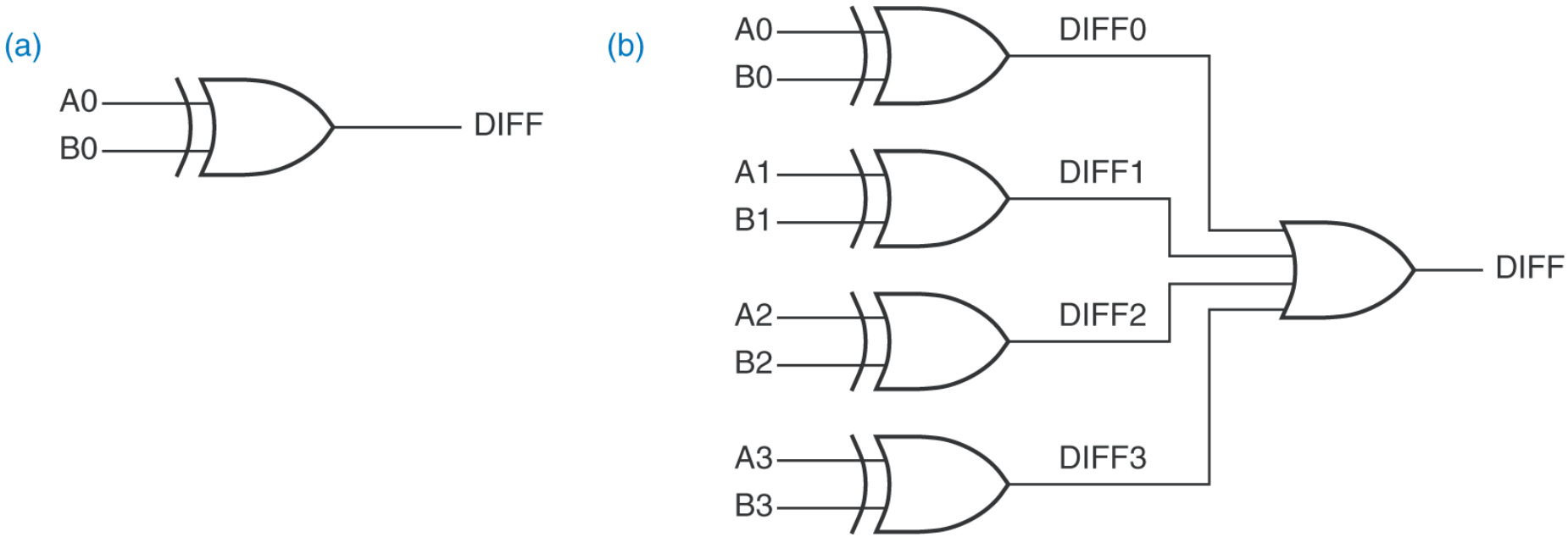


Figure 6: Comparators using XOR gates: (a) 1-bit comparator; (b) 4-bit comparator.

- We can build an n-bit comparator using n XOR gates and an n-input OR gate
- Wider OR functions can be obtained by cascading individual gates
  - A faster circuit is obtained by arranging gates in a tree-like structure
  - Using NORs and NANDs in place of ORs makes circuit even faster

Comparators: Comparator Structure

- Comparators can also be built using XNOR gates
  - A 2-input XNOR gate produces a 1 output if its two inputs are equal
  - A multibit comparator can be constructed using one XNOR gate per bit, and ANDing all of their outputs together
  - Output of AND function is 1 if all of individual bits are pairwise equal
- n-bit comparators in this subsection are called **parallel comparators**
  - They look at each pair of input bits simultaneously and deliver 1-bit comparison results in parallel to an n-input OR or AND function

Comparators: Iterative Circuits

- An iterative circuit contains n identical modules, each of which has both primary inputs and outputs and cascading inputs and outputs
  - Left-most cascading inputs are called boundary inputs and are connected to fixed logic values
  - Right-most cascading outputs are called boundary outputs and usually provide important information

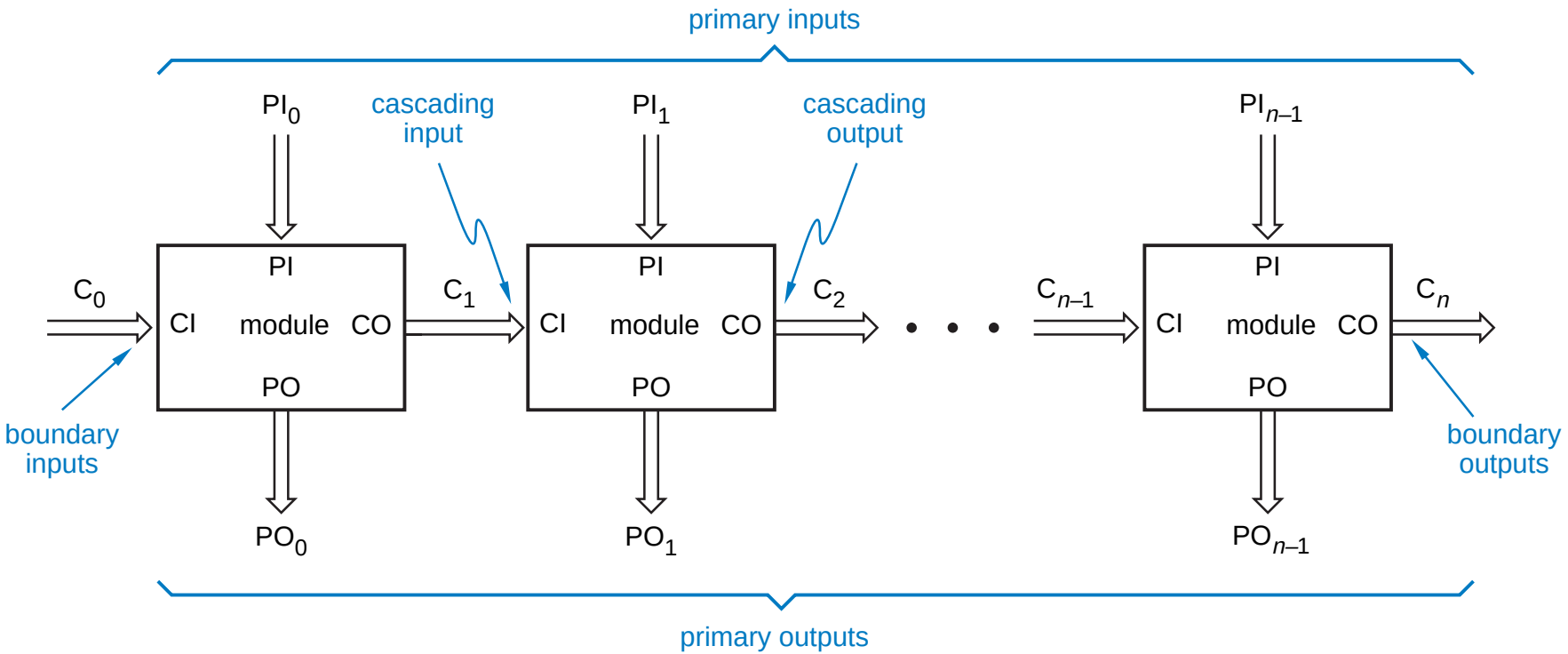


Figure 7: General structure of an iterative combinational circuit.

Comparators: Iterative Circuits

- Iterative circuits are suited to problems that can be solved by an iterative algorithm
  1. Set C_0 to its initial value and set i to 0
  2. Use C_i and PI_i to determine values of PO_i and C_{i+1}
  3. Increment i
  4. If $i < n$, go to step 2

Comparators: An Iterative Comparator Circuit

- To compare two n-bit values X and Y
  1. Set EQ_0 to 1 and set i to 0
  2. If EQ_i is 1 and X_i and Y_i are equal, set EQ_{i+1} to 1, else set EQ_{i+1} to 0
  3. Increment i
  4. If $i < n$, go to step 2

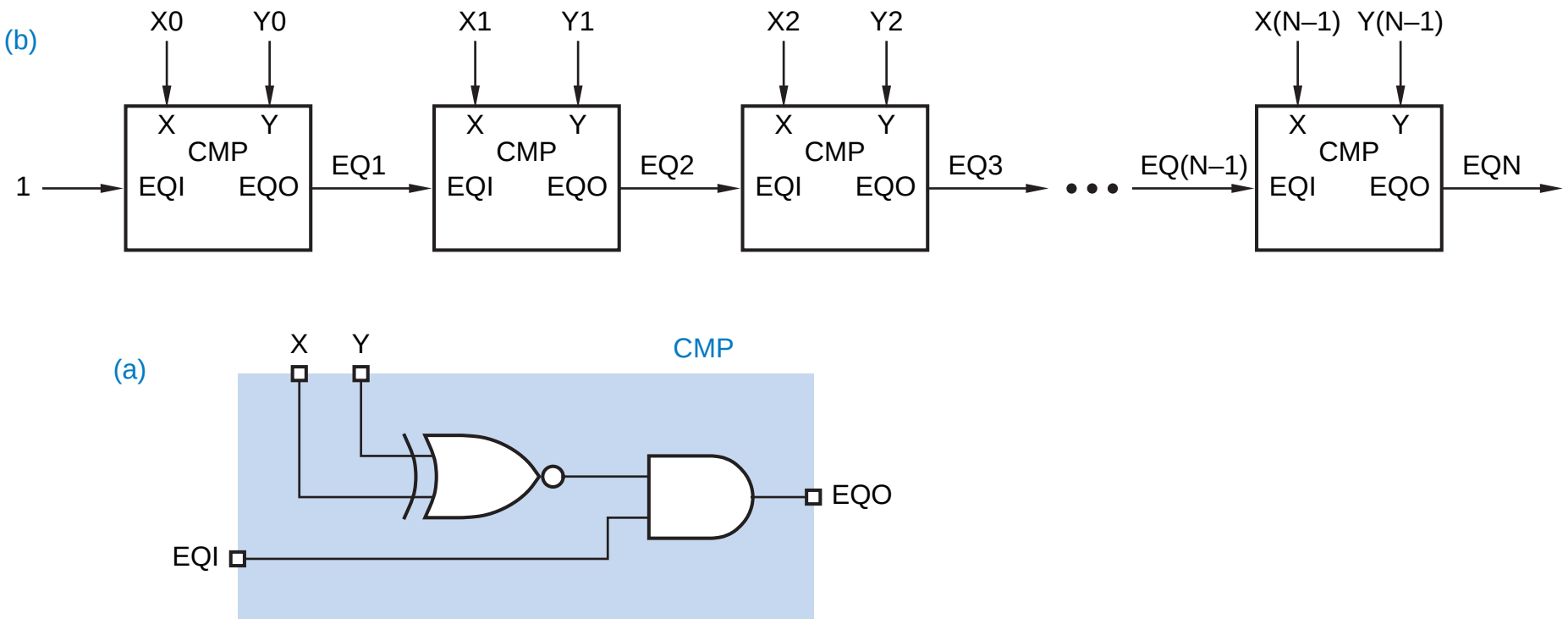


Figure 8: An iterative comparator circuit: (a) module for one bit; (b) complete circuit.

