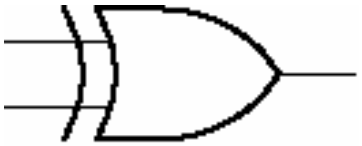
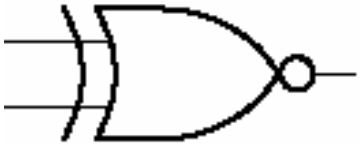
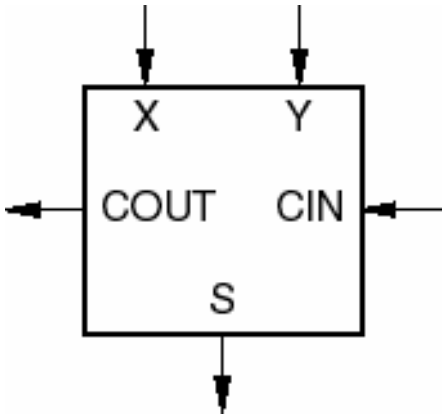




Combinatorial Logic Design Practices

ECGR2181



Reading: Chapter 6

Documentation Standards

Block diagrams

- first step in hierarchical design

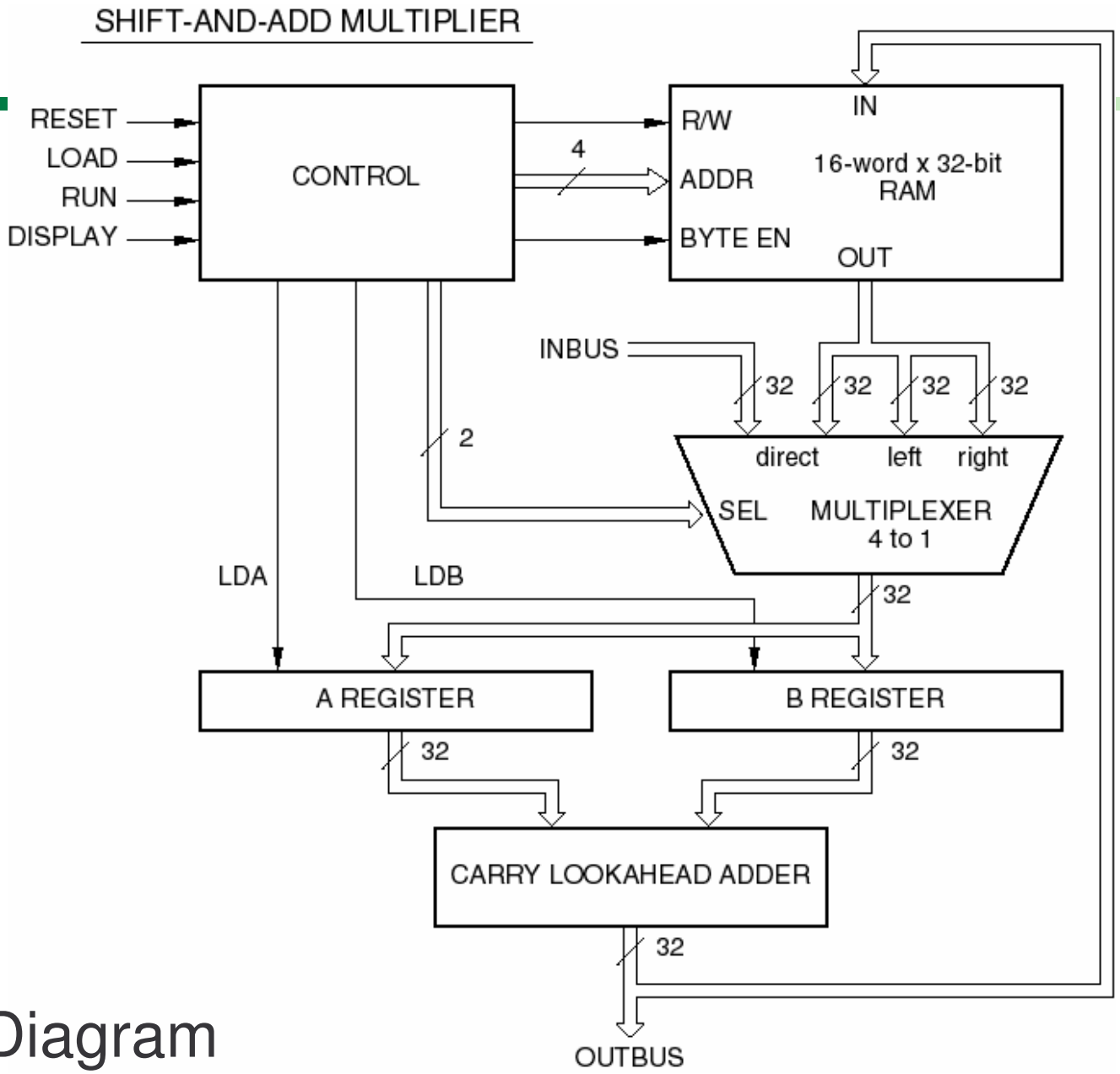
Schematic diagrams

HDL programs (ABEL, Verilog, VHDL)

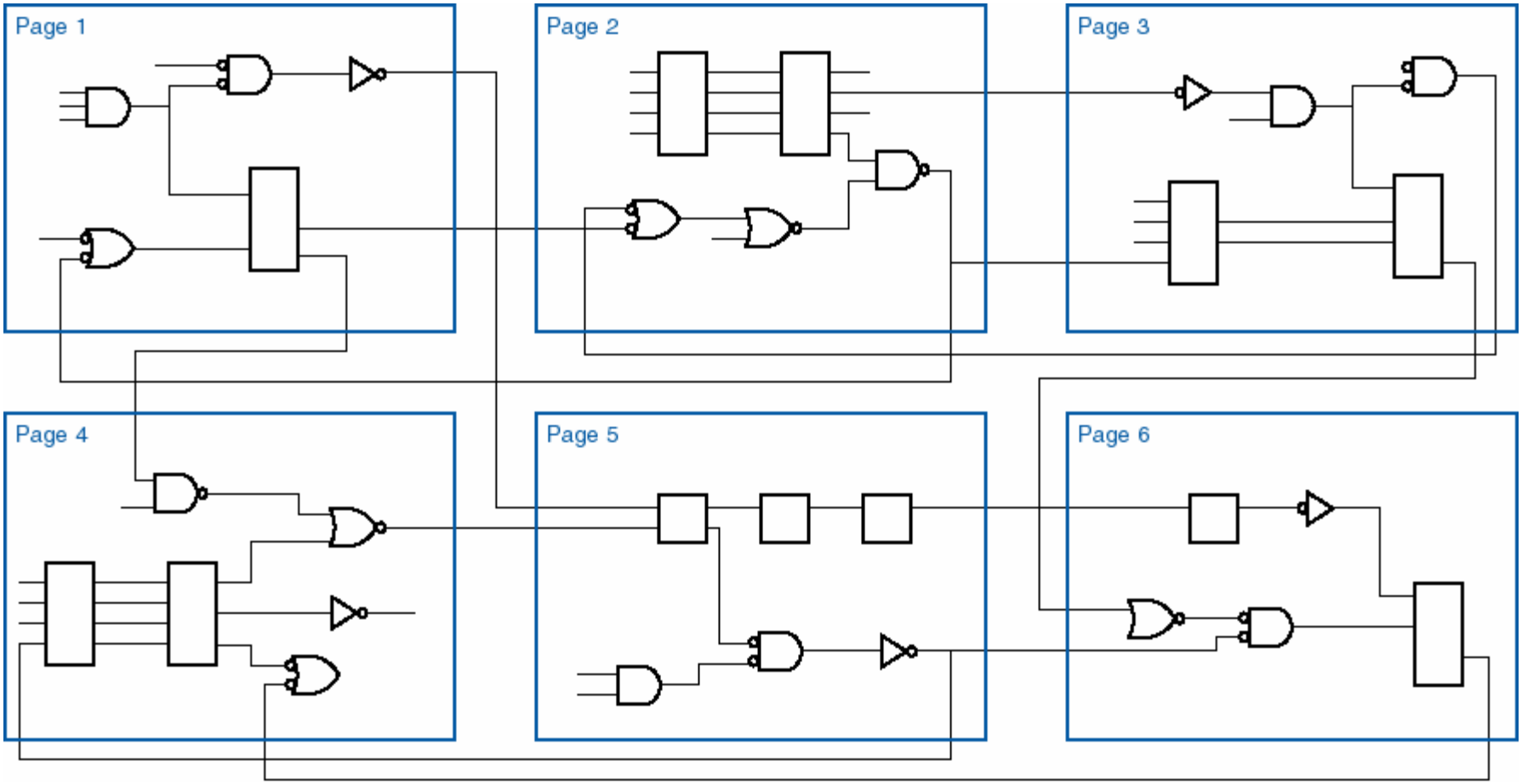
Timing diagrams

Circuit descriptions

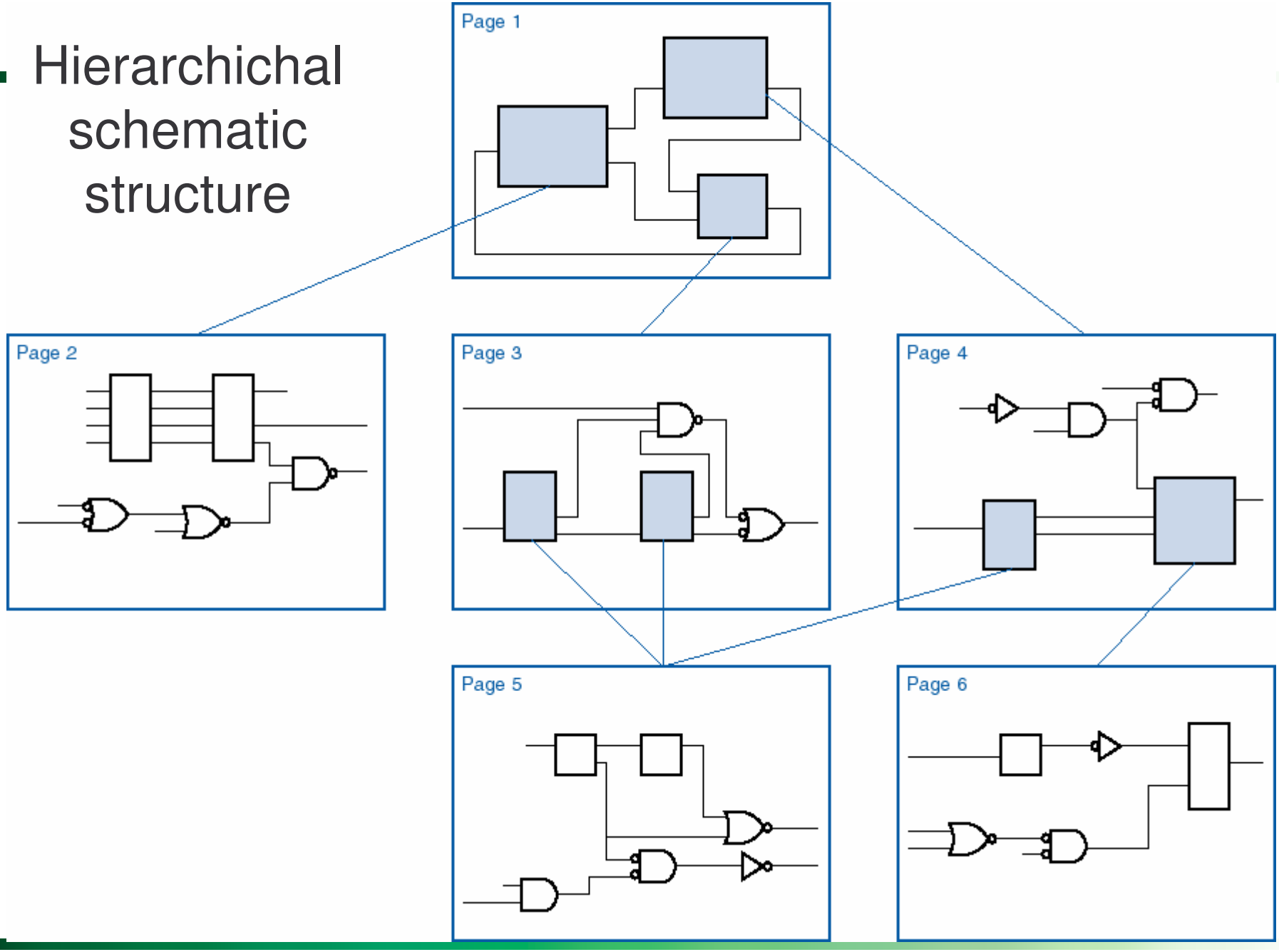
Block Diagram



Flat schematic structure



Hierarchichal schematic structure



Other Documentation

Timing diagrams

- Output from simulator
- Specialized timing-diagram drawing tools

Circuit descriptions

- Text (word processing)
- Can be as big as a book
- Typically incorporate other elements (block diagrams, timing diagrams, etc.)

Signal names and active levels

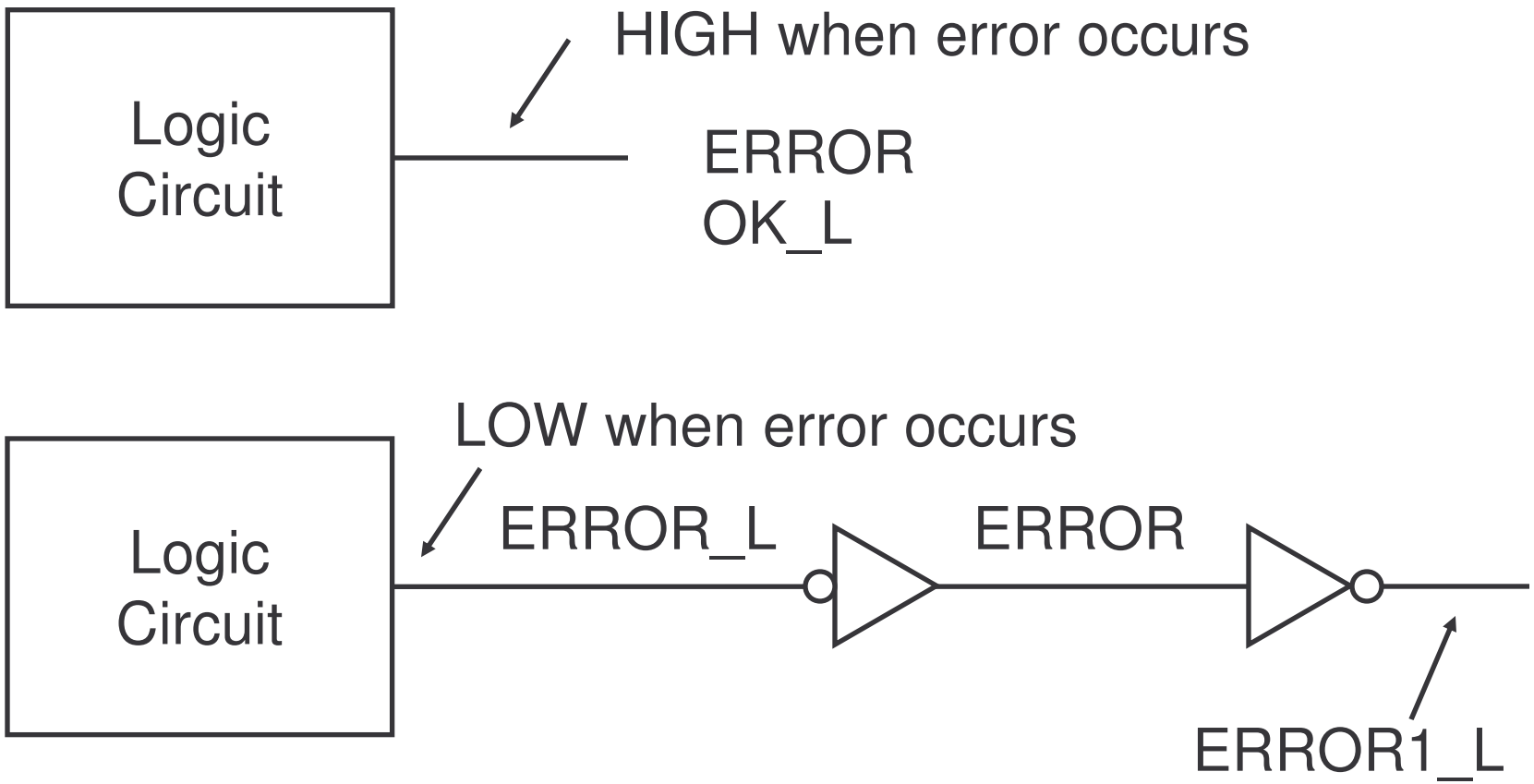
Signal names are chosen to be descriptive.