Comparators: An Iterative Comparator Circuit

- Parallel comparators are preferred over iterative ones
  - Iterative comparators are very slow
    - Cascading signals need time to "ripple" from leftmost to rightmost module
  - Iterative circuits that process more than one bit at a time (using modules like 74x85, discussed next) are much more likely to be used in practical designs

Comparators: Standard MSI Magnitude Comparators

- 74x85 is a 4-bit comparator which provides a greater-than output (AGTBOUT) and a less-than output (ALTBOUT) as well as an equal output (AEQBOUT)
  - '85 also has cascading inputs (AGTBIN, ALTBIN, AEQBIN) for combining multiple '85s to create comparators for more than four bits

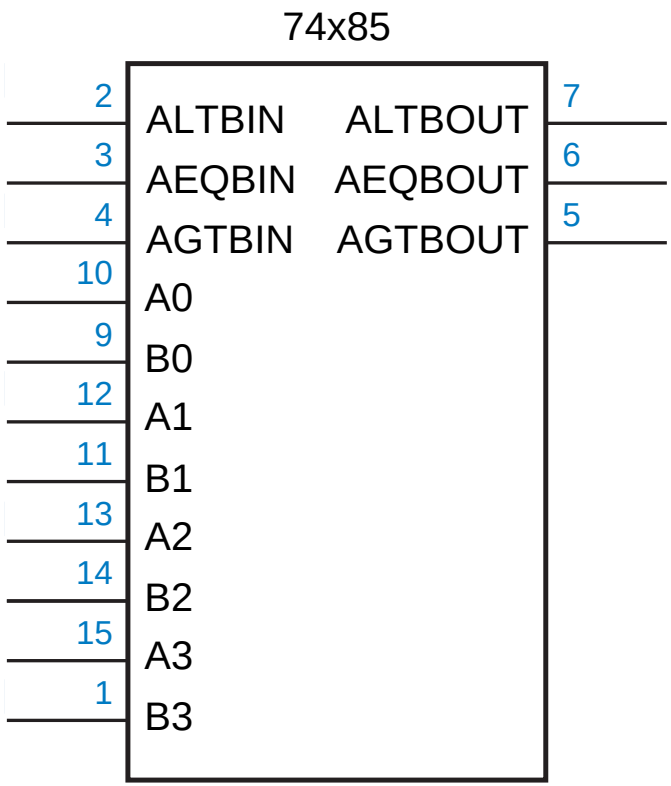


Figure 9: Traditional logic symbol for the 74x85 4-bit comparator.

Comparators: Standard MSI Magnitude Comparators

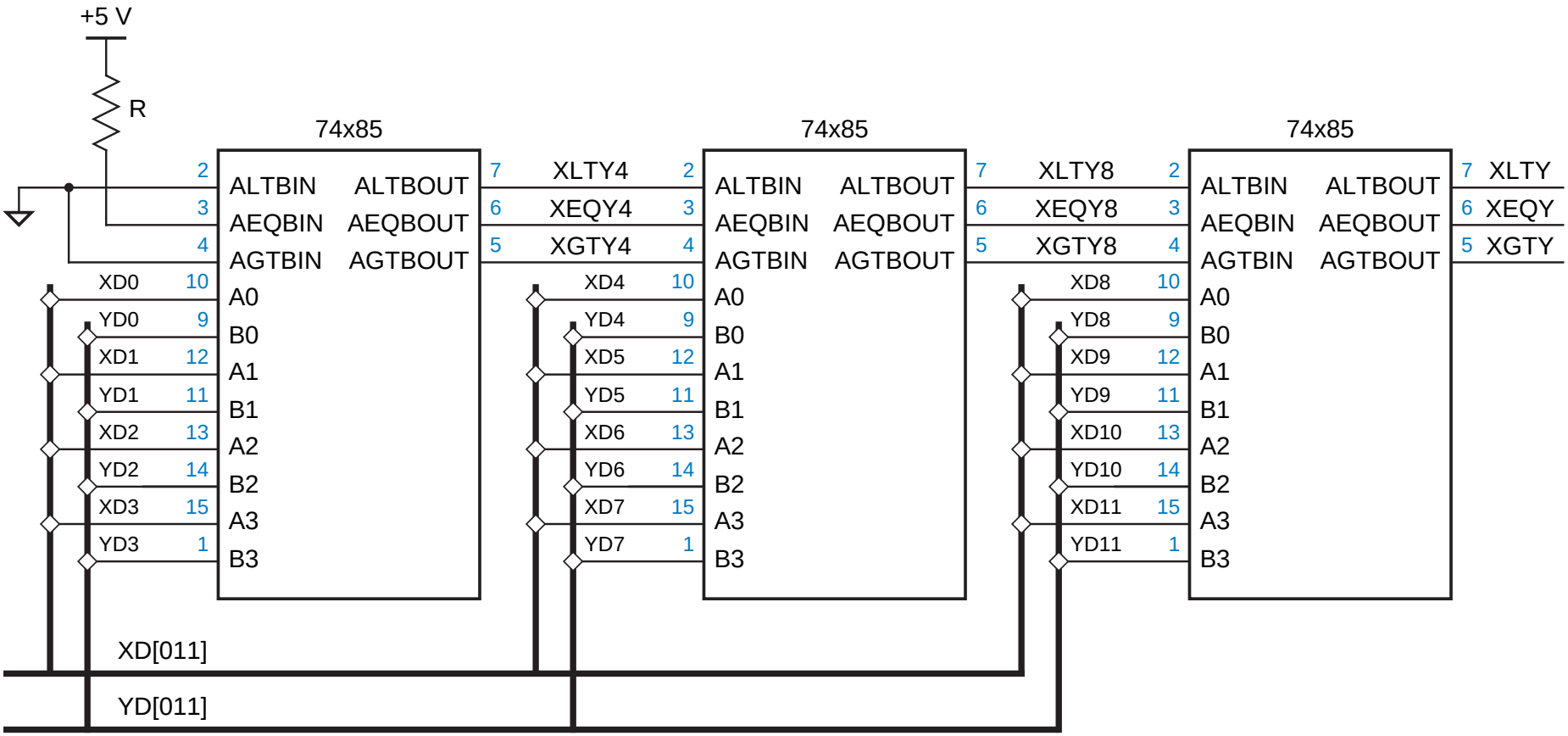


Figure 10: A 12-bit comparator using 74x85s.

Comparators: Standard MSI Magnitude Comparators

- Cascading inputs are defined so outputs of an '85 that compares less-significant bits are connected to inputs of an '85 that compares more-significant bits
- For each '85

$$AGTBOUT = (A > B) + (A = B) \cdot AGTBIN$$
$$AEQBOUT = (A = B) \cdot AEQBIN$$
$$ALTBOUT = (A < B) + (A = B) \cdot ALTBIN$$

Arithmetic comparisons can be expressed using normal logic expressions, e.g.,

$$\begin{aligned}(A > B) = {} & A3 \cdot B3' + \\ & (A3 \oplus B3)' \cdot A2 \cdot B2' + \\ & (A3 \oplus B3)' \cdot (A2 \oplus B2)' \cdot A1 \cdot B1' + \\ & (A3 \oplus B3)' \cdot (A2 \oplus B2)' \cdot (A1 \oplus B1)' \cdot A0 \cdot B0'\end{aligned}$$

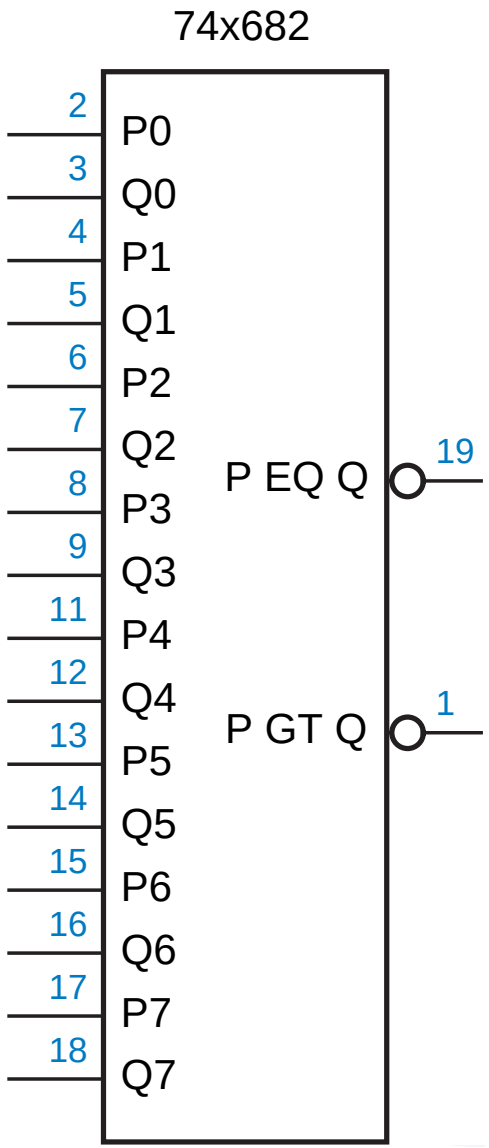


Figure 11: Traditional logic symbol for the 74x682 8-bit comparator.

Comparators: Standard MSI Magnitude Comparators

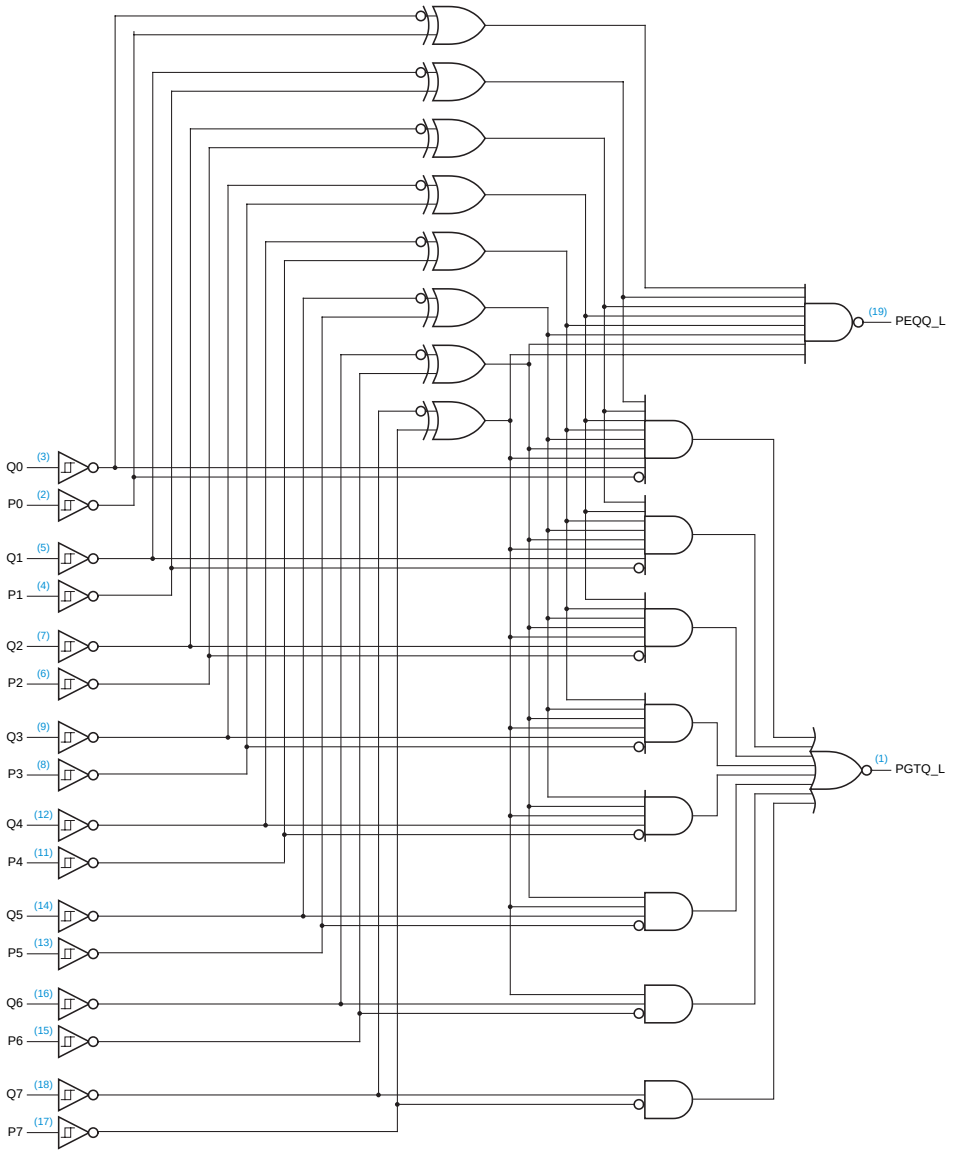


Figure 12: Logic diagram for the 74x682 8-bit comparator, including pin numbers for a standard 20-pin dual in-line package.

Comparators: Standard MSI Magnitude Comparators

- In Fig. 12
  - Top half of circuit checks two 8-bit input words for equality
    - PEQQ_L output is asserted if all eight input-bit pairs are equal
  - Bottom half of circuit compares input words arithmetically
    - PGTQ_L is asserted if P[7–0] $>$ Q[7–0]
- 74x682 does not have cascading inputs and a "less than" output
  - However, any desired condition can be formulated as a function of PEQQ_L and PGTQ_L outputs

Comparators: Standard MSI Magnitude Comparators

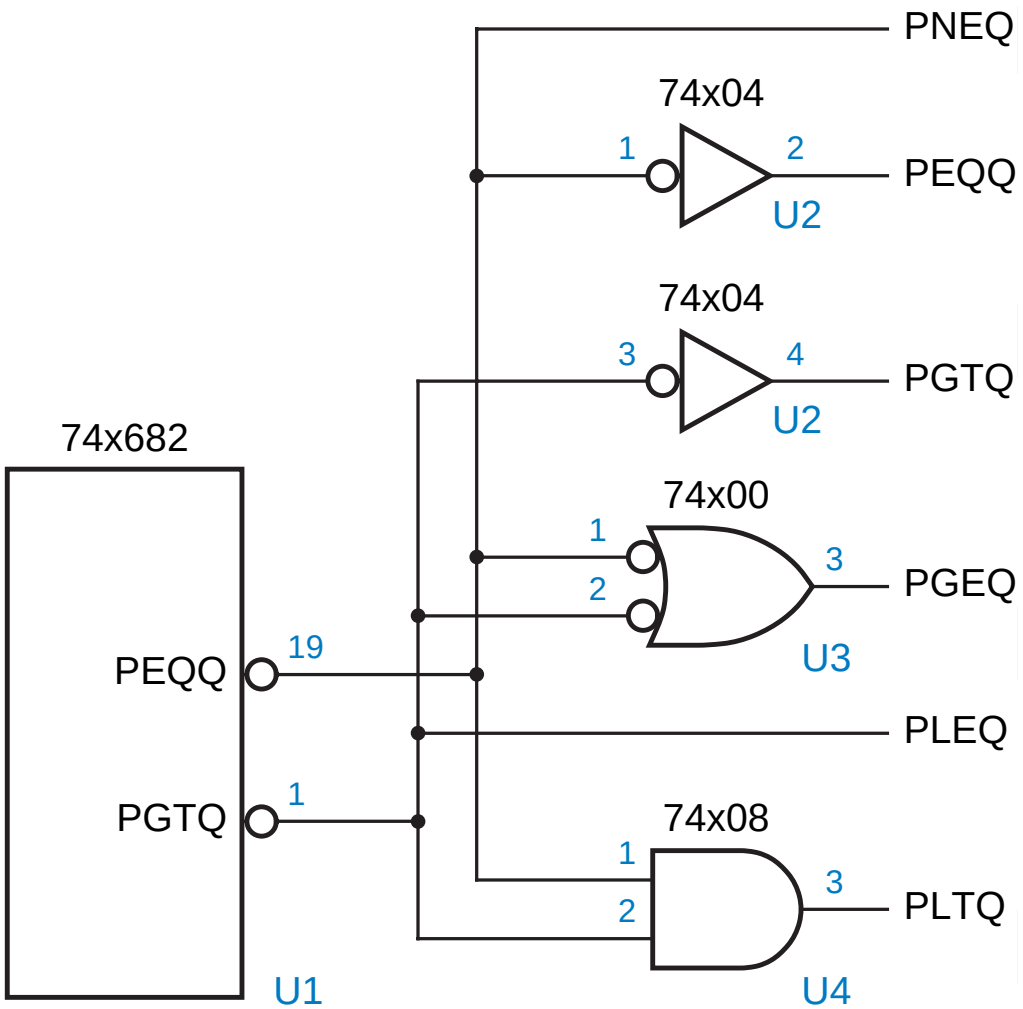


Figure 13: Arithmetic conditions derived from 74x682 outputs.

Comparators in HDLs

- Comparing two bit-vectors for equality or inequality is done in an HDL program, in relational expressions using operators such as "==" and "!="
- Given relational expression "(A==B)", where A and B are bit vectors each with n elements, compiler generates the logic expression

$$((A_1 \oplus B_1) + (A_2 \oplus B_2) + \cdots + (A_n \oplus B_n))'$$

  - In a PLD, this is realized as a complemented sum of $2n$ product terms

$$((A_1 \cdot B_1' + A_1' \cdot B_1) + (A_2 \cdot B_2' + A_2' \cdot B_2) + \cdots + (A_n \cdot B_n' + A_n' \cdot B_n))'$$

  - Logic expression for "(A!=B)" is complement of the ones above

Comparators in HDLs

- Given relational expression "(A<B)", where A and B are bit vectors each with n elements, HDL compiler first builds n equations of the form

$$L_i = (A_i' \cdot (B_i + L_{i-1})) + (A_i \cdot B_i \cdot L_{i-1})$$

for $i = 1$ to n, and $L_0 = 0$

  - This is an iterative definition of less-than function, starting with LSB
  - Logic equation for "(A<B)" is the equation for L_n
  - After creating n equations, HDL compiler collapses them into a single equation for L_n involving only A and B
  - In case of a compiler that is targeting a PLD, final step is to derive a minimal sum-of-products expression from L_n equation
  - Collapsing an iterative circuit into a two-level sum-of-products realization creates an exponential expansion of product terms
    - Requires $2^n - 1$ product terms for an n-bit comparator
  - Results for ">" comparators are identical
  - Logic expressions for ">=" and "<=" are complements of expressions for "<" and ">"

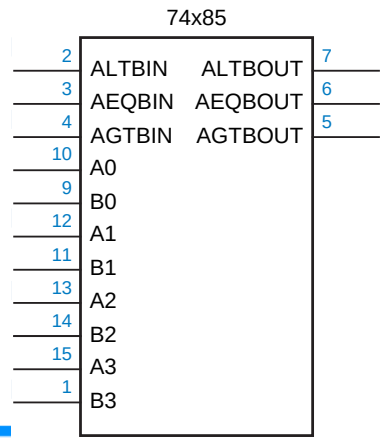
Comparators in Verilog

- Verilog has built-in comparison operators: `>`, `>=`, `<`, `<=`, `==`, `!=`
  - These operators can be applied to bit vectors
  - Bit vectors are interpreted as unsigned numbers with the MSB on left, regardless of how they are numbered
  - Verilog-2001 also supports signed arithmetic
  - Verilog matches up operands of different lengths, by adding zeros on left
  - Equality and inequality checkers are small and fast
    - Built from n XOR or XNOR gates and an n-input AND or OR gate
  - Checking for greater-than or less-than
    - The number of product terms needed for an n-bit comparator grows exponentially, on order of 2^n, when comparator is realized as a two-level sum of products
    - A two-level sum-of-products realization is possible only for small values of n (4 or less)
    - For larger values of n, compiler may synthesize a set of smaller comparator modules, along the lines of 74x85 and 74x682 parts, whose outputs may be cascaded or combined to create larger comparison result

Comparators in Verilog

Table 5: Verilog module with functionality similar to 74x85 magnitude comparator.

```
module Vr74x85(A, B, AGTBIN, ALTBIN, AEQBIN, AGTBOUT, ALTBOUT, AEQBOUT);
  input [3:0] A, B;
  input AGTBIN, ALTBIN, AEQBIN;
  output AGTBOUT, ALTBOUT, AEQBOUT;
  reg AGTBOUT, ALTBOUT, AEQBOUT;

  always @ (A or B or AGTBIN or ALTBIN or AEQBIN)
    if (A == B)
      begin AGTBOUT = AGTBIN; ALTBOUT = ALTBIN; AEQBOUT = AEQBIN; end
    else if (A > B)
      begin AGTBOUT = 1'b1; ALTBOUT = 1'b0; AEQBOUT = 1'b0; end
    else
      begin AGTBOUT = 1'b0; ALTBOUT = 1'b1; AEQBOUT = 1'b0; end
endmodule
```

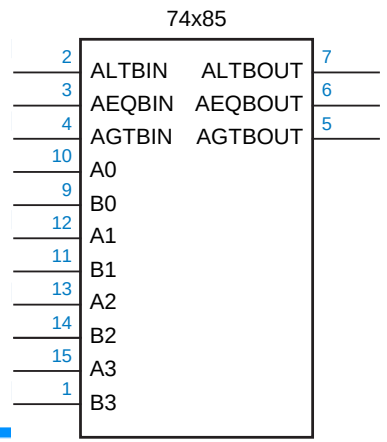


- Module of Tab. 5 does not perform an explicit check for `A<B`, to avoid synthesizing another comparator
  - If we missed this optimization and included `A<B` check, it is necessary also to include a final `else` statement (Tab. 6)
  - Without final `else` clause, compiler will infer a latch to hold previous value of each cascading output if none of logic paths through `always` block assigned a value to that output

Comparators in Verilog

Table 6: Verilog comparator module with three explicit comparisons.