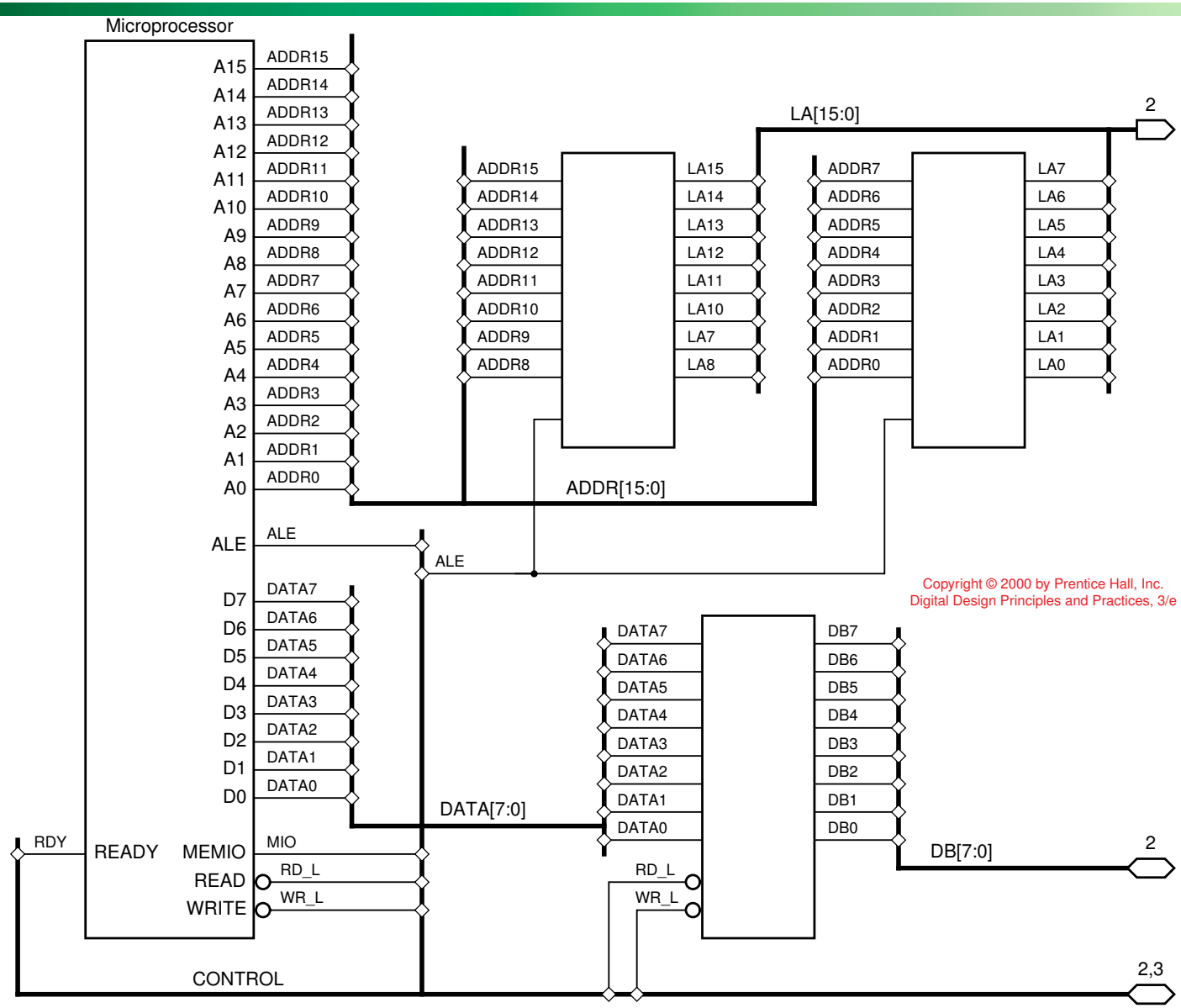
Active levels -- HIGH or LOW

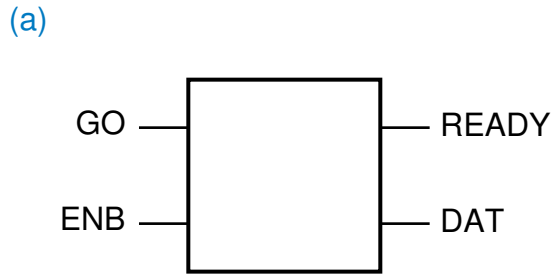
– named condition or action occurs in either the HIGH or the LOW state, according to the active-level designation in the name.

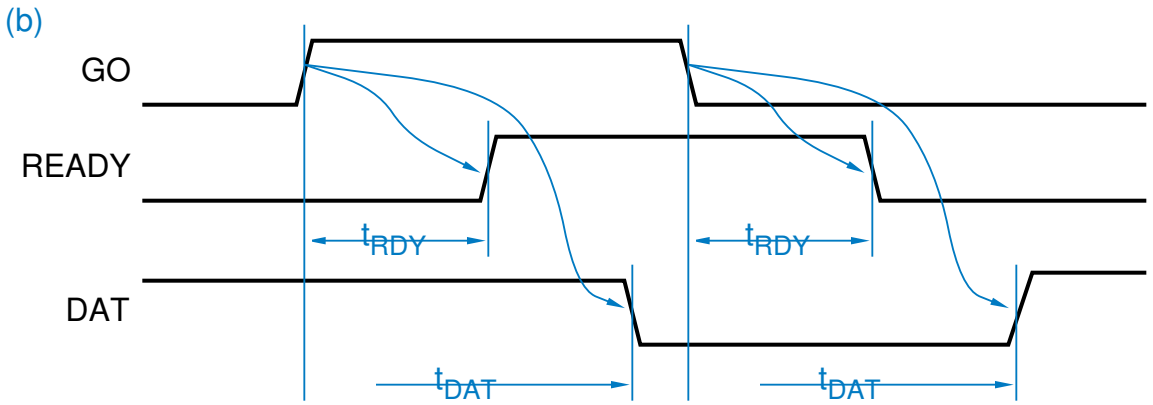
Active Low	*Active High*
READY–	READY+
ERROR.L	ERROR.H
ADDR15(L)	ADDR15(H)
RESET*	RESET
ENABLE~	ENABLE
~GO	GO
/RECEIVE	RECEIVE
TRANSMIT_L	TRANSMIT

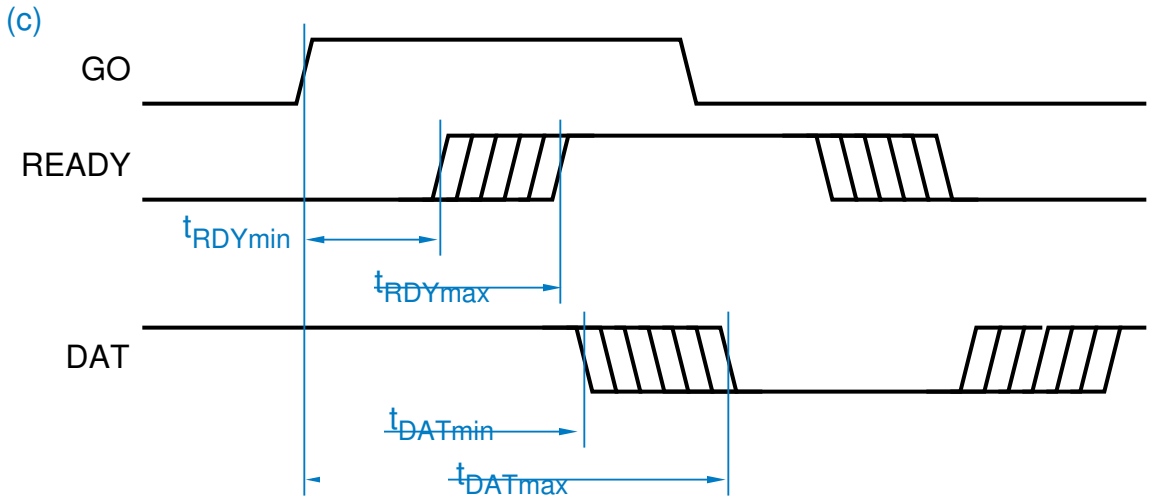
Example

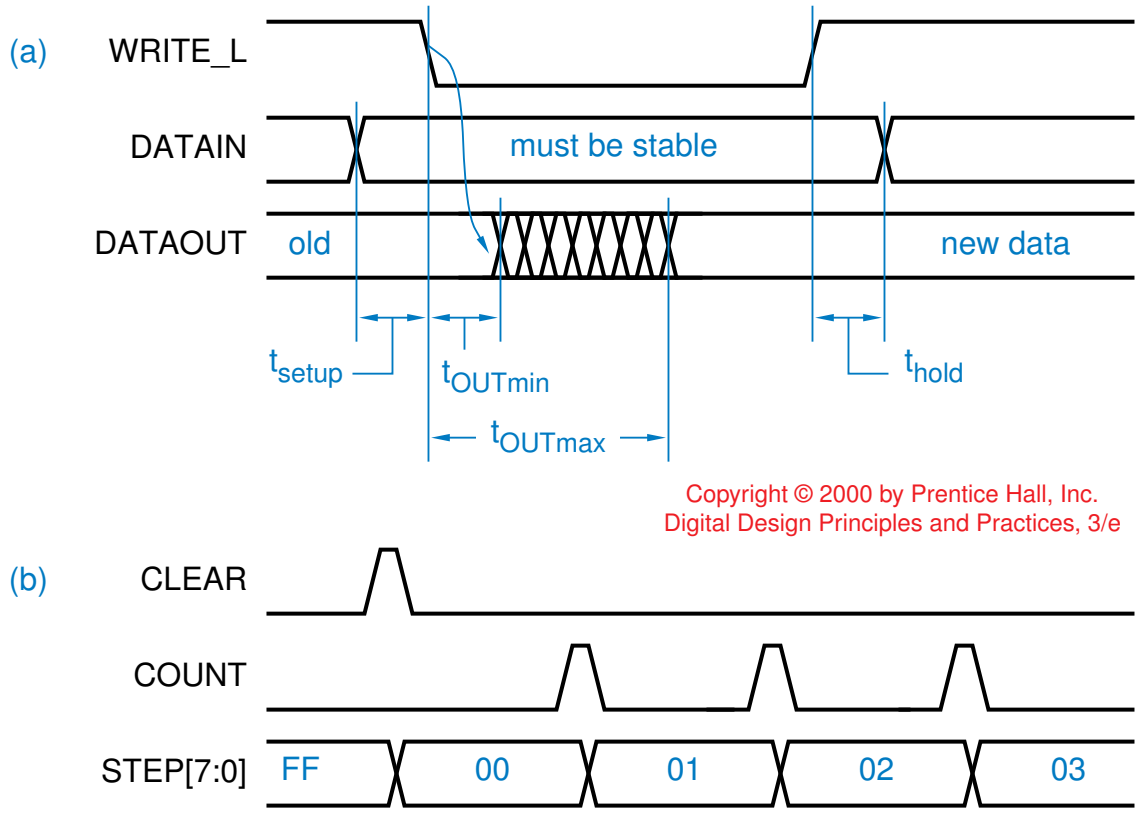

Logic
Circuit
HIGH when error occurs
ERROR
OK_L
Logic
Circuit
LOW when error occurs
ERROR_L
ERROR
ERROR1_L











Example – Timing Diagram

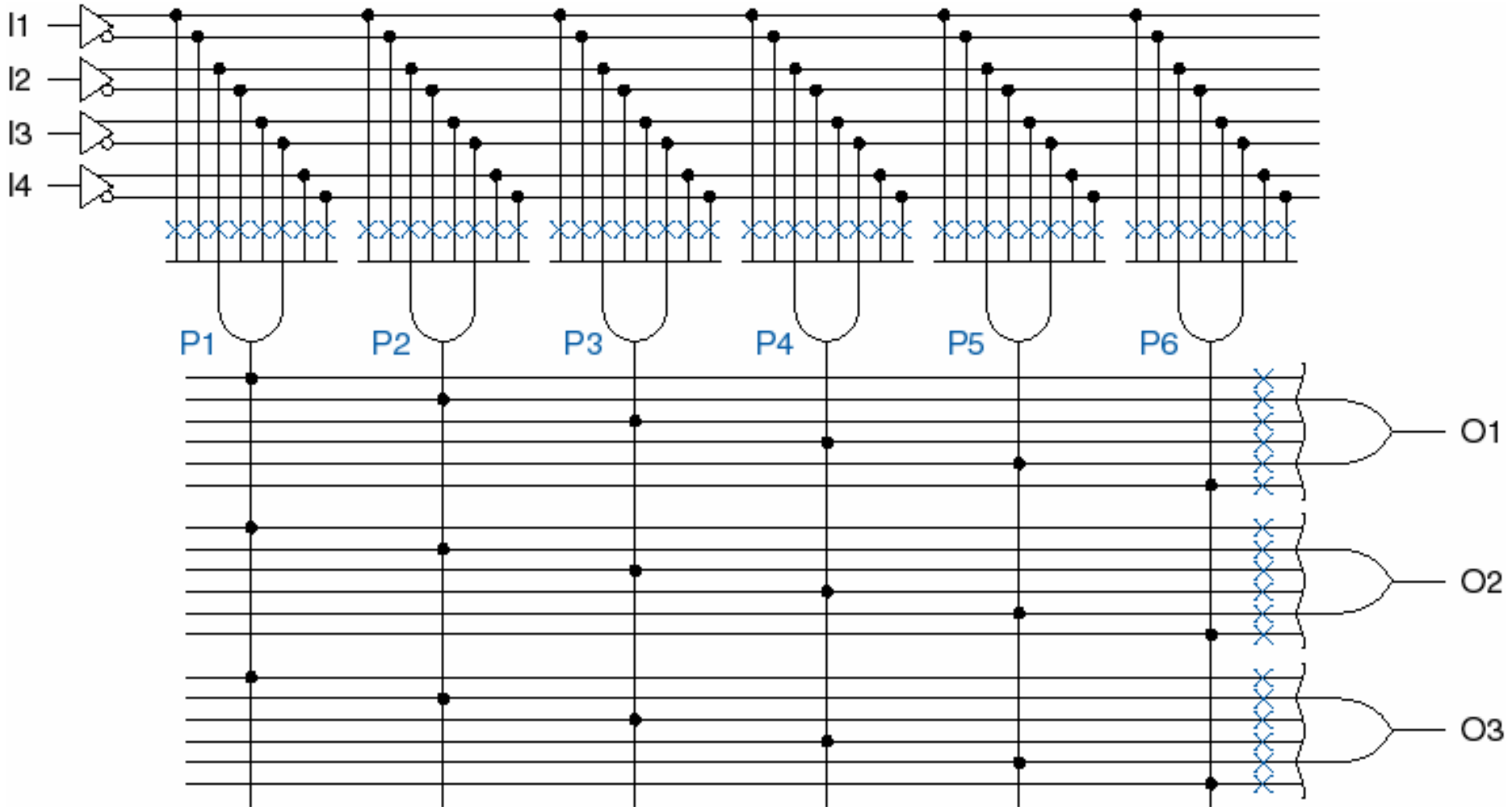
Programmable Logic Arrays (PLAs)

Any combinational logic function can be realized as a sum of products.

Idea: Build a large AND-OR array with lots of inputs and product terms, and programmable connections.

- *n* inputs
  - AND gates have 2*n* inputs -- true and complement of each variable.
- *m* outputs, driven by large OR gates
  - Each AND gate is programmably connected to each output's OR gate.
- *p* AND gates ($p << 2^n$)

Example: 4x3 PLA, 6 product terms



Programmable Array Logic (PALs)

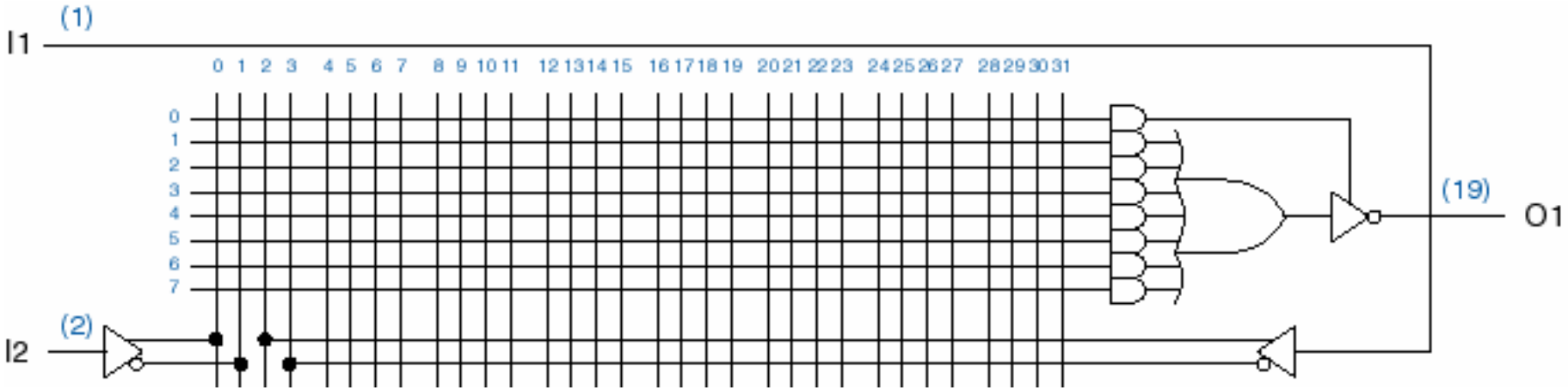
How beneficial is product sharing?