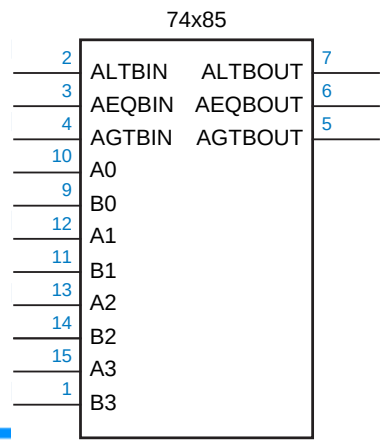
```
module Vr74x85s(A, B, AGTBIN, ALTBIN, AEQBIN, AGTBOUT, ALTBOUT, AEQBOUT);
  input [3:0] A, B;
  input AGTBIN, ALTBIN, AEQBIN;
  output AGTBOUT, ALTBOUT, AEQBOUT;
  reg AGTBOUT, ALTBOUT, AEQBOUT;

  always @ (A or B or AGTBIN or ALTBIN or AEQBIN)
    if (A == B)
      begin AGTBOUT = AGTBIN; ALTBOUT = ALTBIN; AEQBOUT = AEQBIN; end
    else if (A > B)
      begin AGTBOUT = 1'b1; ALTBOUT = 1'b0; AEQBOUT = 1'b0; end
    else if (A < B)
      begin AGTBOUT = 1'b0; ALTBOUT = 1'b1; AEQBOUT = 1'b0; end
    else
      begin AGTBOUT = 1'bx; ALTBOUT = 1'bx; AEQBOUT = 1'bx; end
endmodule
```



Comparators in Verilog

Table 7: Verilog comparator module with cascading from more to less significant stages.

```
module Vr74x85r(A, B, AGTBIN, ALTBIN, AEQBIN, AGTBOUT, ALTBOUT, AEQBOUT);
  input [3:0] A, B;
  input AGTBIN, ALTBIN, AEQBIN;
  output AGTBOUT, ALTBOUT, AEQBOUT;
  reg AGTBOUT, ALTBOUT, AEQBOUT;

  always @ (A or B or AGTBIN or ALTBIN or AEQBIN)
    if (AGTBIN)
      begin AGTBOUT = 1'b1; ALTBOUT = 1'b0; AEQBOUT = 1'b0; end
    else if (ALTBIN)
      begin AGTBOUT = 1'b0; ALTBOUT = 1'b1; AEQBOUT = 1'b0; end
    else if (AEQBIN)
      begin
        AGTBOUT = (A > B) ? 1'b1 : 1'b0 ;
        AEQBOUT = (A == B) ? 1'b1 : 1'b0;
        ALTBOUT = ~AGTBOUT & ~AEQBOUT;
      end
    else
      begin AGTBOUT = 1'bx; ALTBOUT = 1'bx; AEQBOUT = 1'bx; end
endmodule
```

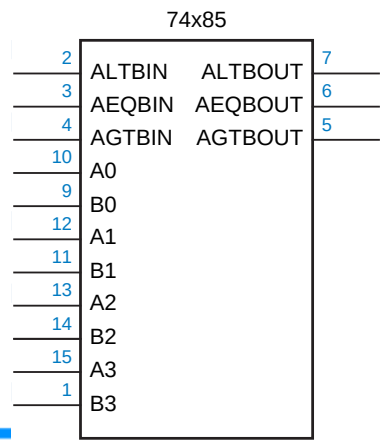


- With a series of `if-else` statements, compiler synthesizes priority logic
  - It checks the first condition, and only then the second, and so on
  - We can use a `case` statement instead

Comparators in Verilog

Table 8: Verilog comparator module using a case statement.

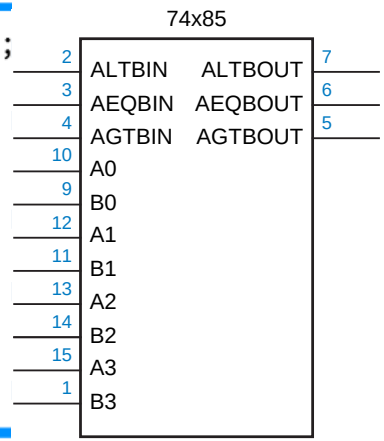
```
module Vr74x85rc(A, B, AGTBIN, ALTBIN, AEQBIN, AGTBOUT, ALTBOUT, AEQBOUT);
  input [3:0] A, B;
  input AGTBIN, ALTBIN, AEQBIN;
  output AGTBOUT, ALTBOUT, AEQBOUT;
  reg AGTBOUT, ALTBOUT, AEQBOUT;

  always @ (A or B or AGTBIN or ALTBIN or AEQBIN)
    case ({AGTBIN, ALTBIN, AEQBIN})
      3'b100: begin AGTBOUT = 1'b1; ALTBOUT = 1'b0; AEQBOUT = 1'b0; end
      3'b010: begin AGTBOUT = 1'b0; ALTBOUT = 1'b1; AEQBOUT = 1'b0; end
      3'b001: begin
                AGTBOUT = (A > B) ? 1'b1 : 1'b0 ;
                AEQBOUT = (A == B) ? 1'b1 : 1'b0;
                ALTBOUT = ~AGTBOUT & ~AEQBOUT;
              end
      default: begin AGTBOUT = 1'bx; ALTBOUT = 1'bx; AEQBOUT = 1'bx; end
    endcase
endmodule
```



Comparators in Verilog

Table 9: Verilog comparator module using continuous assignments.

```
module Vr74x85re(A, B, AGTBIN, ALTBIN, AEQBIN, AGTBOUT, ALTBOUT, AEQBOUT);
  input [3:0] A, B;
  input AGTBIN, ALTBIN, AEQBIN;
  output AGTBOUT, ALTBOUT, AEQBOUT;

  assign AGTBOUT = AGTBIN | (AEQBIN & ( (A > B) ? 1'b1 : 1'b0 ) );
  assign AEQBOUT = AEQBIN & ((A == B) ? 1'b1 : 1'b0) ;
  assign ALTBOUT = ~AGTBOUT & ~AEQBOUT;
endmodule
```



Adders, Subtractors, and ALUs

- The same addition rules and therefore the same adders are used for both unsigned and two's-complement numbers
- An adder can perform subtraction as addition of minuend and complemented subtrahend
  - But we can also build subtractor circuits that perform subtraction directly
- ALUs perform addition, subtraction, or any of several other operations according to an operation code supplied to device

Adders, Subtractors, and ALUs: Half Adders & Full Adders

- A half adder adds two 1-bit operands X and Y, producing a 2-bit sum

$$\begin{aligned} HS &= X \oplus Y \\ &= X \cdot Y' + X' \cdot Y \\ CO &= X \cdot Y \end{aligned}$$

(HS = half sum, and CO = carry-out)

- To add operands with more than one bit, we must provide for carries between bit positions
  - Building block for this operation is called a full adder

$$\begin{aligned} S &= X \oplus Y \oplus CIN \\ &= X \cdot Y' \cdot CIN' + X' \cdot Y \cdot CIN' + X' \cdot Y' \cdot CIN + X \cdot Y \cdot CIN \\ COUT &= X \cdot Y + X \cdot CIN + Y \cdot CIN \end{aligned}$$

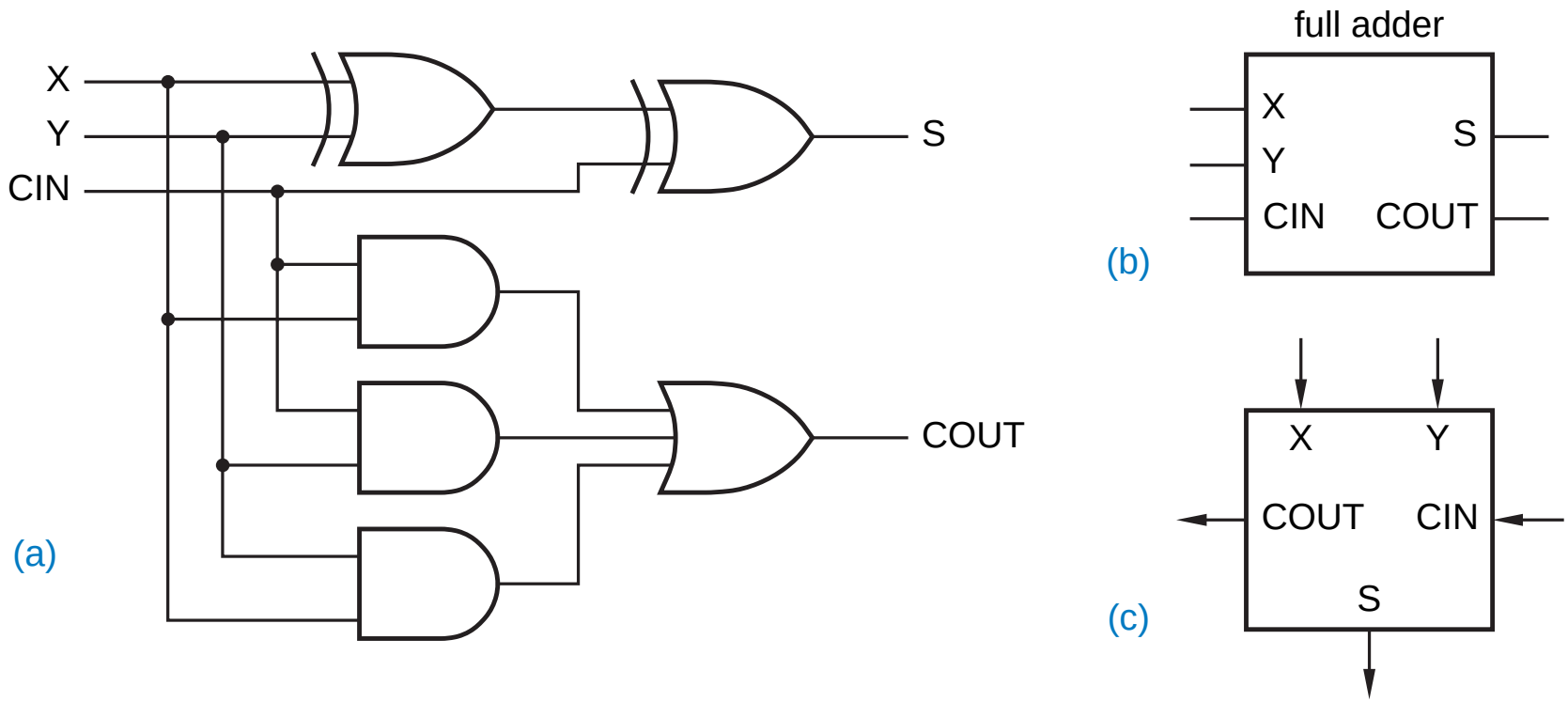


Figure 14: Full adder: (a) gate-level circuit diagram; (b) logic symbol; (c) alternate logic symbol suitable for cascading.

Adders, Subtractors, and ALUs: Ripple Adders

- A ripple adder is a cascade of n full-adder stages, each of which handles one bit, to add two n-bit binary words

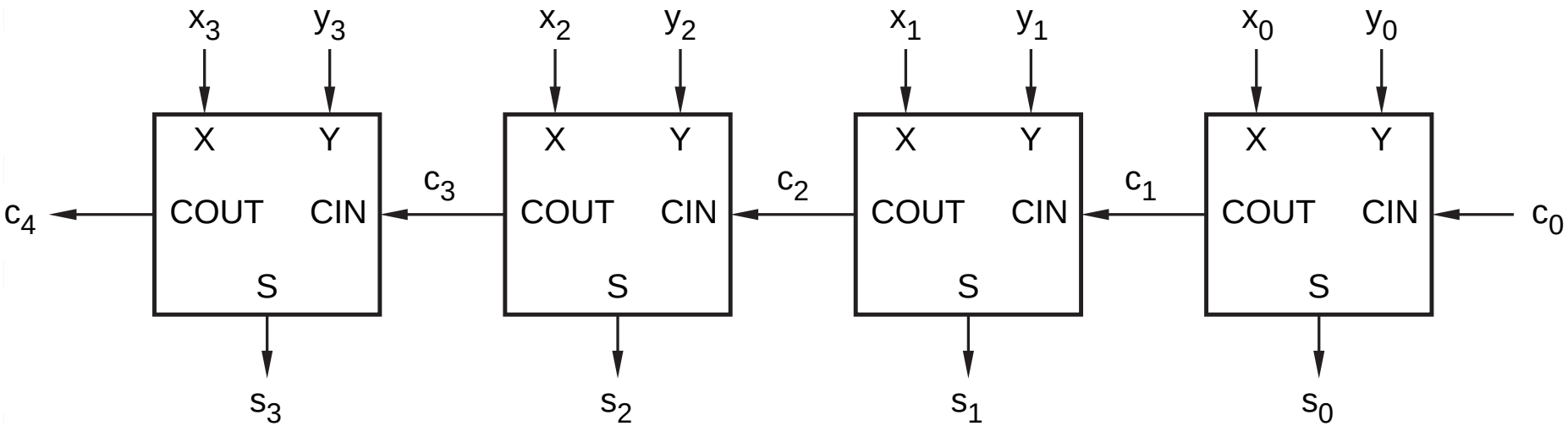


Figure 15: A 4-bit ripple adder.