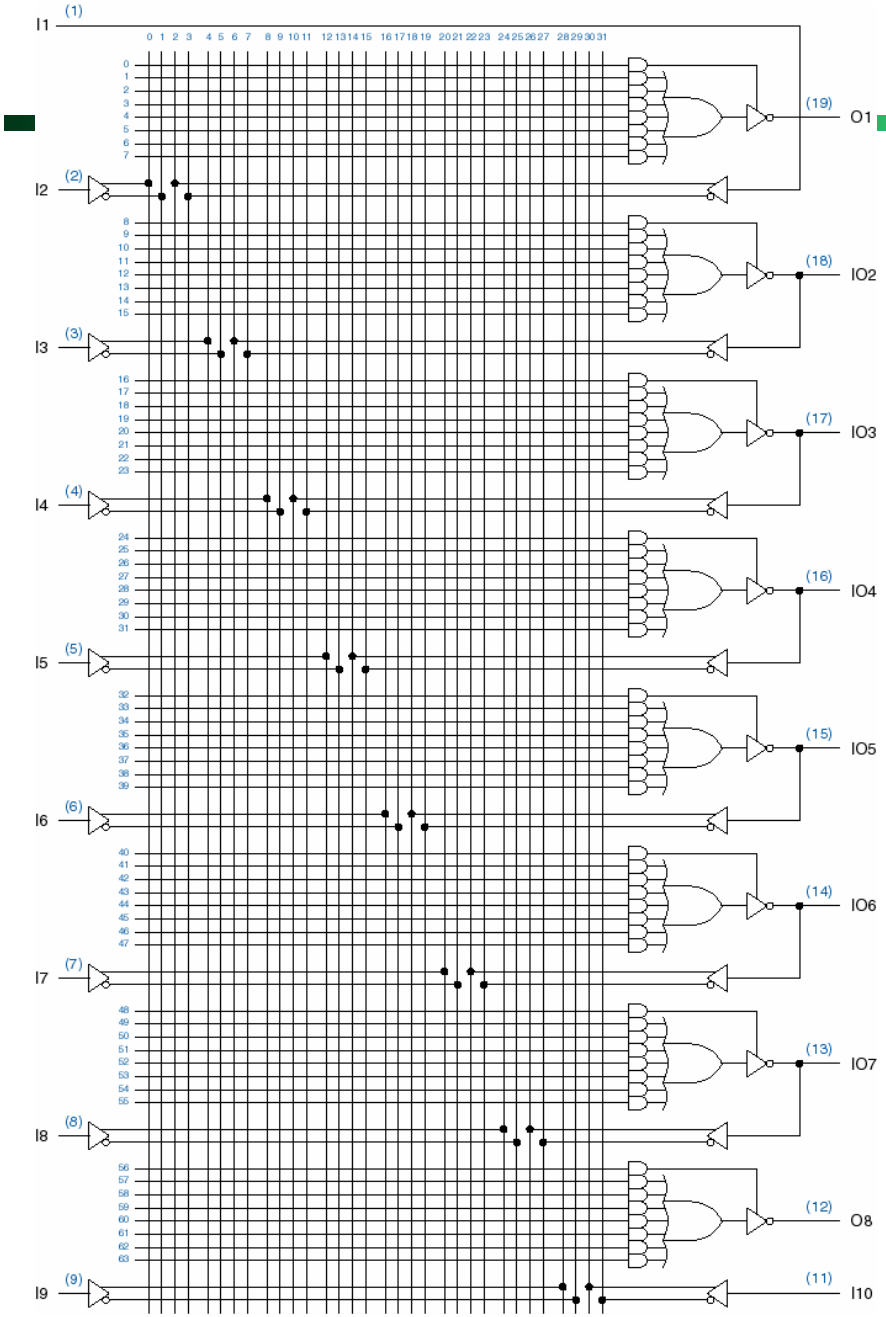
- Not enough to justify the extra AND array

PALs ==> *fixed* OR array

- Each AND gate is permanently connected to a certain OR gate.

Example: PAL16L8





- 10 primary inputs
- 8 outputs, with 7 ANDs per output
- 1 AND for 3-state enable
- 6 outputs available as inputs
  - more inputs, at expense of outputs
  - two-pass logic, helper terms
- Note inversion on outputs
  - output is complement of sum-of-products
  - newer PALs have selectable inversion

Designing with PALs

Compare number of inputs and outputs of the problem with available resources in the PAL.

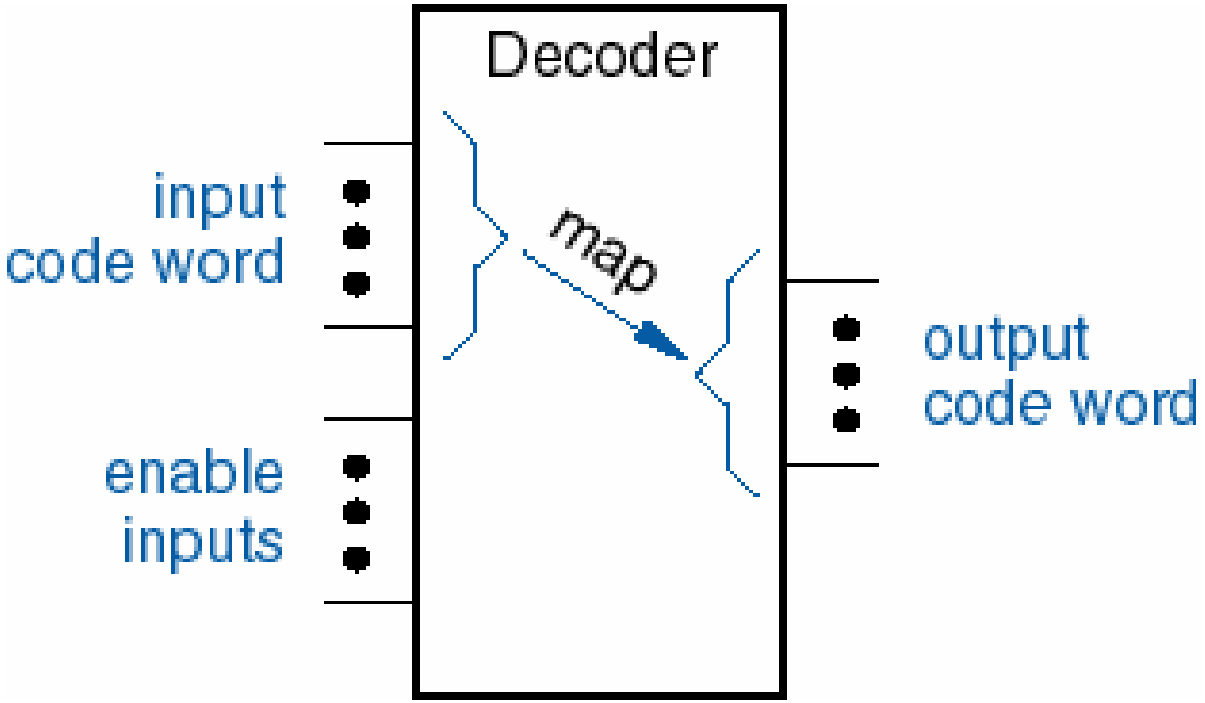
Write equations for each output using HDL.

Compile the HDL program, determine whether minimized equations fit in the available AND terms.

If no fit, try modifying equations.

Decoders

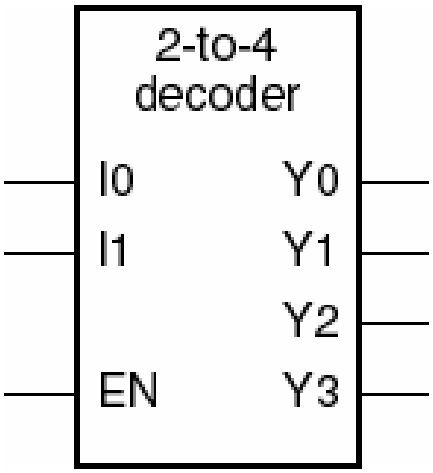
General decoder structure



Typically n inputs, 2^n outputs

– 2-to-4, 3-to-8, 4-to-16, etc.

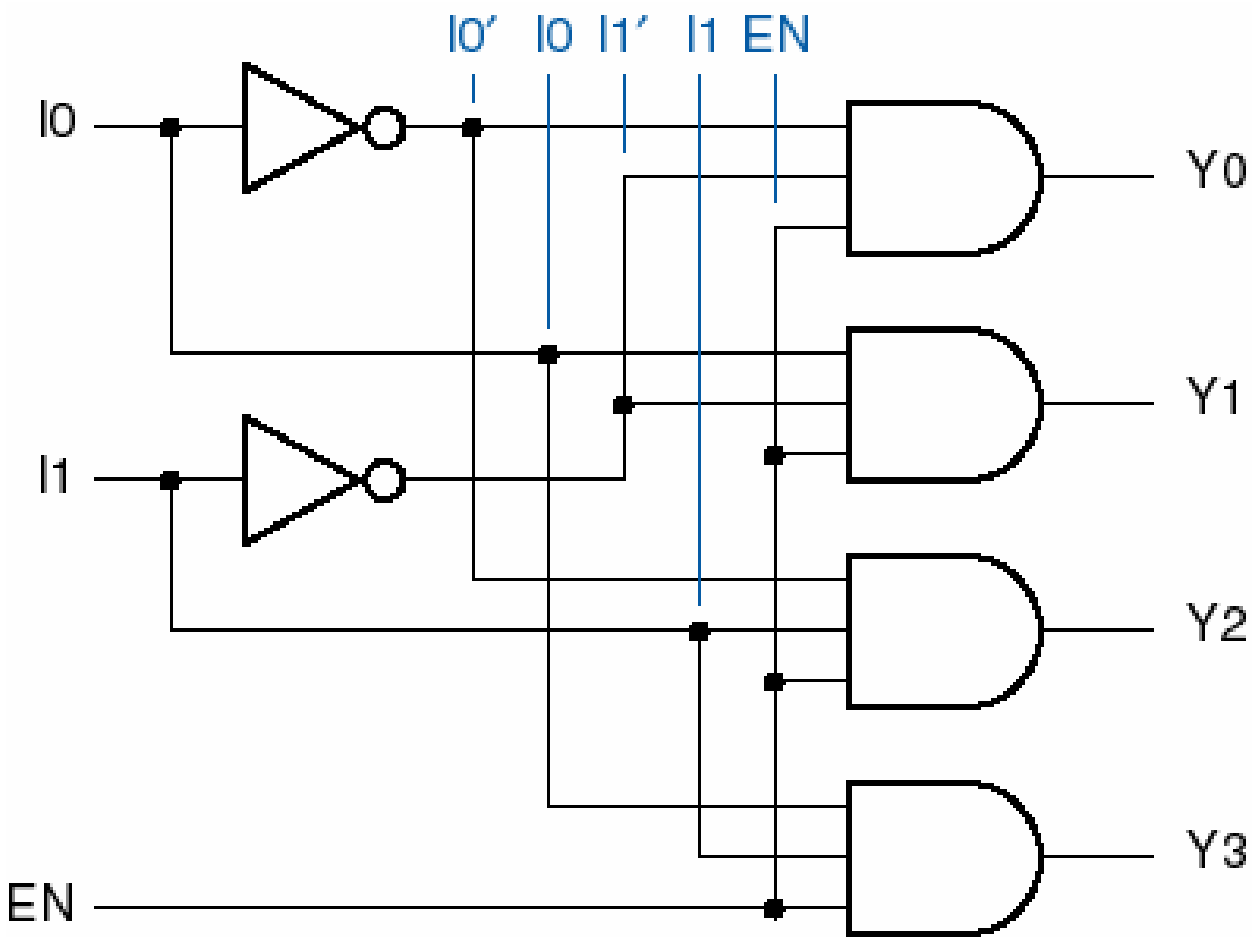
Binary 2-to-4 decoder



Inputs			*Outputs*			
EN	I1	I0	Y3	Y2	Y1	Y0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Note “x” (don’t care) notation.

2-to-4-decoder logic diagram



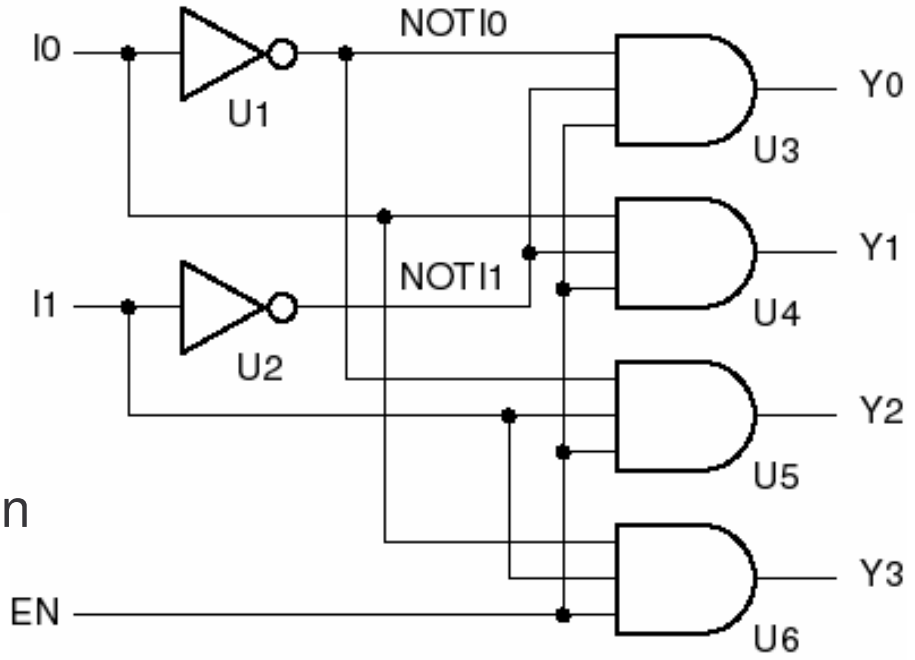
Example: 2-to-4 decoder

Architecture

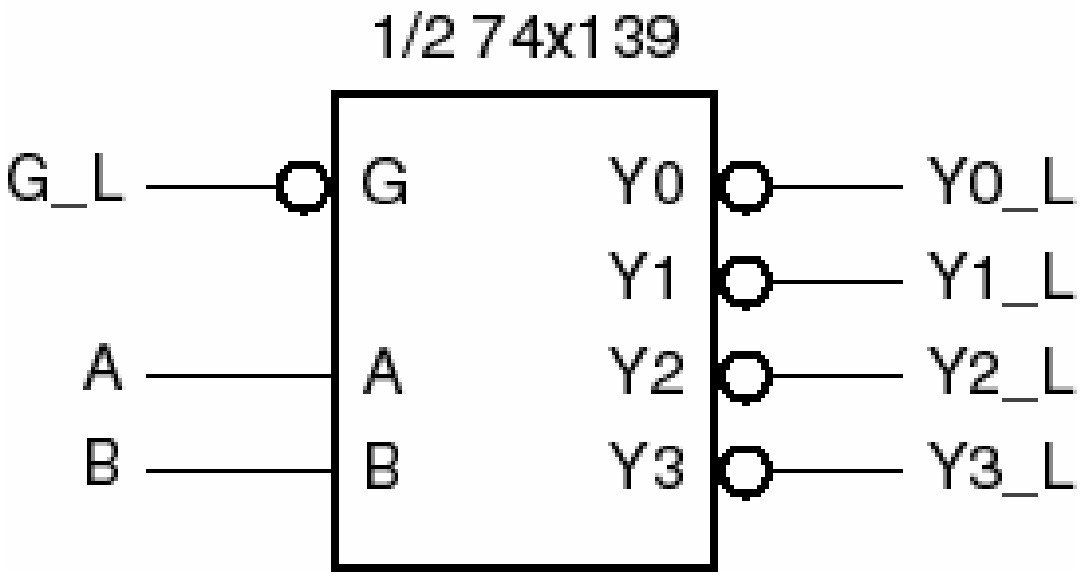
```
architecture V2to4dec_s of V2to4dec is
  signal NOTI0, NOTI1: STD_LOGIC;
  component inv port (I: in STD_LOGIC; O: out STD_LOGIC ); end component;
  component and3 port (I0, I1, I2: in STD_LOGIC; O: out STD_LOGIC ); end component;
begin
  U1: inv port map (I0,NOTI0);
  U2: inv port map (I1,NOTI1);
  U3: and3 port map (NOTI0,NOTI1,EN,Y0);
  U4: and3 port map (   I0,NOTI1,EN,Y1);
  U5: and3 port map (NOTI0,   I1,EN,Y2);
  U6: and3 port map (   I0,   I1,EN,Y3);
end V2to4dec_s;
```

built-in library components

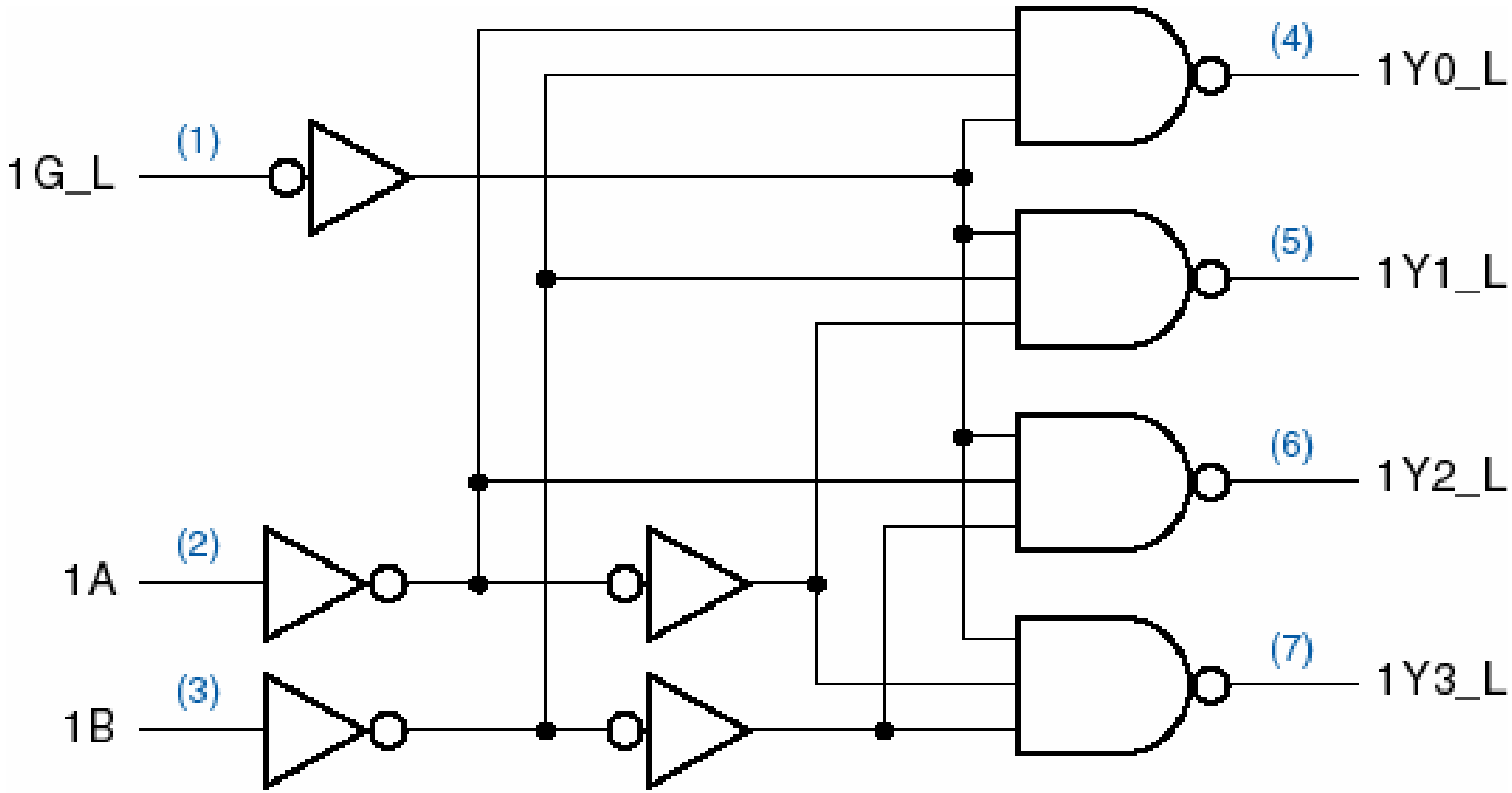
positional correspondence with entity definition



Decoder Symbol



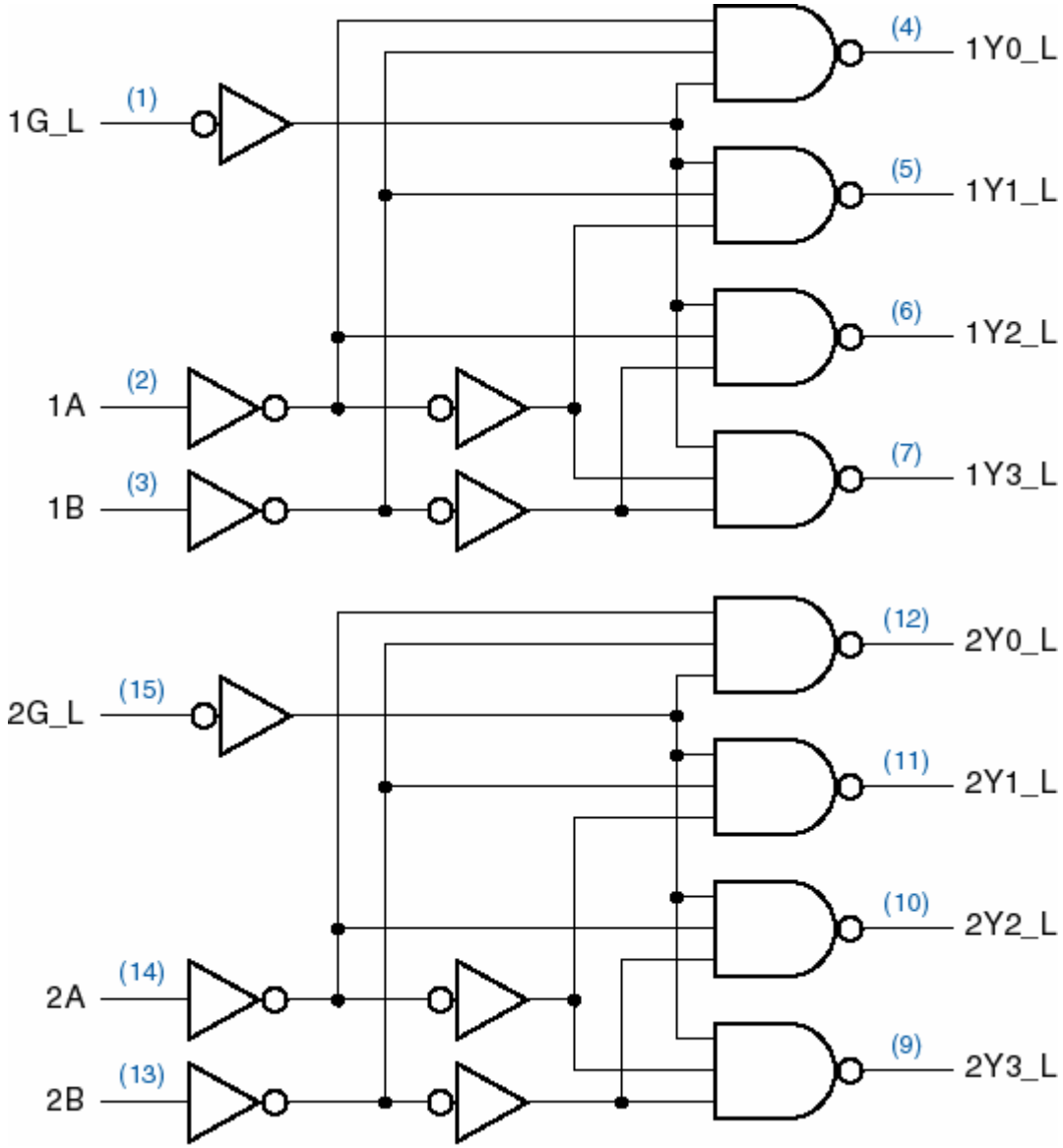
MSI 2-to-4 decoder

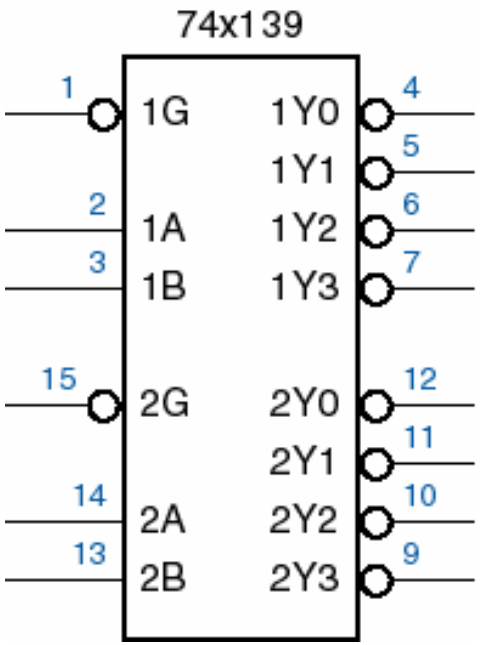


Input buffering (less load)

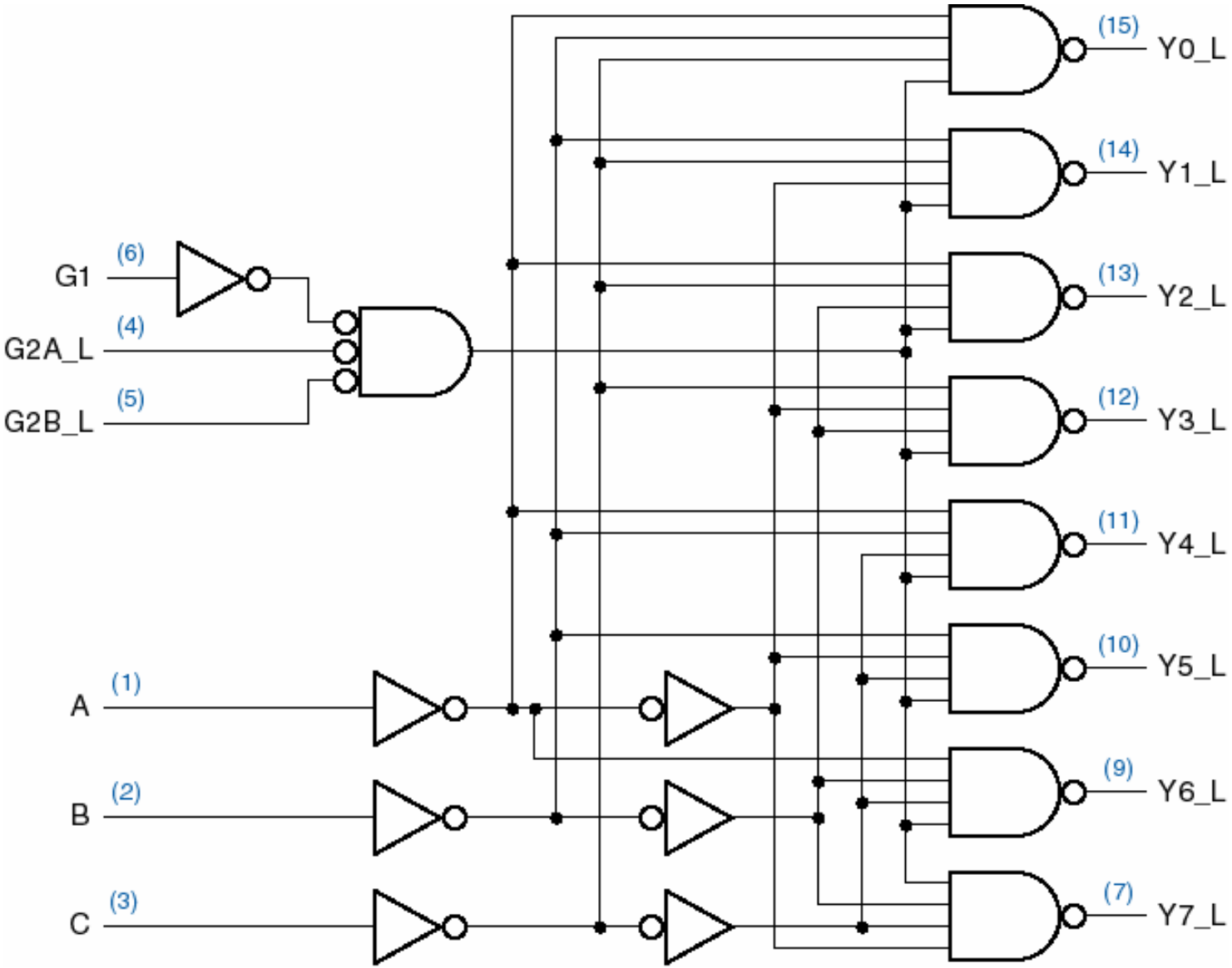
NAND gates (faster)

Complete 74x139 Decoder

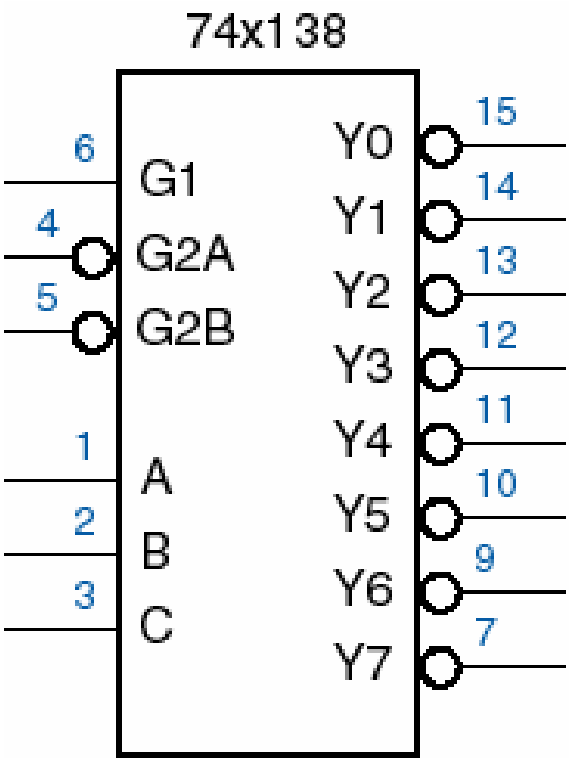




3-to-8 decoder



74x138 3-to-8-decoder symbol



Dataflow-style program for 3-to-8 decoder

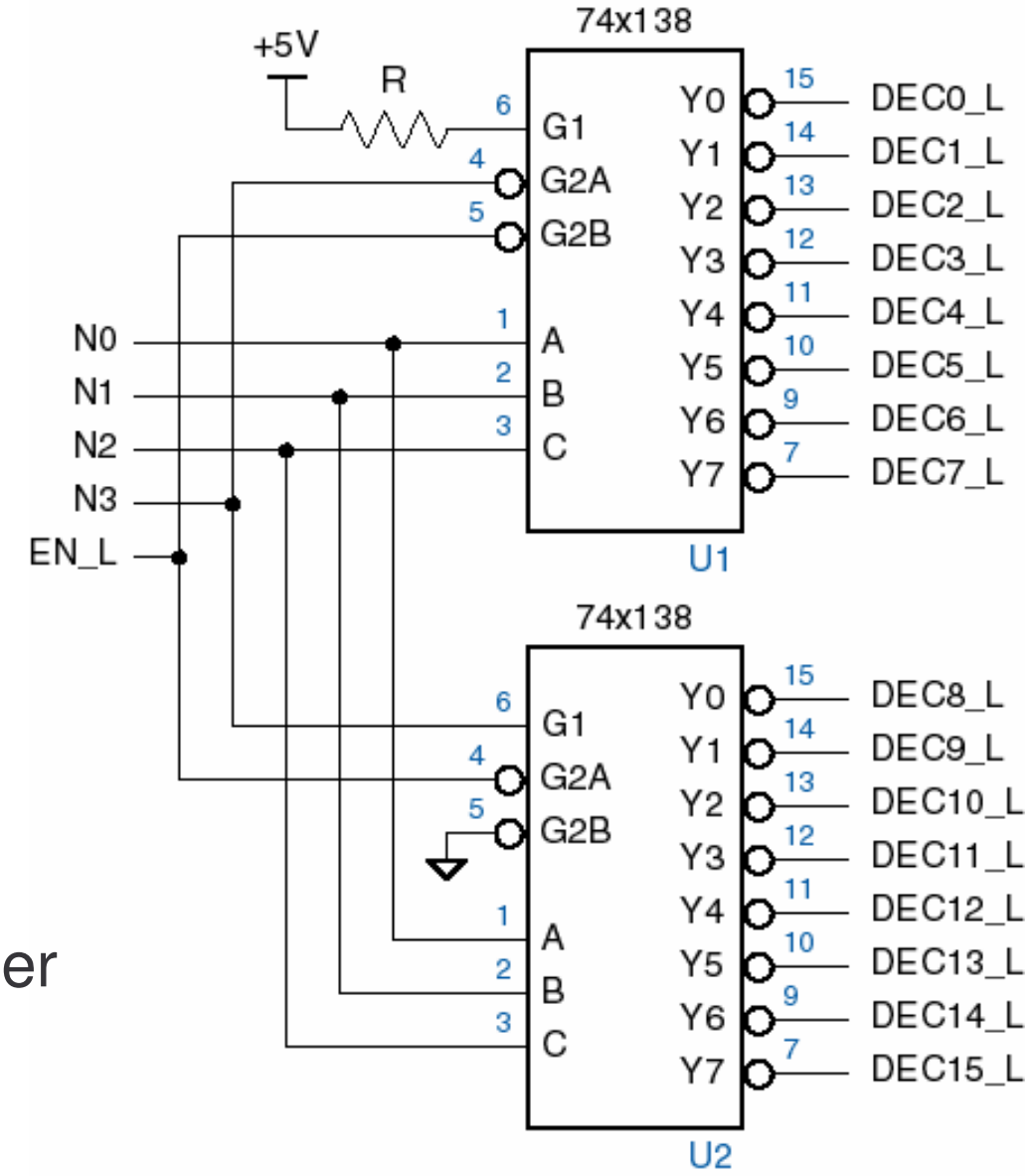
```
library IEEE;
use IEEE.std_logic_1164.all;

entity V74x138 is
    port (G1, G2A_L, G2B_L: in STD_LOGIC;       -- enable inputs
          A: in STD_LOGIC_VECTOR (2 downto 0);   -- select inputs
          Y_L: out STD_LOGIC_VECTOR (0 to 7) );  -- decoded outputs
 end V74x138;
```