- c_0 is normally set to 0

Adders, Subtractors, and ALUs: Ripple Adders

- A ripple adder is slow
  - In worst case, a carry must propagate from least significant full adder to most significant one
  - E.g., adding $11\ldots11$ and $00\ldots01$
  - Total worst-case delay

$$t_{ADD} = t_{XYCout} + (n-2) \cdot t_{CinCout} + t_{CinS}$$

    - t_{XYCout}: delay from X or Y to COUT in least significant stage
    - $t_{CinCout}$: delay from CIN to COUT in middle stages
    - t_{CinS}: delay from CIN to S in most significant stage

Adders, Subtractors, and ALUs: Subtractors

- Binary subtraction is performed similar to binary addition, but using borrows (b_{in} and b_{out}) between steps, and producing a difference bit d
- When subtracting y from x

$$x \geq y + b_{in} \longrightarrow b_{out} = 0$$
$$x < y + b_{in} \longrightarrow b_{out} = 1$$
$$d = x - y - b_{in} + 2b_{out}$$

Table 10: Binary subtraction table.

b_{in}	x	y	b_{out}	d
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

Adders, Subtractors, and ALUs: Subtractors

- Logic equations for a full subtractor

$$D = X \oplus Y \oplus BIN$$
$$BOUT = X' \cdot Y + X' \cdot BIN + Y \cdot BIN$$

- Manipulating logic equations above

$$BOUT = X' \cdot Y + X' \cdot BIN + Y \cdot BIN$$
$$\begin{aligned} BOUT' &= (X + Y') \cdot (X + BIN') \cdot (Y' + BIN') \\ &= X \cdot Y' + X \cdot BIN' + Y' \cdot BIN' \end{aligned}$$
$$\begin{aligned} D &= X \oplus Y \oplus BIN \\ &= X \oplus Y' \oplus BIN' \end{aligned}$$

- Comparing with equations for a full adder, we can build a full subtractor from a full adder
  - Any n-bit adder circuit can be made to function as a subtractor by complementing subtrahend and treating carry-in and carry-out signals as borrows with opposite active level

Adders, Subtractors, and ALUs: Subtractors

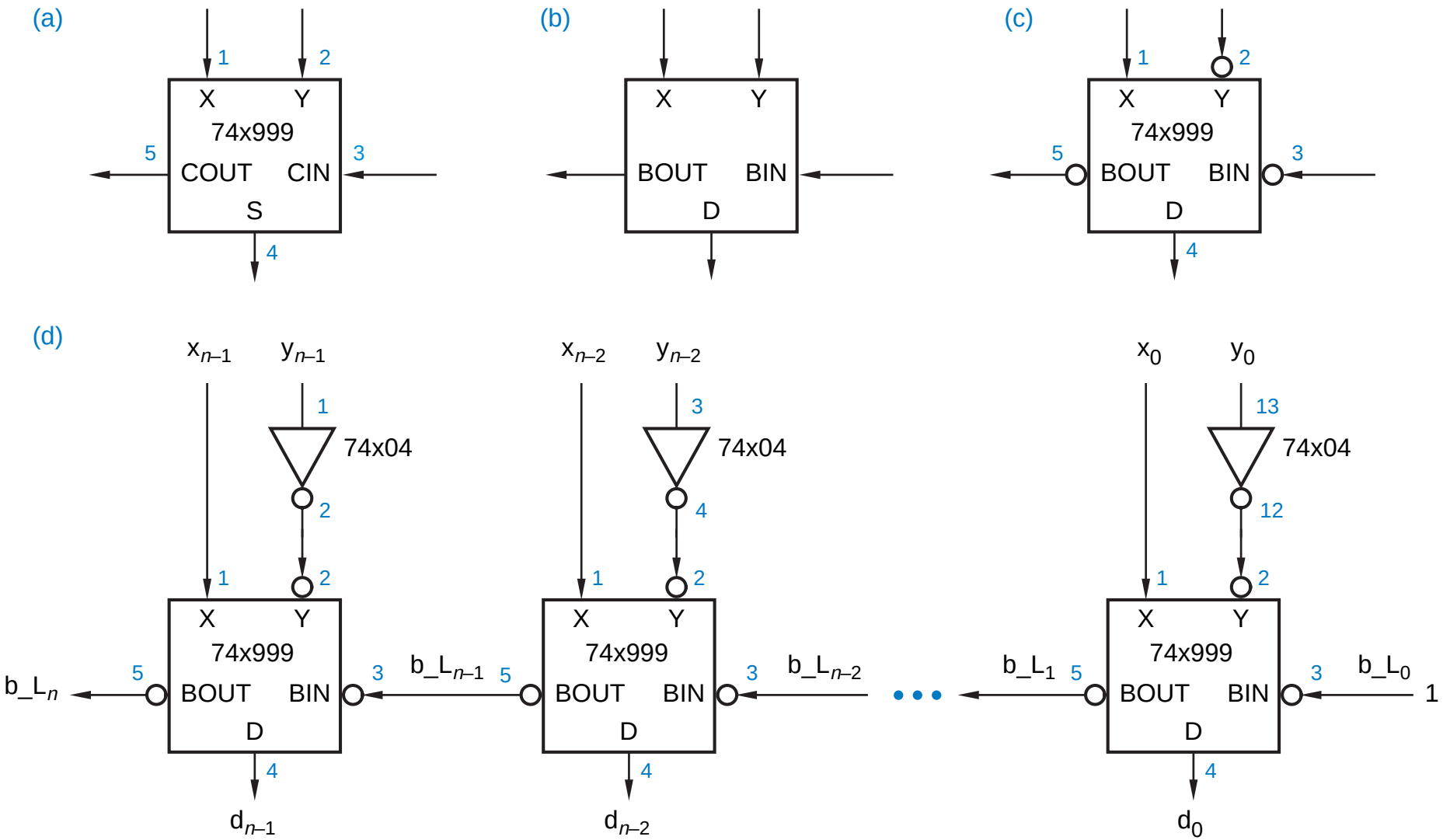


Figure 16: Subtractor design using adders: (a) full adder; (b) full subtractor; (c) interpreting 74x999 as a full subtractor; (d) ripple subtractor.

Adders, Subtractors, and ALUs: Carry-Lookahead Adders

- A faster adder than ripple can be built by obtaining each sum output $s_i = x_i \oplus y_i \oplus c_i$ with just two levels of logic
  - This can be accomplished by expanding c_i in terms of $x_0 - x_{i-1}$, $y_0 - y_{i-1}$, and c_0
  - More complexity is introduced by expanding XORs
  - We can keep XORs and design c_i logic using ideas of carry lookahead

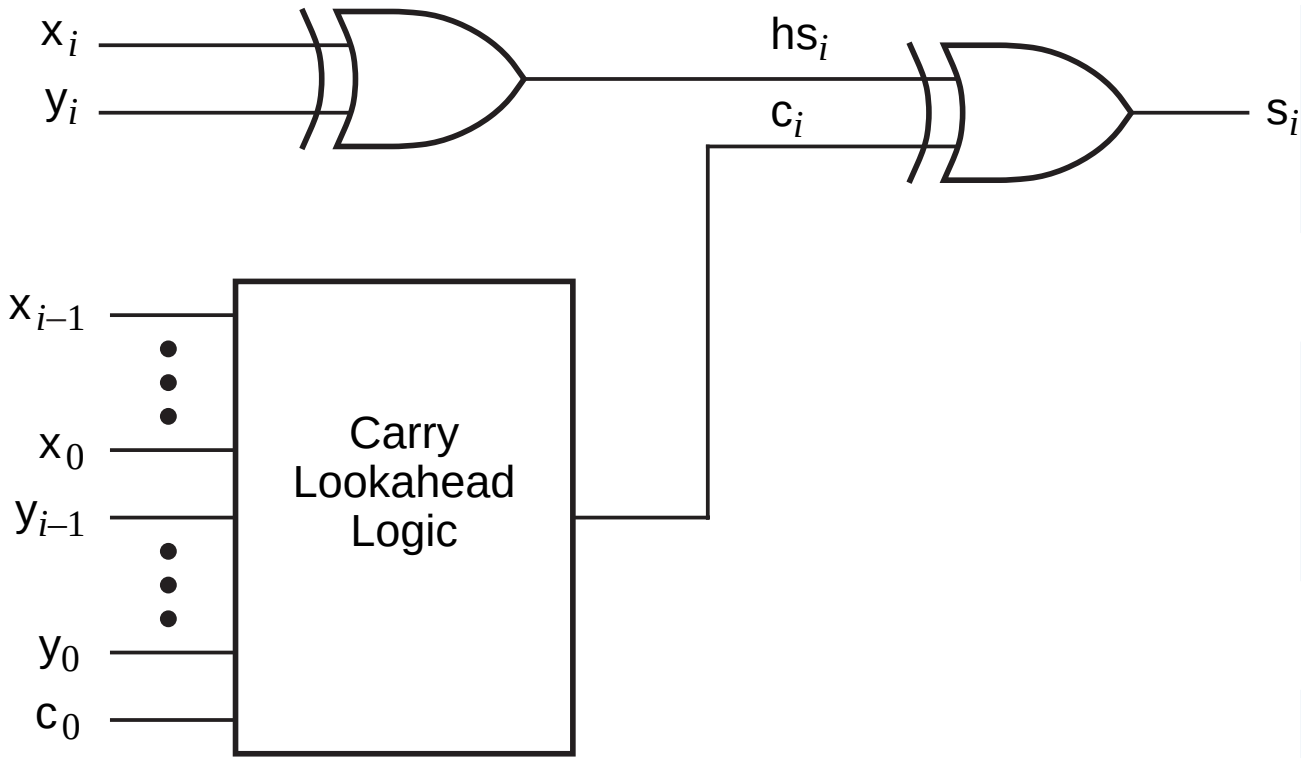


Figure 17: Structure of one stage of a carry-lookahead adder.