Dataflow-style program for 3-to-8 decoder

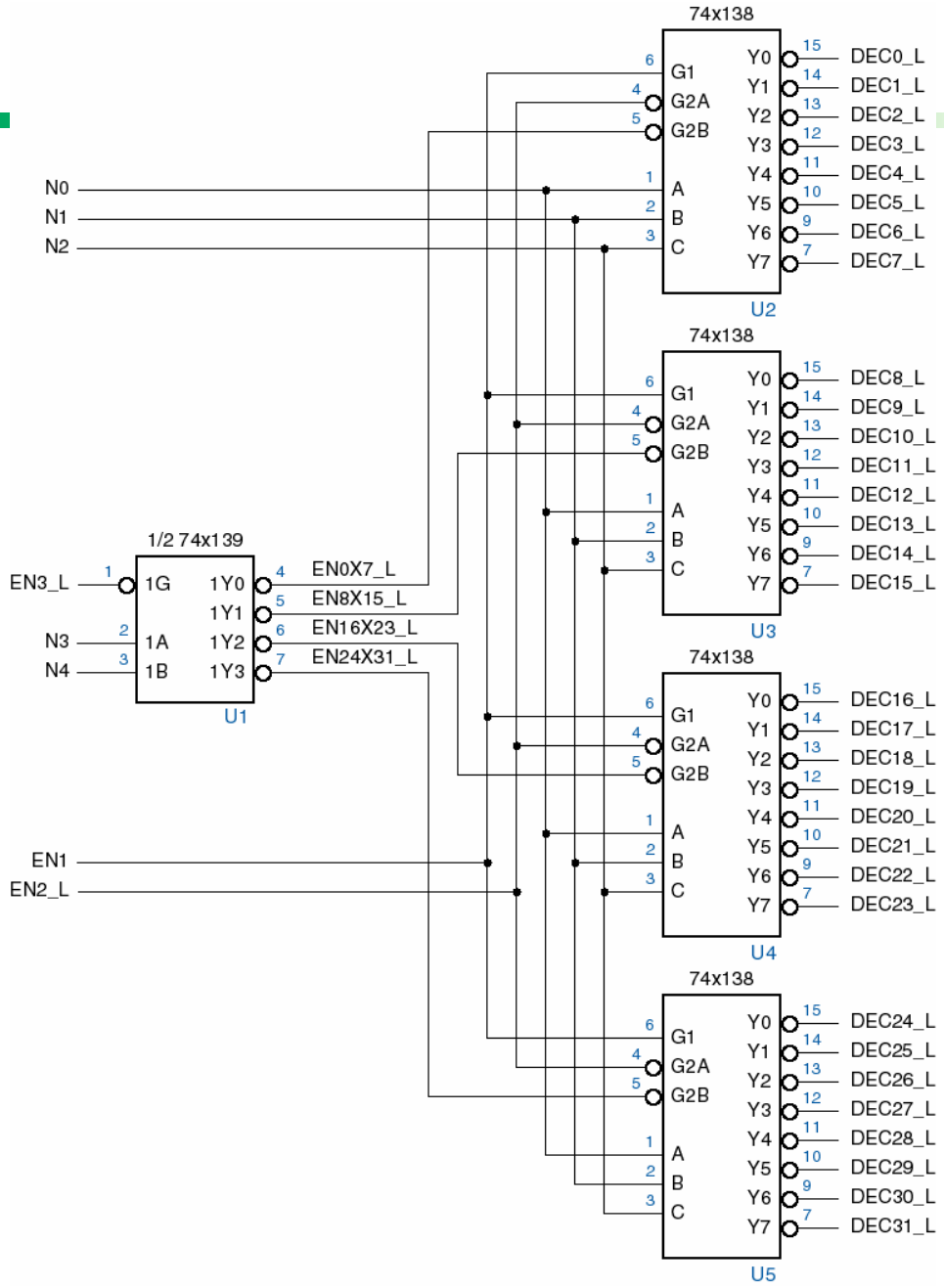
```
architecture V74x138_b of V74x138 is
  signal G2A, G2B: STD_LOGIC;              -- active-high version of inputs
  signal Y: STD_LOGIC_VECTOR (0 to 7);     -- active-high version of outputs
  signal Y_s: STD_LOGIC_VECTOR (0 to 7);   -- internal signal
begin
  G2A <= not G2A_L; -- convert inputs
  G2B <= not G2B_L; -- convert inputs
  Y_L <= Y;         -- convert outputs
  with A select Y_s <=
    "10000000" when "000",
    "01000000" when "001",
    "00100000" when "010",
    "00010000" when "011",
    "00001000" when "100",
    "00000100" when "101",
    "00000010" when "110",
    "00000001" when "111",
    "00000000" when others;
  Y <= not Y_s when (G1 and G2A and G2B)='1' else "00000000";
end V74x138_b;
```

Decoder cascading



4-to-16 decoder

More cascading



5-to-32 decoder

Decoder applications

Microprocessor memory systems

- selecting different banks of memory

Microprocessor input/output systems

- selecting different devices

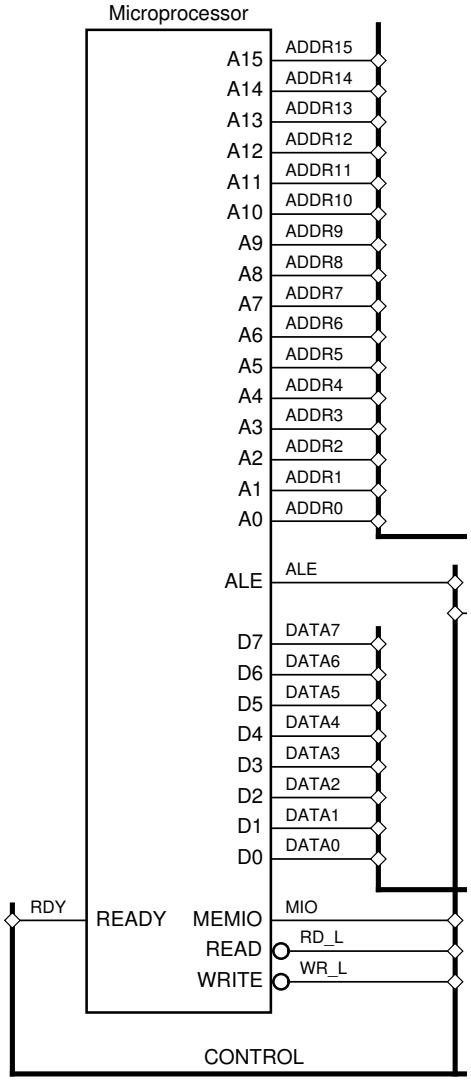
Microprocessor instruction decoding

- enabling different functional units

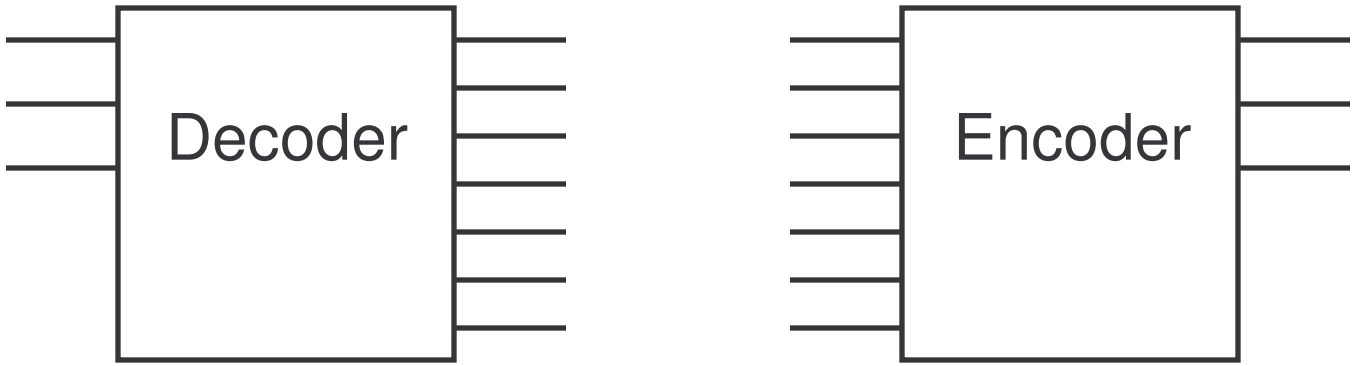
Memory chips

- enabling different rows of memory depending on address

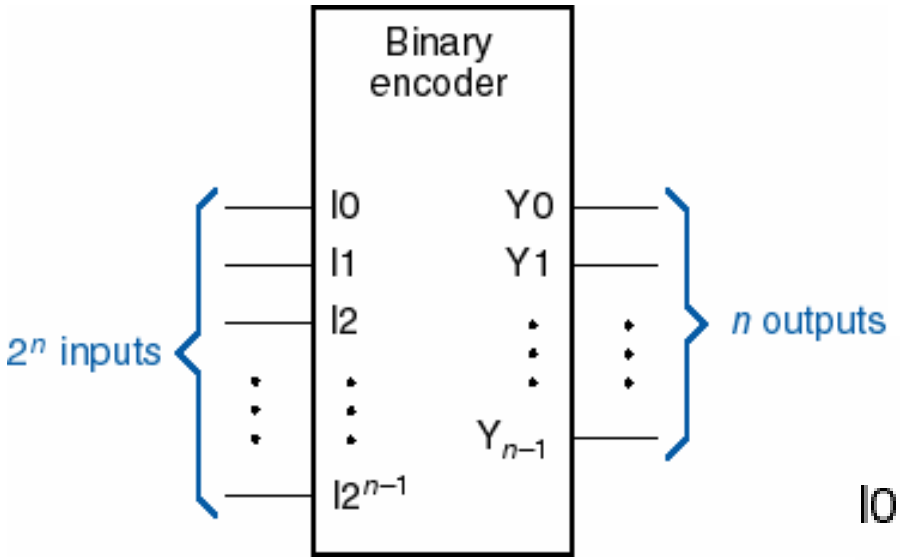
Example – Microprocessor Application

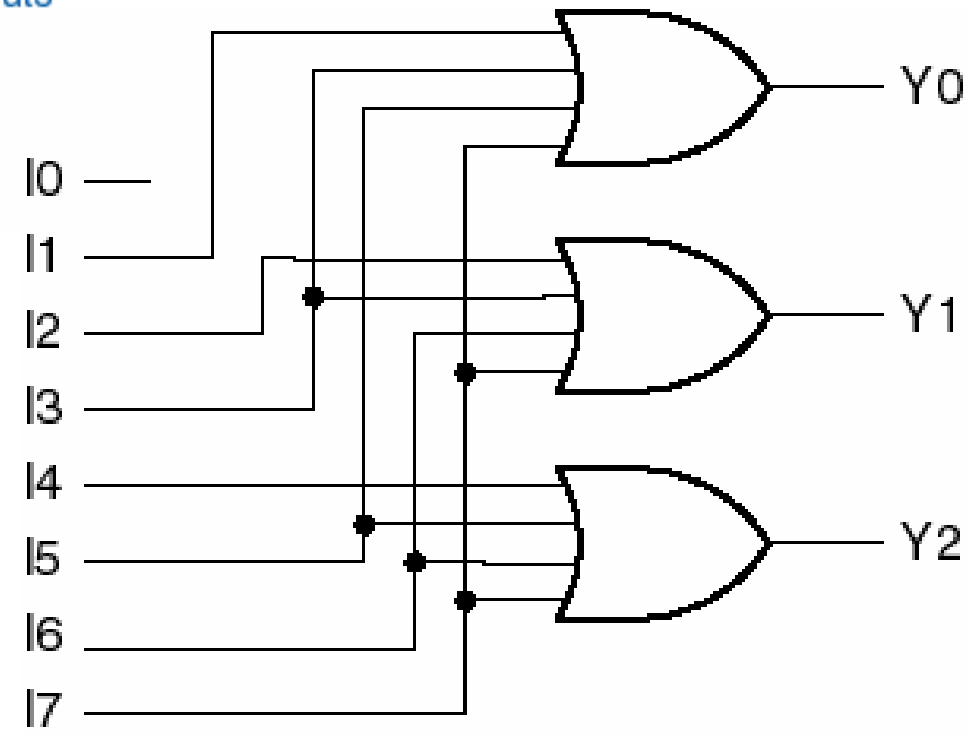


Encoders vs. Decoders

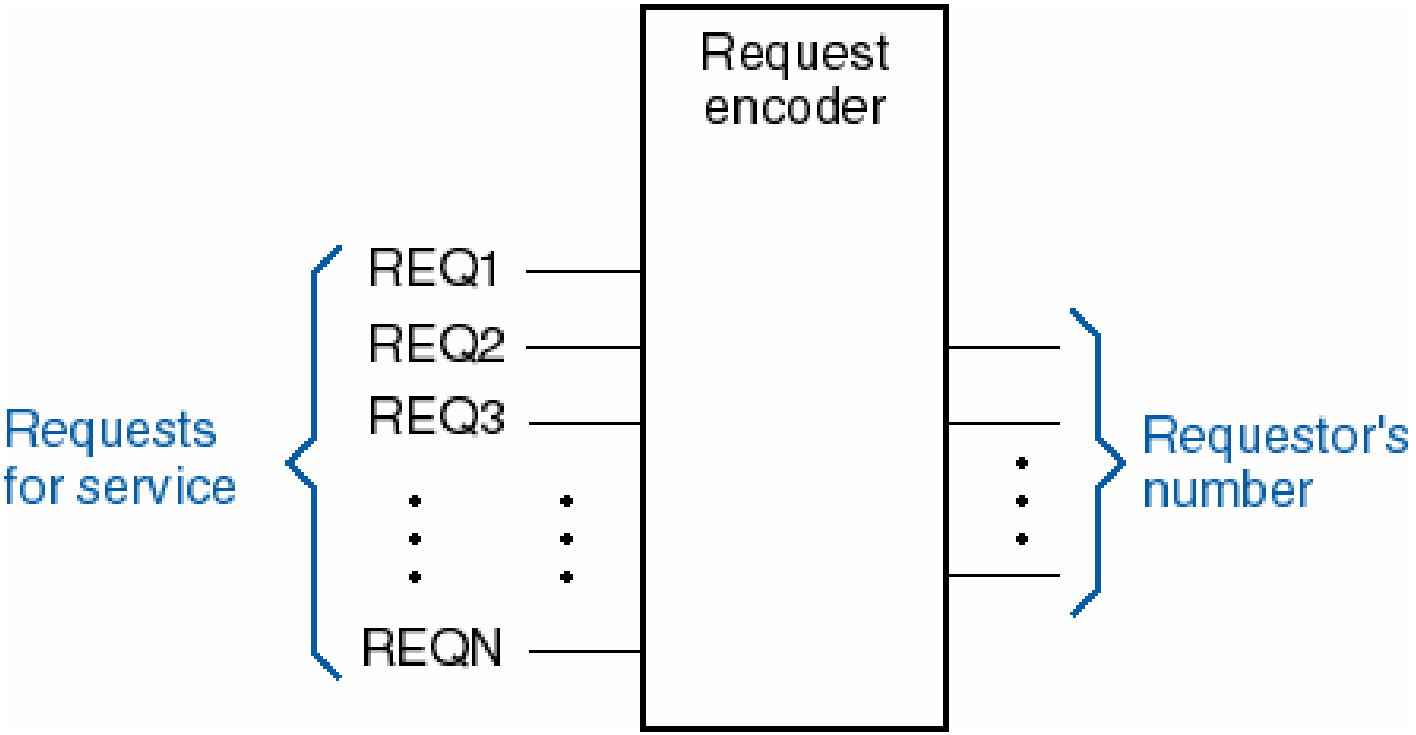


Binary encoders

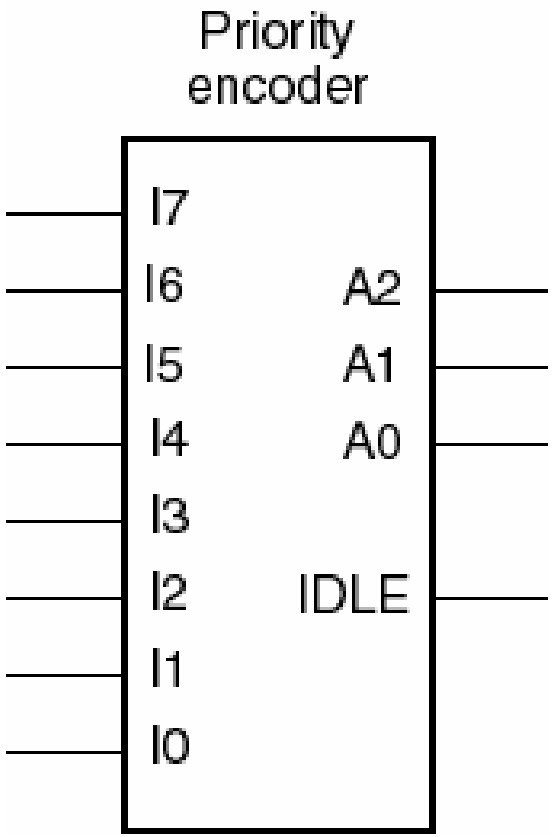




Need priority in most applications



8-input priority encoder



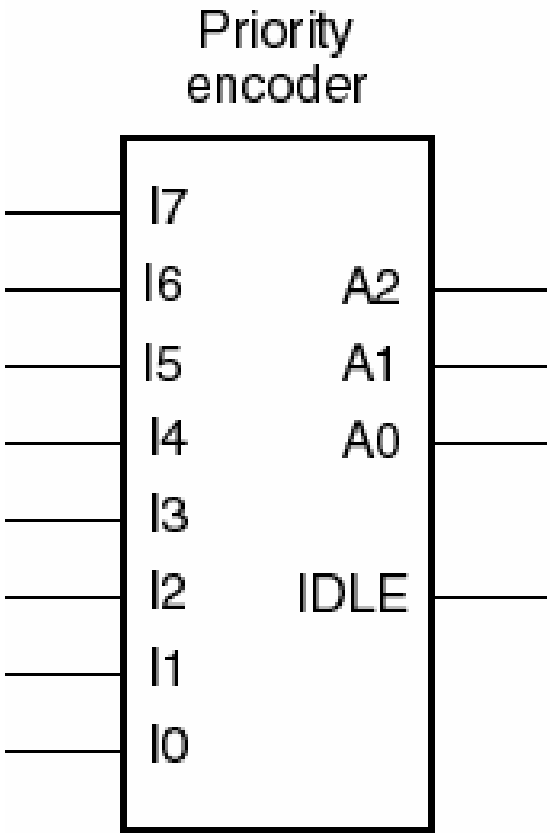
Priority-encoder logic equations

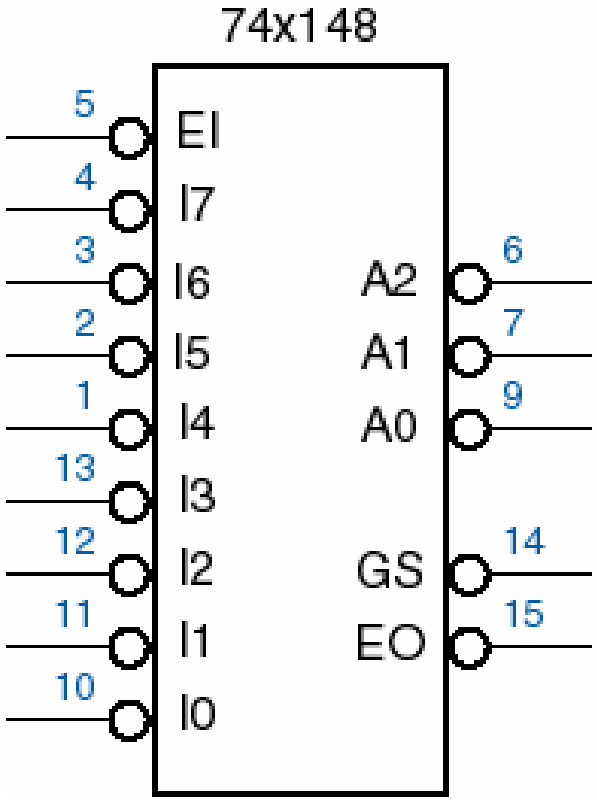
$$H7 = I7$$
$$H6 = I6 \cdot I7'$$
$$H5 = I5 \cdot I6' \cdot I7'$$
$$\cdots$$
$$H0 = I0 \cdot I1' \cdot I2' \cdot I3' \cdot I4' \cdot I5' \cdot I6' \cdot I7'$$

$$A2 = H4 + H5 + H6 + H7$$
$$A1 = H2 + H3 + H6 + H7$$
$$A0 = H1 + H3 + H5 + H7$$

$$IDLE = (I0 + I1 + I2 + I3 + I4 + I5 + I6 + I7)'$$
$$= I0' \cdot I1' \cdot I2' \cdot I3' \cdot I4' \cdot I5' \cdot I6' \cdot I7'$$

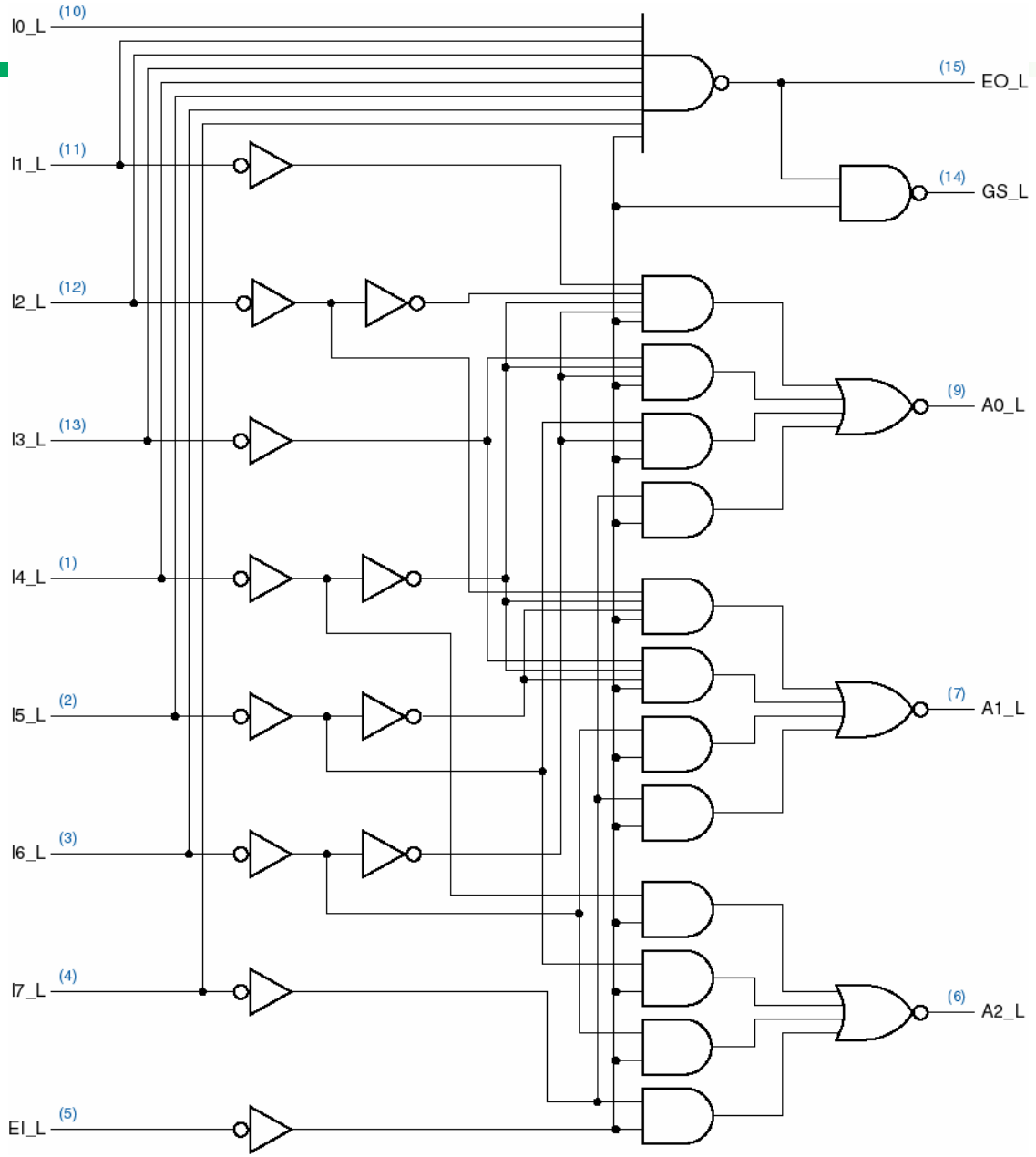
74x148 8-input priority encoder





- Active-low I/O
- Enable Input
- "Got Something"
- Enable Output

74x148 circuit

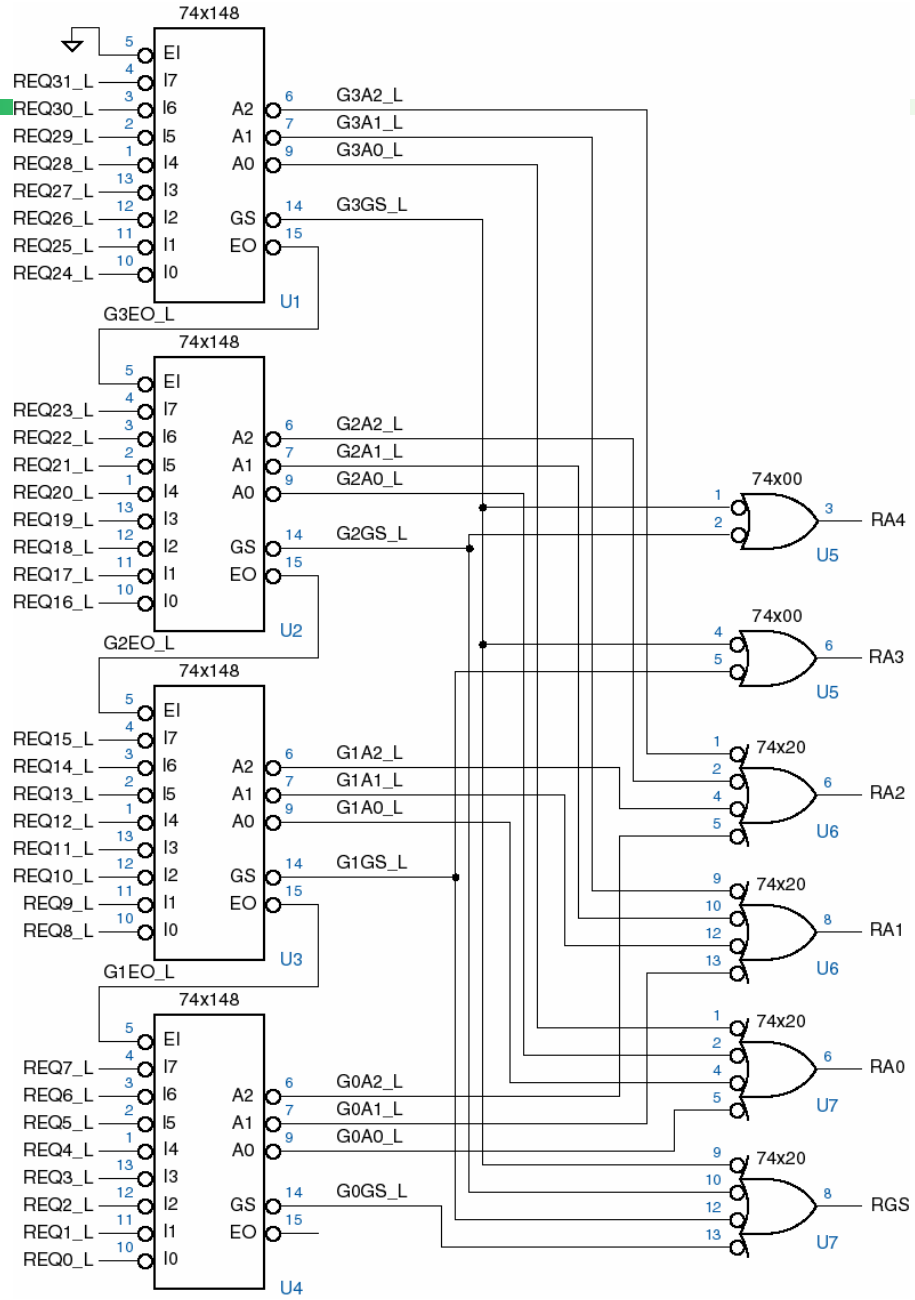


74x148 Truth Table

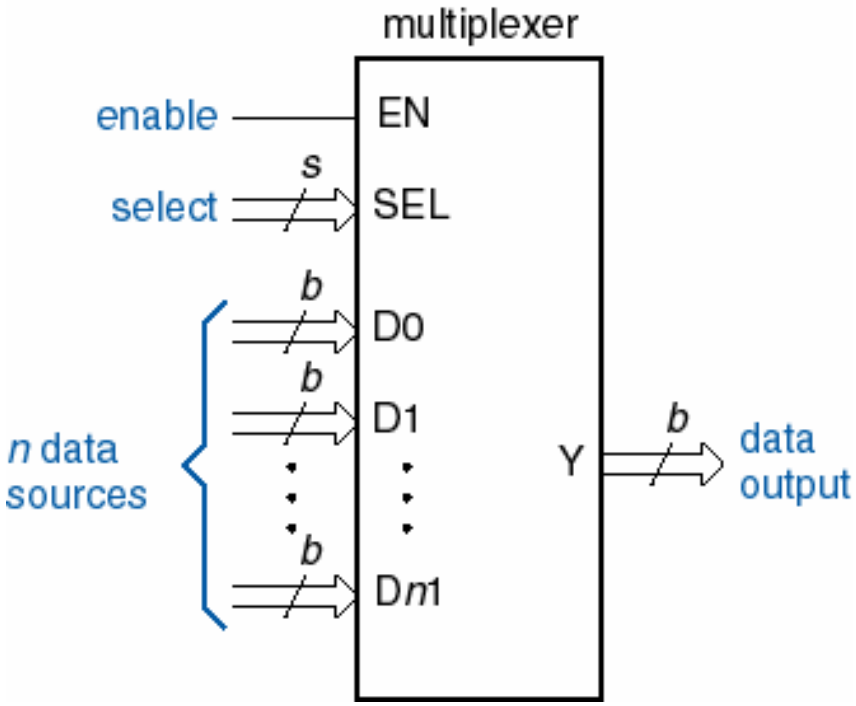
Inputs									*Outputs*				
EI_L	I0_L	I1_L	I2_L	I3_L	I4_L	I5_L	I6_L	I7_L	A2_L	A1_L	A0_L	GS_L	EO_L
1	x	x	x	x	x	x	x	x	1	1	1	1	1
0	x	x	x	x	x	x	x	0	0	0	0	0	1
0	x	x	x	x	x	x	0	1	0	0	1	0	1
0	x	x	x	x	x	0	1	1	0	1	0	0	1
0	x	x	x	x	0	1	1	1	0	1	1	0	1
0	x	x	x	0	1	1	1	1	1	0	0	0	1
0	x	x	0	1	1	1	1	1	1	0	1	0	1
0	x	0	1	1	1	1	1	1	1	1	0	0	1
0	0	1	1	1	1	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0