Adders, Subtractors, and ALUs: Carry-Lookahead Adders

- Adder stage i is said to **generate** a carry if it produces a $c_{i+1} = 1$ independent of inputs on $x_0 - x_{i-1}$, $y_0 - y_{i-1}$, and c_0
  - This happens when both of addend bits of that stage are 1

$$g_i = x_i \cdot y_i$$

- Adder stage i is said to **propagate** carries if it produces a $c_{i+1} = 1$ in presence of an input combination of $x_0 - x_{i-1}$, $y_0 - y_{i-1}$, and c_0 that causes a $c_i = 1$
  - This happens when at least one of addend bits of that stage is 1

$$p_i = x_i + y_i$$

- Carry output of a stage can be written as

$$c_{i+1} = g_i + p_i \cdot c_i$$

Adders, Subtractors, and ALUs: Carry-Lookahead Adders

- To eliminate carry ripple, we recursively expand c_i term for each stage and multiply out to obtain a two-level AND-OR expression

$$
\begin{aligned}
c_1 &= g_0 + p_0 \cdot c_0 \\
c_2 &= g_1 + p_1 \cdot c_1 \\
&= g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) \\
&= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \\
c_3 &= g_2 + p_2 \cdot c_2 \\
&= g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0) \\
&= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \\
c_4 &= g_3 + p_3 \cdot c_3 \\
&= g_3 + p_3 \cdot (g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0) \\
&= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0
\end{aligned}
$$

Hence, "Carry Lookahead Logic" in Fig. 17 has three levels of delay; one for generate and propagate signals, and two for SOPs shown

- 74x283 uses carry-lookahead technique

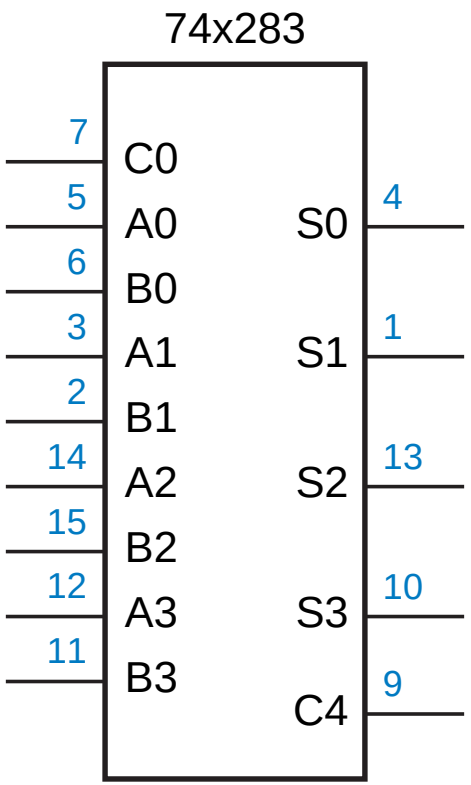


Figure 18: Traditional logic symbol for the 74x283 4-bit binary adder.

Adders, Subtractors, and ALUs: MSI Adders

- 74x283
  - It produces g_i' and p_i', since inverting gates are faster
  - Manipulating half-sum equation

$$\begin{aligned} hs_i &= x_i \oplus y_i \\ &= x_i \cdot y_i' + x_i' \cdot y_i \\ &= x_i \cdot y_i' + x_i \cdot x_i' + x_i' \cdot y_i + y_i \cdot y_i' \\ &= (x_i + y_i) \cdot (x_i' + y_i') \\ &= (x_i + y_i) \cdot (x_i \cdot y_i)' \\ &= p_i \cdot g_i' \end{aligned}$$

Thus, an AND gate with an inverted input is used instead of an XOR gate to create each half-sum bit

Adders, Subtractors, and ALUs: MSI Adders

- 74x283
  - It creates carry signals using an INVERT-OR-AND structure ($\equiv$ AND-OR-INVERT), which has same delay as a single inverting gate

$$c_{i+1} = g_i + p_i \cdot c_i = p_i \cdot g_i + p_i \cdot c_i = p_i \cdot (g_i + c_i)$$

(p_i is always 1 when g_i is 1)

$$\begin{aligned}
c_1 &= p_0 \cdot (g_0 + c_0) \\
c_2 &= p_1 \cdot (g_1 + c_1) \\
&= p_1 \cdot (g_1 + p_0 \cdot (g_0 + c_0)) \\
&= p_1 \cdot (g_1 + p_0) \cdot (g_1 + g_0 + c_0) \\
c_3 &= p_2 \cdot (g_2 + c_2) \\
&= p_2 \cdot (g_2 + p_1 \cdot (g_1 + p_0) \cdot (g_1 + g_0 + c_0)) \\
&= p_2 \cdot (g_2 + p_1) \cdot (g_2 + g_1 + p_0) \cdot (g_2 + g_1 + g_0 + c_0) \\
c_4 &= p_3 \cdot (g_3 + c_3) \\
&= p_3 \cdot (g_3 + p_2 \cdot (g_2 + p_1) \cdot (g_2 + g_1 + p_0) \cdot (g_2 + g_1 + g_0 + c_0)) \\
&= p_3 \cdot (g_3 + p_2) \cdot (g_3 + g_2 + p_1) \cdot (g_3 + g_2 + g_1 + p_0) \\
&\quad \cdot (g_3 + g_2 + g_1 + g_0 + c_0)
\end{aligned}$$

Adders, Subtractors, and ALUs: MSI Adders

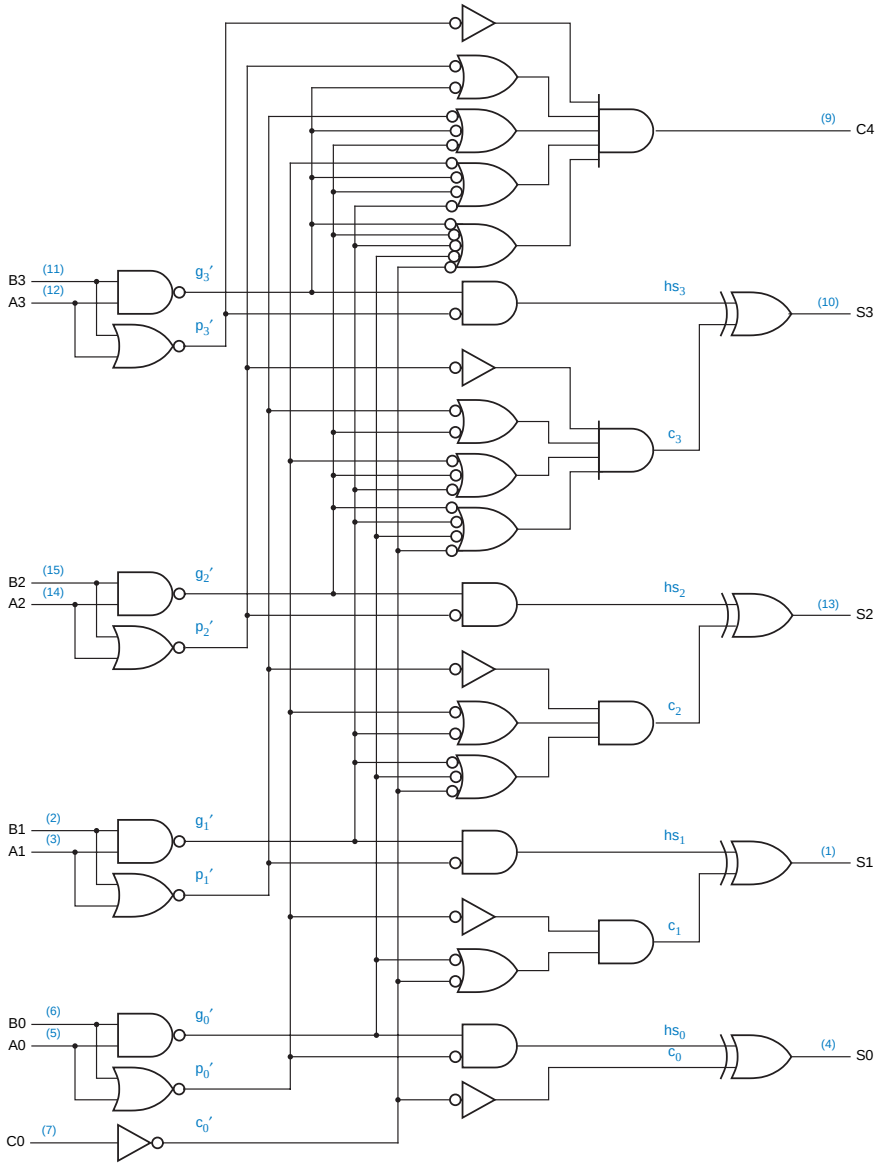


Figure 19: Logic diagram for the 74x283 4-bit binary adder.

Adders, Subtractors, and ALUs: MSI Adders

- Propagation delay from $C0$ input to $C4$ output of '283 is very short, same as two inverting gates
  - As a result, fast **group-ripple adders** with more than four bits can be made by cascading carry outputs and inputs of '283s
  - Total propagation delay from $C0$ to $C16$ in Fig. 20 is same as that of eight inverting gates

Adders, Subtractors, and ALUs: MSI Adders

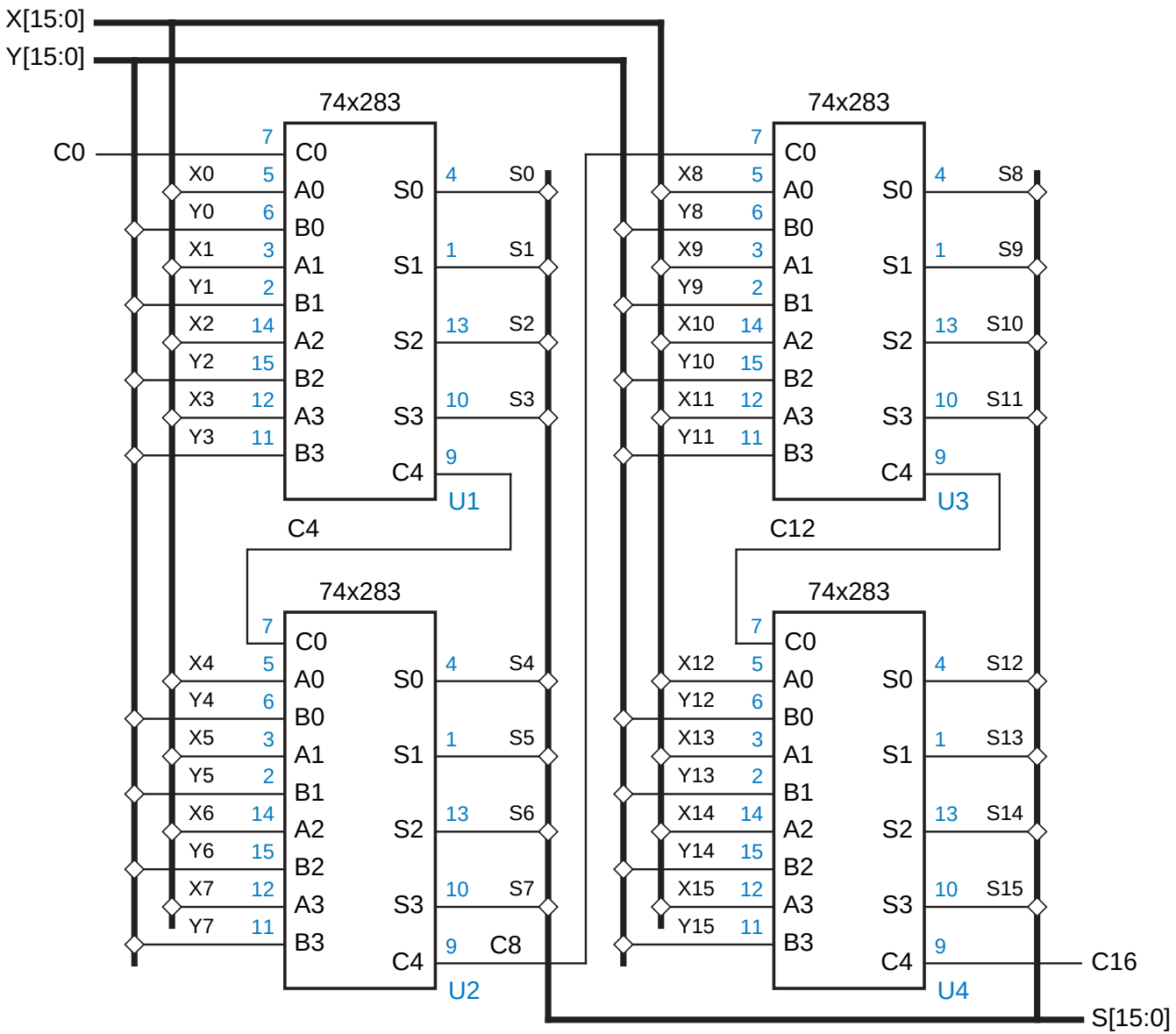


Figure 20: A 16-bit group-ripple adder.