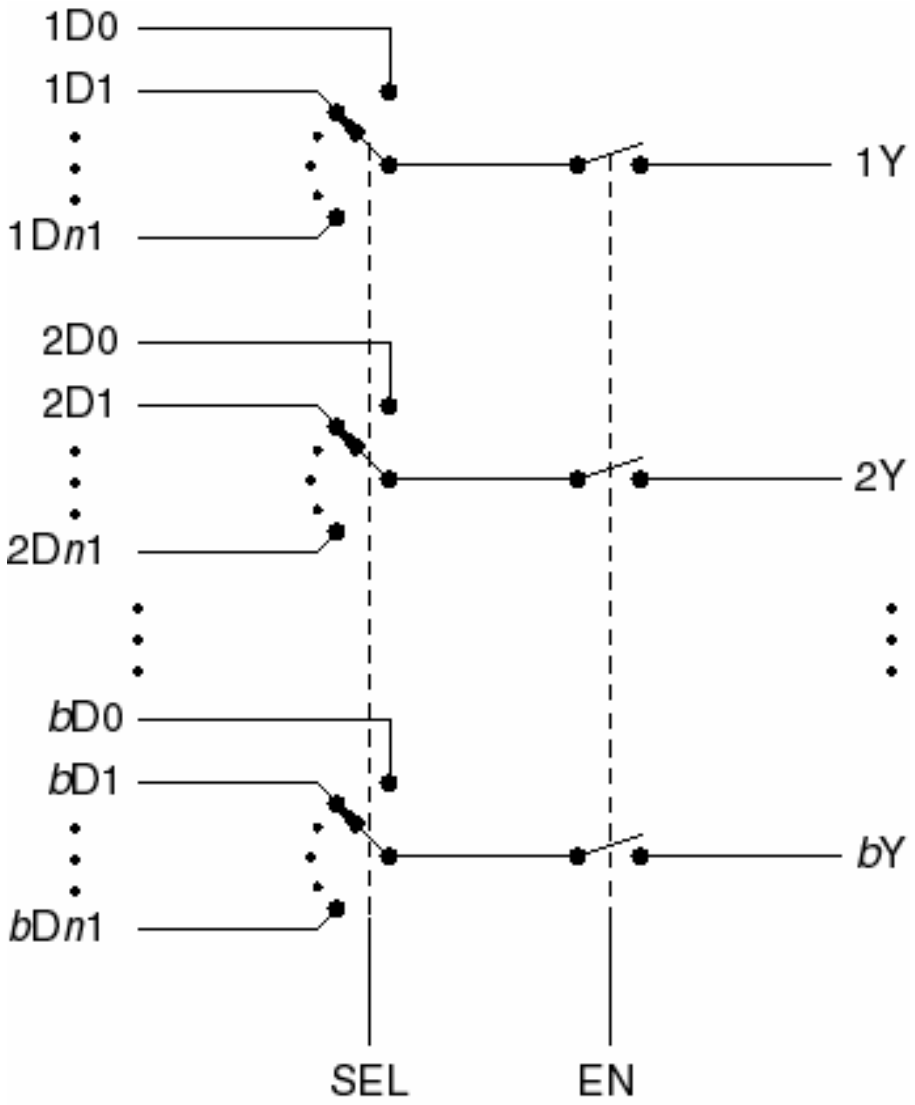
Cascading priority encoders

32-input
priority encoder



Multiplexers





Multiplexer - Gate-Level Modeling - Verilog

2-to-1 Multiplexer

```
// 2-to-1 Multiplexer module
module mux_2 (out, i0, i1, sel);      // header
input i0, i1, sel;                    // input & output ports
output out;
wire x1, x2, x3;                      // internal nets
or (out, x2, x3);                     // form output
and (x2, i0, x1);                     // i0 • sel'
and (x3, i1, sel);                    // i1 • sel
not (x1, sel);                        // invert sel
endmodule
```

Multiplexer - Dataflow Modeling - Verilog

4-bit Multiplexer

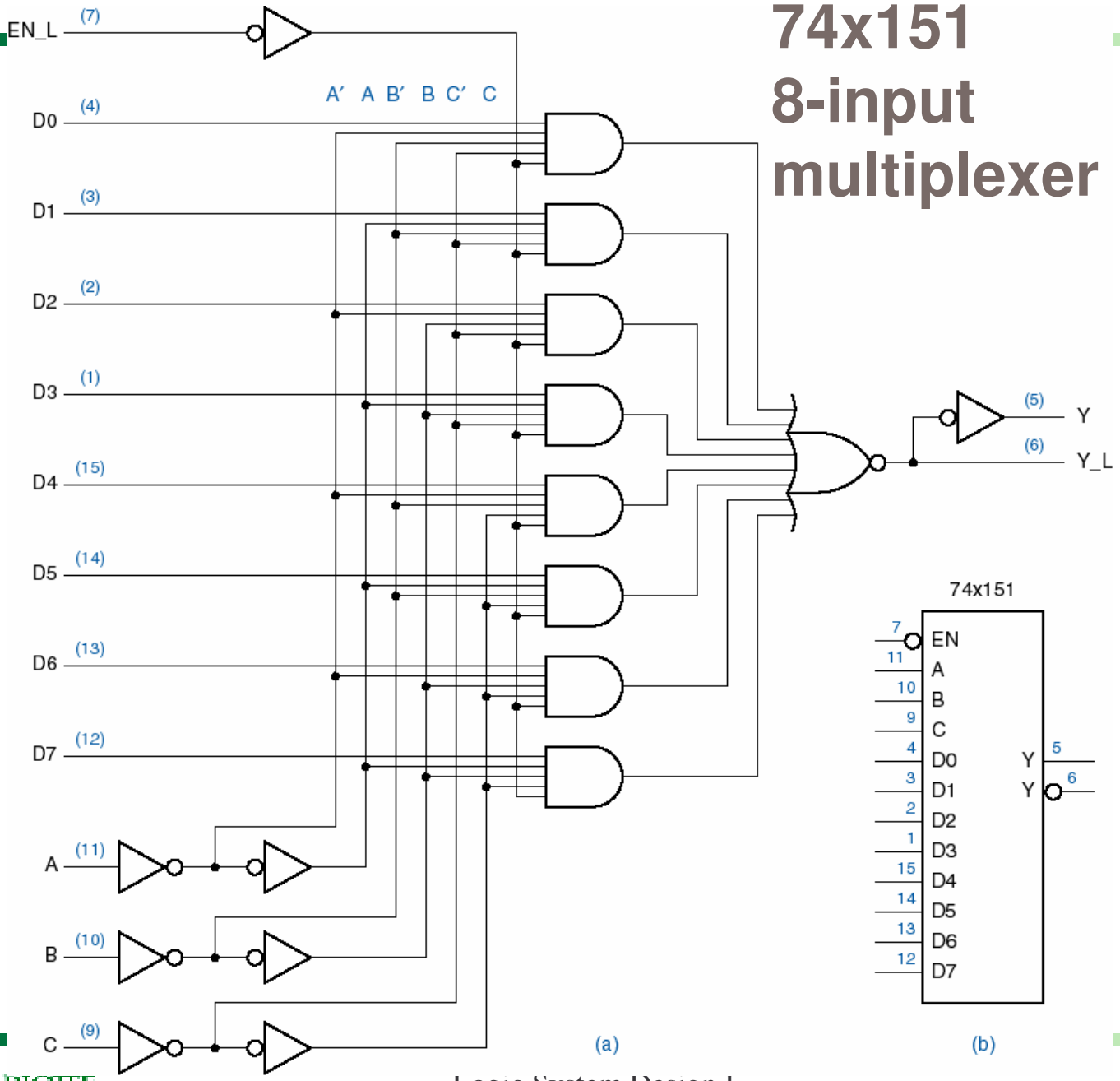
```
// Four-bit 2-to-1 multiplexer
module mux_4bit (Out, A, B, sel);
input [3:0] A, B;
input sel;
output [3:0] Out;
assign Out = sel ? B, A;
endmodule
```

Multiplexer - Behavioral Modeling - Verilog

Conditional Statements

```
module mux4_to_1 (A, B, C, D, OUT, select);
input [7:0] A, B, C, D;
input [1:0] select;
output [7:0] OUT;
reg [7:0] OUT;
always @ (A or B or C or D or select)
case (select)
  2'd0: OUT = A;
  2'd1: OUT = B;
  2'd2: OUT = C;
  2'd3: OUT = D;
endcase
end
```

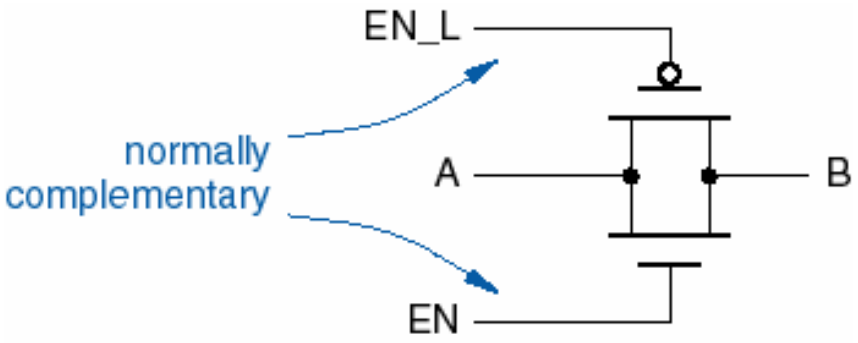
74x151 8-input multiplexer

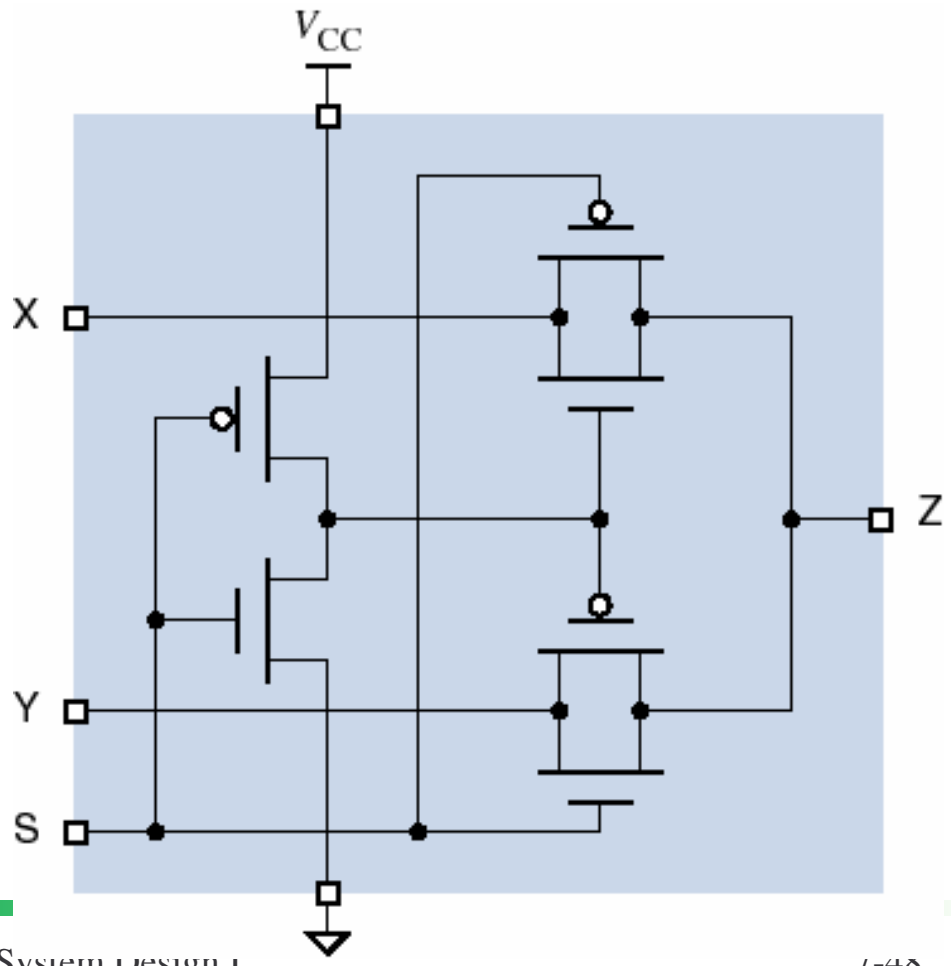


74x151 truth table

Inputs				Outputs	
EN_L	C	B	A	Y	Y_L
1	x	x	x	0	1
0	0	0	0	D0	D0′
0	0	0	1	D1	D1′
0	0	1	0	D2	D2′
0	0	1	1	D3	D3′
0	1	0	0	D4	D4′
0	1	0	1	D5	D5′
0	1	1	0	D6	D6′
0	1	1	1	D7	D7′

CMOS transmission gates





Other multiplexer varieties

2-input, 4-bit-wide
- 74x157

Inputs		Outputs			
G_L	S	1Y	2Y	3Y	4Y
1	x	0	0	0	0
0	0	1A	2A	3A	4A
0	1	1B	2B	3B	4B

4-input, 2-bit-wide
- 74x153

Barrel shifter design example

n data inputs, *n* data outputs

Control inputs specify number of positions to rotate or shift data inputs

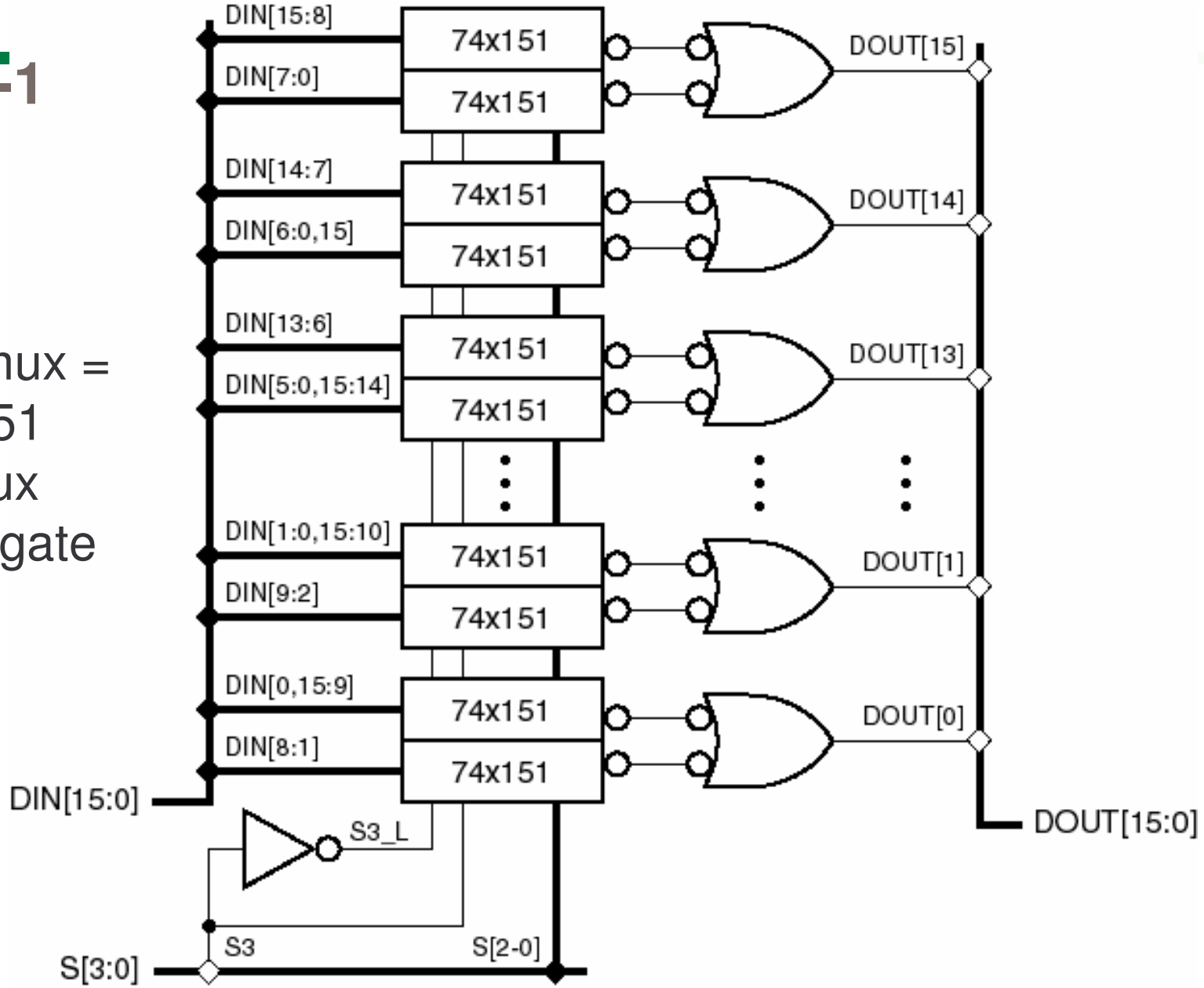
Example: n = 16

- DIN[15:0], DOUT[15:0], S[3:0] (shift amount)