Adders, Subtractors, and ALUs: MSI ALUs

- An arithmetic and logic unit (ALU) is a combinational circuit that can perform any of a number of different arithmetic and logical operations on a pair of b-bit operands
  - The operation to be performed is specified by a set of function-select inputs

Table 11: Functions performed by the 74x181 4-bit ALU.

Inputs				Function	
S3	S2	S1	S0	$M = 0$ (arithmetic)	$M = 1$ (logic)
0	0	0	0	$F = A$ minus 1 plus CIN	$F = A'$
0	0	0	1	$F = A \cdot B$ minus 1 plus CIN	$F = A' + B'$
0	0	1	0	$F = A \cdot B'$ minus 1 plus CIN	$F = A' + B$
0	0	1	1	$F = 1111$ plus CIN	$F = 1111$
0	1	0	0	$F = A$ plus $(A + B')$ plus CIN	$F = A' \cdot B'$
0	1	0	1	$F = A \cdot B$ plus $(A + B')$ plus CIN	$F = B'$
0	1	1	0	$F = A$ minus B minus 1 plus CIN	$F = A \oplus B'$
0	1	1	1	$F = A + B'$ plus CIN	$F = A + B'$
1	0	0	0	$F = A$ plus $(A + B)$ plus CIN	$F = A' \cdot B$
1	0	0	1	$F = A$ plus B plus CIN	$F = A \oplus B$
1	0	1	0	$F = A \cdot B'$ plus $(A + B)$ plus CIN	$F = B$
1	0	1	1	$F = A + B$ plus CIN	$F = A + B$
1	1	0	0	$F = A$ plus A plus CIN	$F = 0000$
1	1	0	1	$F = A \cdot B$ plus A plus CIN	$F = A \cdot B'$
1	1	1	0	$F = A \cdot B'$ plus A plus CIN	$F = A \cdot B$
1	1	1	1	$F = A$ plus CIN	$F = A$

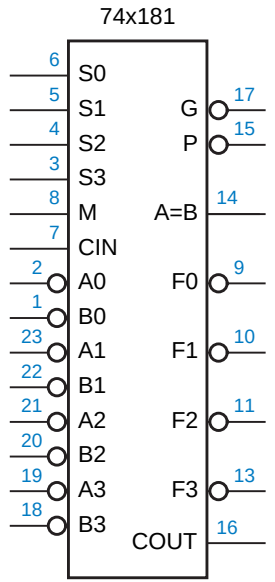


Figure 21: Logic symbol for the 74x181 4-bit ALU.

Adders, Subtractors, and ALUs: MSI ALUs

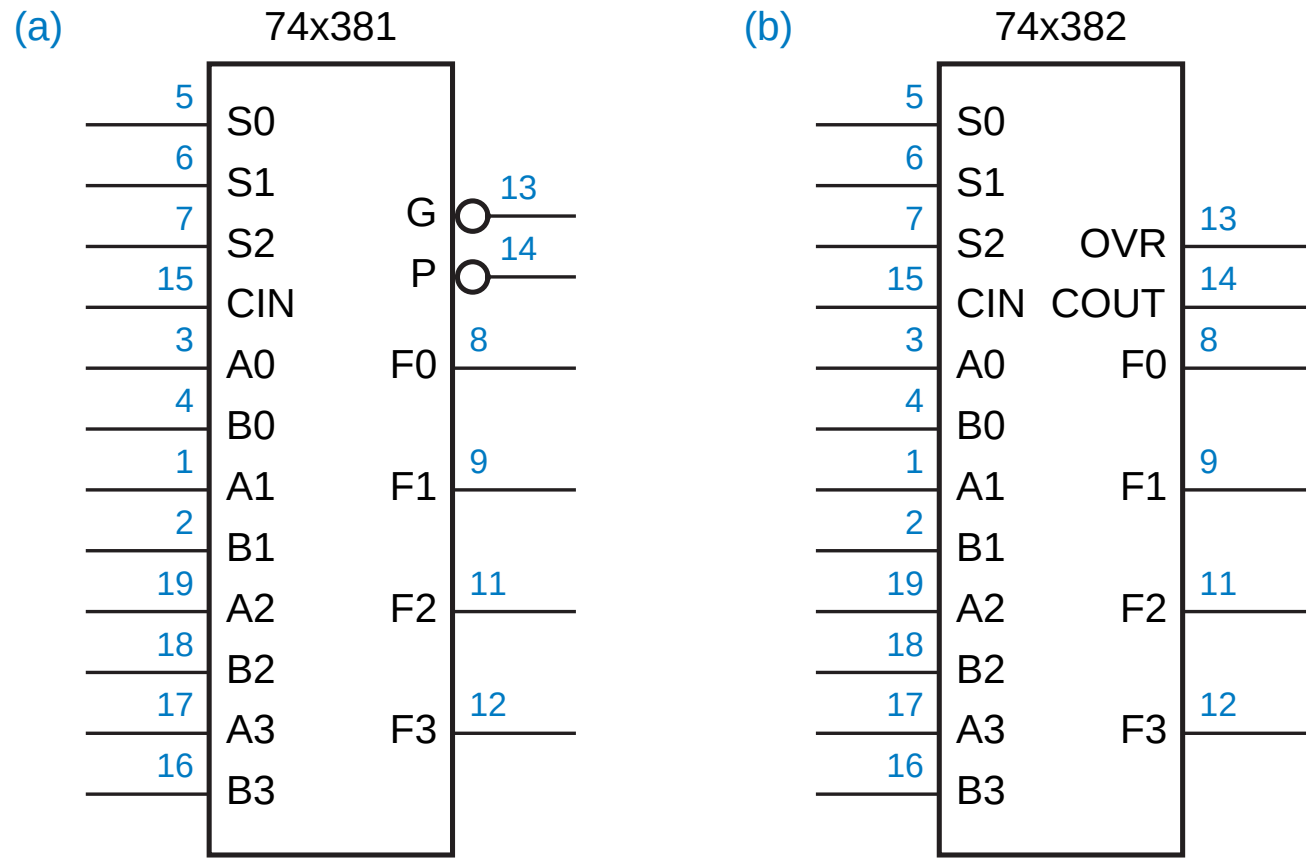


Figure 22: Logic symbols for 4-bit ALUs: (a) 74x381; (b) 74x382.

Table 12: Functions performed by the 74x381 and 74x382 4-bit ALUs.

Inputs			
S2	S1	S0	Function
0	0	0	$F = 0000$
0	0	1	$F = B$ minus A minus 1 plus CIN
0	1	0	$F = A$ minus B minus 1 plus CIN
0	1	1	$F = A$ plus B plus CIN
1	0	0	$F = A \oplus B$
1	0	1	$F = A + B$
1	1	0	$F = A \cdot B$
1	1	1	$F = 1111$

Adders, Subtractors, and ALUs: MSI ALUs

- '381 provides group-carry-lookahead outputs while '382 provides ripple carry and overflow outputs

Adders, Subtractors, and ALUs: Group-Carry Lookahead

- '181 and '381 provide group-carry-lookahead outputs that allow multiple ALUs to be cascaded without rippling carries between 4-bit groups
  - Like 74x283, ALUs use carry lookahead to produce carries internally
  - However, they also provide G_L and P_L outputs that are carry-lookahead signals for entire 4-bit group
- G_L output is asserted if ALU produces a carry-out whether or not there is a carry-in

$$G\_L = (g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0)'$$

- P_L output is asserted if ALU produces a carry-out if there is a carry-in

$$P\_L = (p_3 \cdot p_2 \cdot p_1 \cdot p_0)'$$

Adders, Subtractors, and ALUs: Group-Carry Lookahead

- When ALUs are cascaded, group-carry-lookahead outputs may be combined in just two levels of logic to produce carry input to each ALU
  - A lookahead carry circuit performs this operation

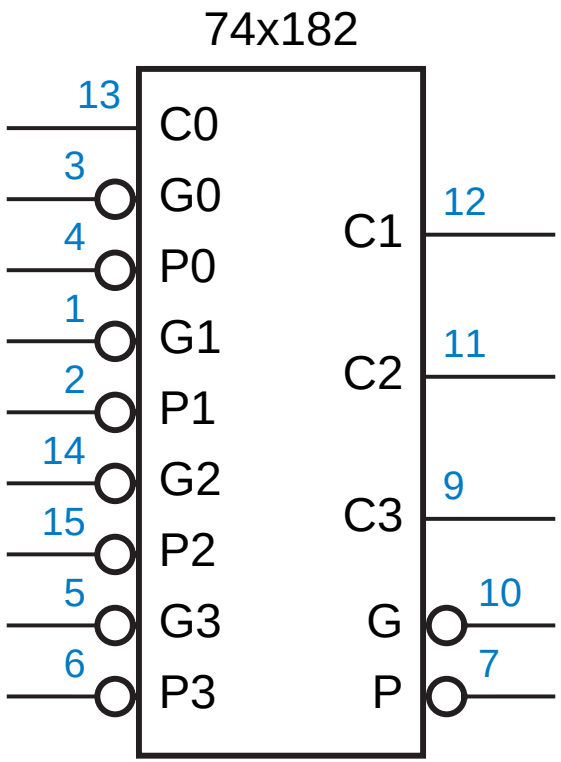


Figure 23: Logic symbol for the 74x182 lookahead carry circuit.

Adders, Subtractors, and ALUs: Group-Carry Lookahead

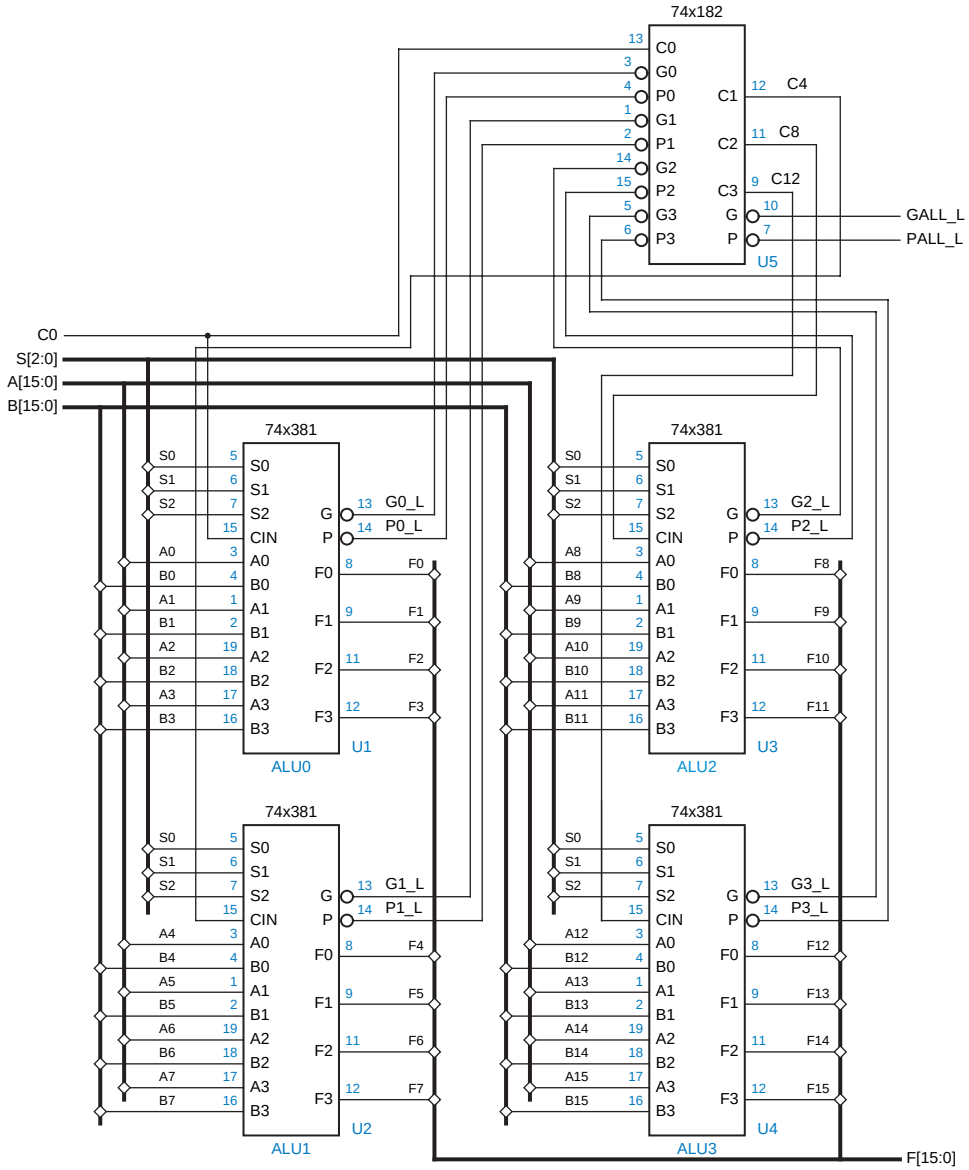


Figure 24: A 16-bit ALU using group-carry lookahead.

Adders, Subtractors, and ALUs: Group-Carry Lookahead

- '182's carry equations
  - '182 realizes each of these equations with one level of delay—an INVERT-OR-AND gate

$$c_{i+1} = g_i + p_i \cdot c_i = (g_i + p_i) \cdot (g_i + c_i)$$

$$C1 = (G0 + P0) \cdot (G0 + C0)$$
$$C2 = (G1 + P1) \cdot (G1 + G0 + P0) \cdot (G1 + G0 + C0)$$
$$C3 = (G2 + P2) \cdot (G2 + G1 + P1) \cdot (G2 + G1 + G0 + P0) \cdot (G2 + G1 + G0 + C0)$$

Adders, Subtractors, and ALUs: Group-Carry Lookahead

- When more than four ALUs are cascaded, they may be partitioned into "supergroups," each with its own '182
  - E.g., a 64-bit adder would have four supergroups, each containing four ALUs and a '182
  - G_L and P_L outputs of each '182 can be combined in a next-level '182, since they indicate whether the supergroup generates or propagates carries

$$G\_L = ((G3 + P3) \cdot (G3 + G2 + P2) \cdot (G3 + G2 + G1 + P1) \cdot (G3 + G2 + G1 + G0))'$$

$$P\_L = (P0 \cdot P1 \cdot P2 \cdot P3)'$$

Adders, Subtractors, and ALUs: Adders in Verilog

- Verilog has built-in addition ($+$) and subtraction ($-$) operators for bit vectors
  - Bit vectors are considered to be unsigned or two's-complement signed numbers
  - Actual addition or subtraction operation is exactly the same for either interpretation of bit vectors
  - Since exactly the same logic circuit is synthesized for either interpretation, Verilog compiler does not need to know how we are interpreting bit vectors
  - Only handling of carry, borrow, and overflow conditions differs by interpretation, and that is done separately from addition or subtraction itself

Table 13: Verilog program with addition of both signed and unsigned numbers.

```
module Vradders(A, B, C, D, S, T, OVFL, COUT);
  input [7:0] A, B, C, D;
  output [7:0] S, T;
  output OVFL, COUT;

  // S and OVFL -- signed interpretation
  assign S = A + B;
  assign OVFL = (A[7]==B[7]) && (S[7]!=A[7]);
  // T and COUT -- unsigned interpretation
  assign {COUT, T} = C + D;
endmodule
```

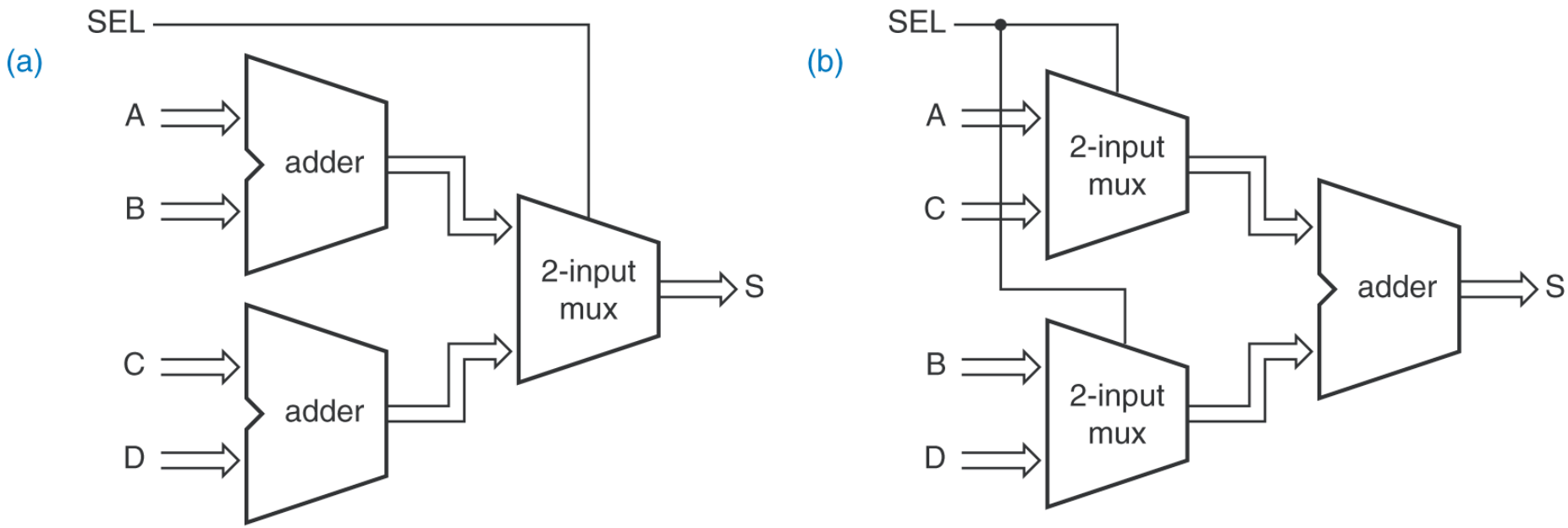


Figure 25: Two ways to synthesize a selectable addition: (a) two adders and a selectable sum; (b) one adder with selectable inputs.

Adders, Subtractors, and ALUs: Adders in Verilog

- Addition and subtraction are expensive in terms of number of gates required
  - Most Verilog compilers attempt to reuse adder blocks whenever possible
- Tab. 14 is a Verilog module that includes two different additions
  - Fig. 25(a) shows a circuit that might be synthesized
  - However, many compilers are smart enough to use approach (b)
  - An n-bit 2-input multiplexer is smaller than an n-bit adder

Table 14: Verilog module that allows adder sharing.

```
module Vraddersh(SEL, A, B, C, D, S);
  input SEL;
  input [7:0] A, B, C, D;
  output [7:0] S;
  reg [7:0] S;

  always @ (SEL, A, B, C, D)
    if (SEL) S = A + B;
    else S = C + D;
endmodule
```

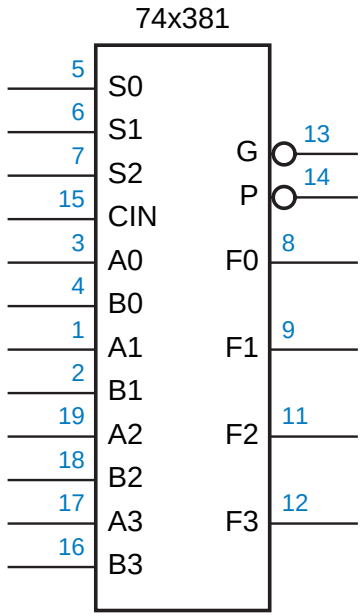
Table 15: Alternate version of Tab. 14, using a continuous-assignment statement.

```
module Vraddersc(SEL, A, B, C, D, S);
  input SEL;
  input [7:0] A, B, C, D;
  output [7:0] S;

  assign S = (SEL) ? A + B : C + D;
endmodule
```

- A typical compiler should synthesize the same circuit for either module of Tab. 14 or 15

Table 16: An 8-bit 74x381-like ALU.

```
module Vr74x381(S, A, B, CIN, F, G_L, P_L);
  input [2:0] S;
  input [7:0] A, B;
  input CIN;
  output [7:0] F;
  output G_L, P_L;
  reg [7:0] F;
  reg G_L, P_L, GG, GP;
  reg [7:0] G, P;
  integer i;

  always @ (S or A or B or CIN or G or P or GG or GP) begin
    for (i = 0; i <= 7; i = i + 1) begin
      G[i] = A[i] & B[i];
      P[i] = A[i] | B[i];
    end
    GG = G[0]; GP = P[0];
    for (i = 1; i <= 7; i = i + 1) begin
      GG = G[i] | (GG & P[i]);
      GP = P[i] & GP;
    end
    G_L = ~GG;  P_L = ~GP;
    case (S)
      3'd0: F = 8'b0;
      3'd1: F = B - A - 1 + CIN;
      3'd2: F = A - B - 1 + CIN;
      3'd3: F = A + B + CIN;
      3'd4: F = A ^ B;
      3'd5: F = A | B;
      3'd6: F = A & B;
      3'd7: F = 8'b11111111;
      default: F = 8'b0;
    endcase
  end
endmodule
```



Inputs			
S2	S1	S0	Function
0	0	0	$F = 0000$
0	0	1	$F = B$ minus A minus 1 plus CIN
0	1	0	$F = A$ minus B minus 1 plus CIN
0	1	1	$F = A$ plus B plus CIN
1	0	0	$F = A \oplus B$
1	0	1	$F = A + B$
1	1	0	$F = A \cdot B$
1	1	1	$F = 1111$

References

John F. Wakerly, *Digital Design: Principles and Practices (4th Edition)*, Prentice Hall, 2005.