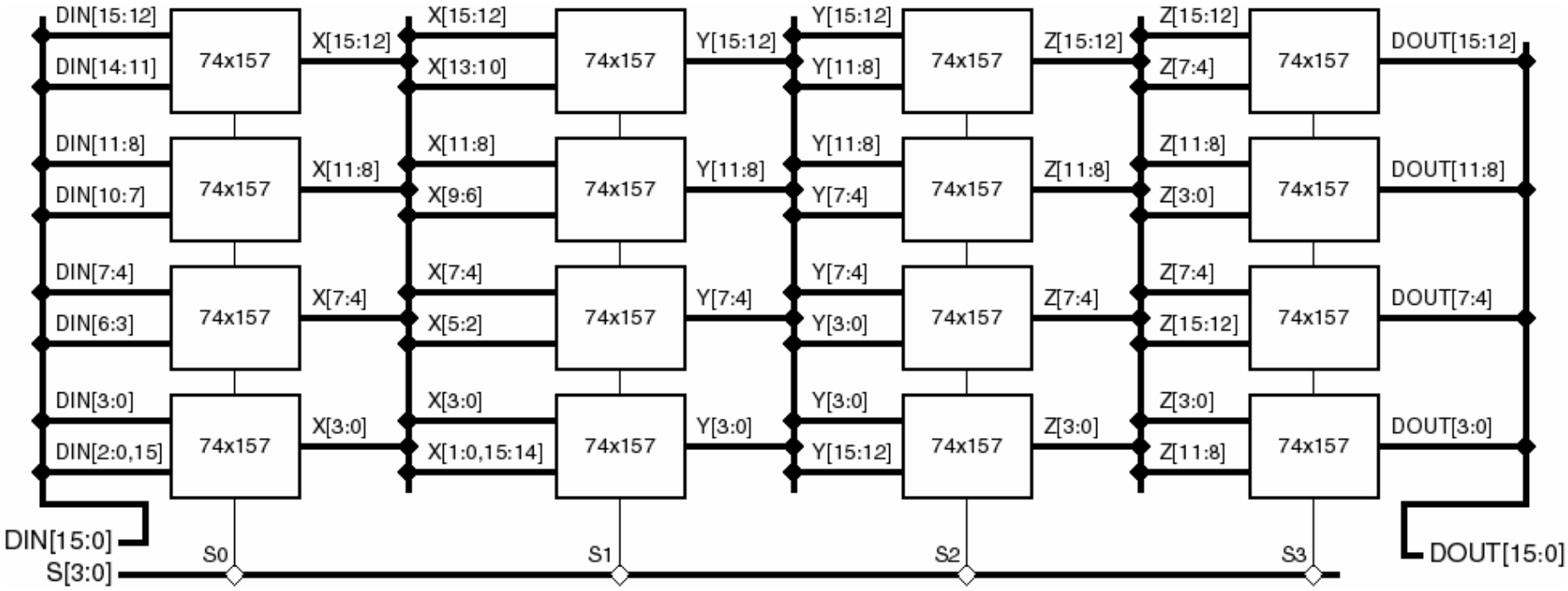
Many possible solutions, all based on multiplexers

16 16-to-1 muxes

16-to-1 mux =
2 x 74x151
8-to-1 mux
+ NAND gate



4 16-bit 2-to-1 muxes



16-bit 2-to-1 mux = 4 x 74x157 4-bit 2-to-1 mux

Properties of different approaches

Multiplexer Component	Data Loading	Data Delay	Control Loading	Total ICs
74x151	16	2	32	36
74x251	16	1	32	32
74x153	4	2	8	16
74x157	2	4	4	16

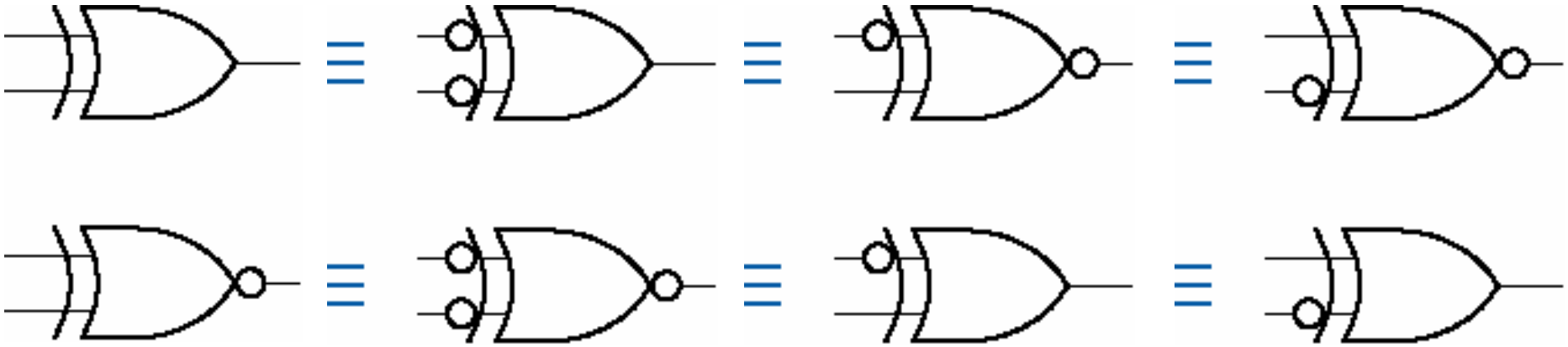
2-input XOR gates

Like an OR gate, but ***excludes*** the case where both inputs are 1.

X	Y	X ⊕ Y (XOR)	(X ⊕ Y)' (XNOR)
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

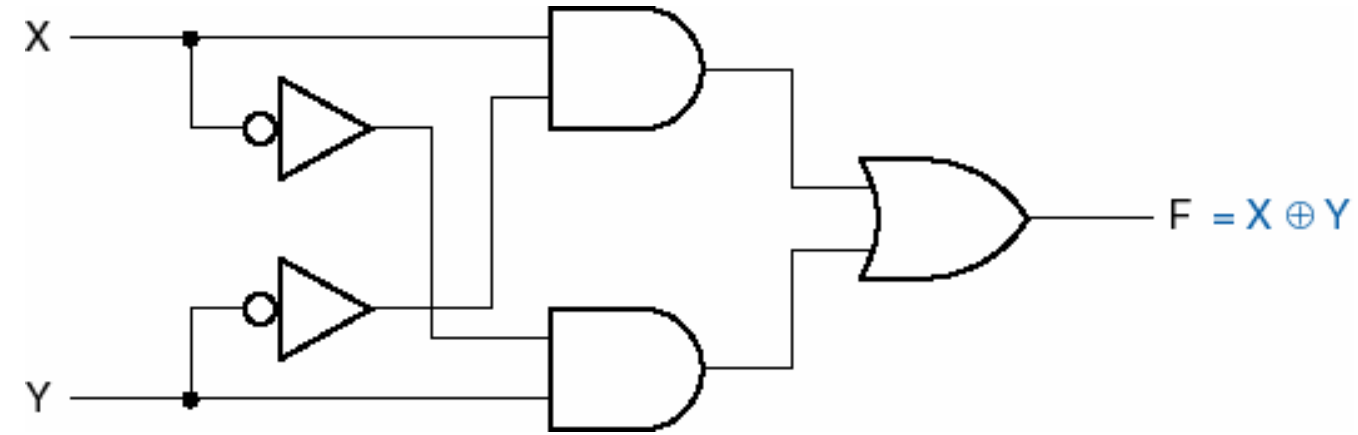
XNOR: complement of XOR

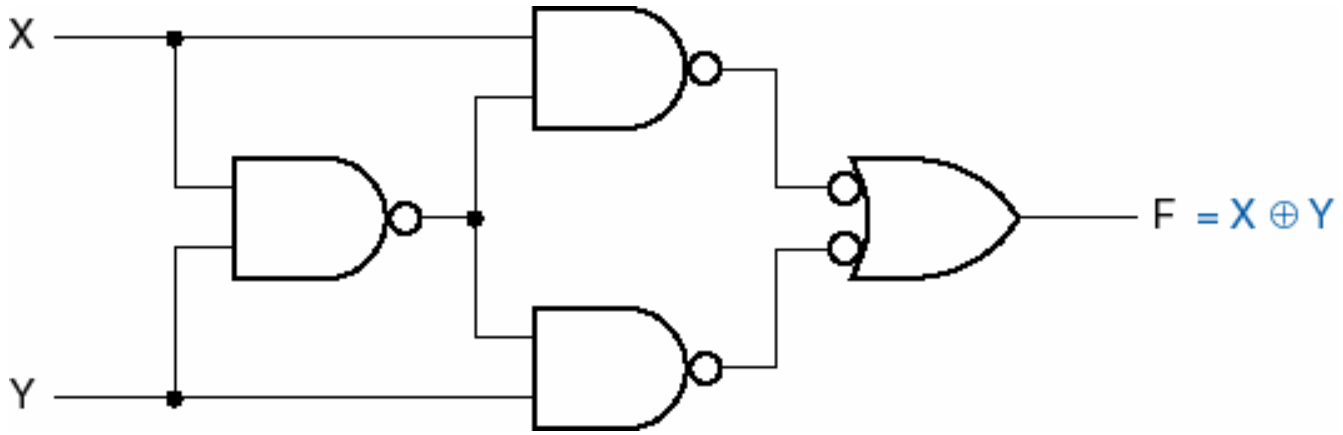
XOR and XNOR symbols



Gate-level XOR circuits

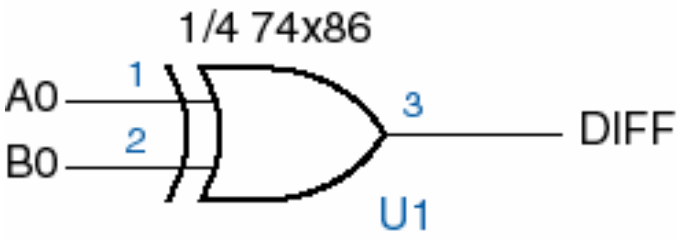
No direct realization with just a few transistors.



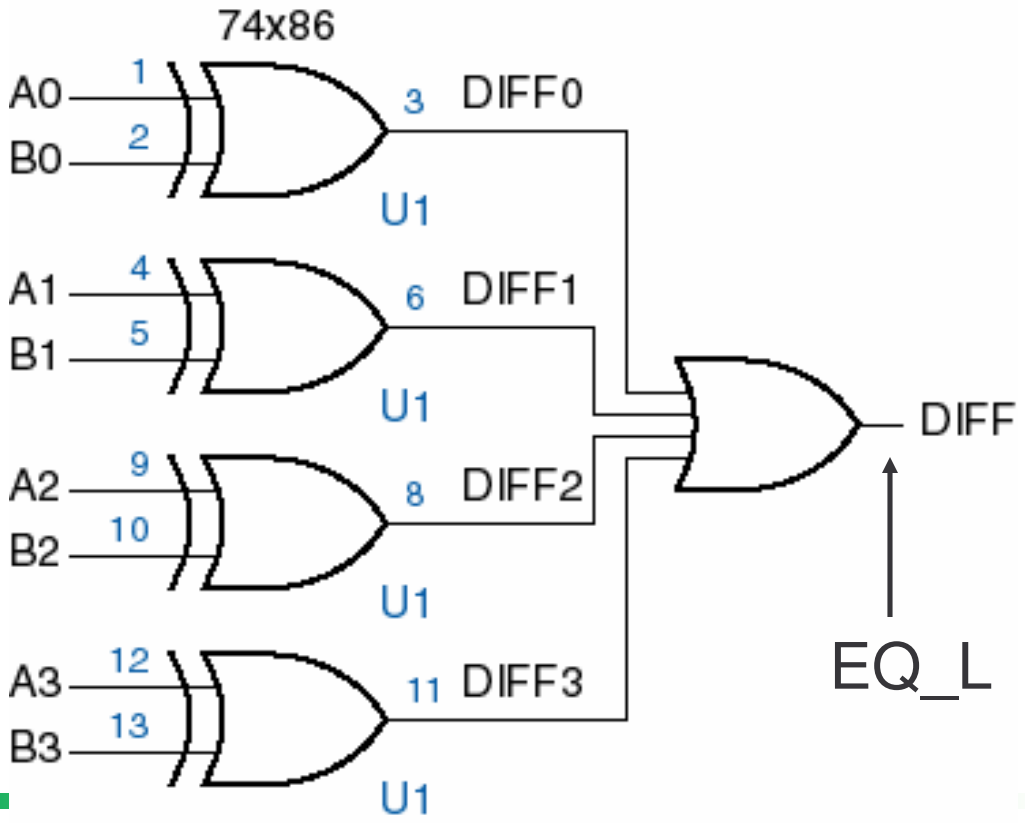


Equality Comparators

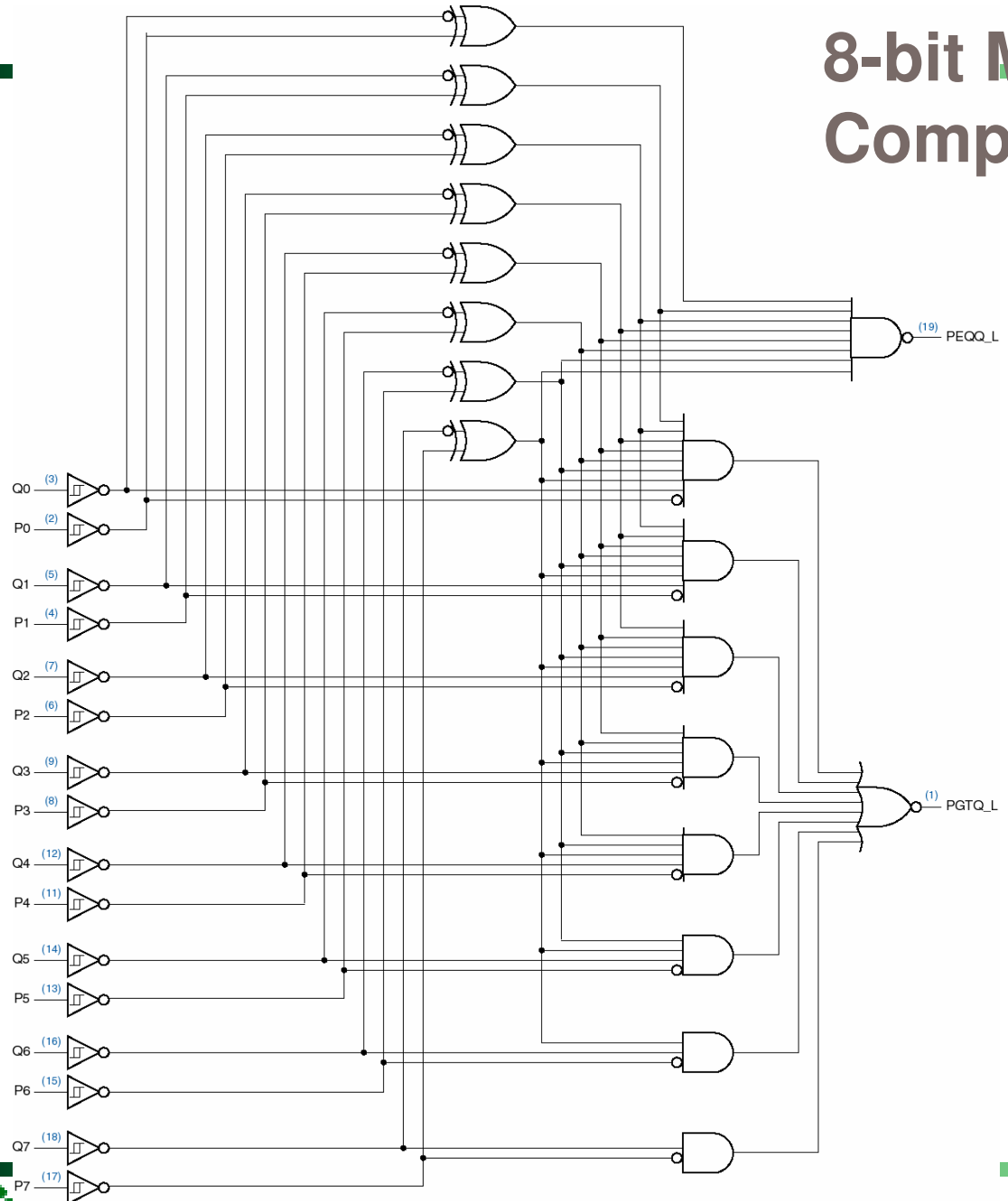
1-bit comparator

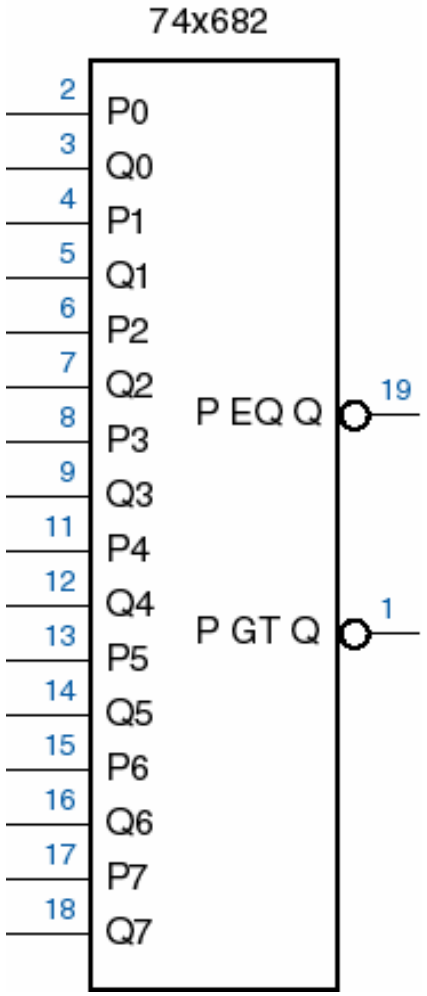


4-bit comparator

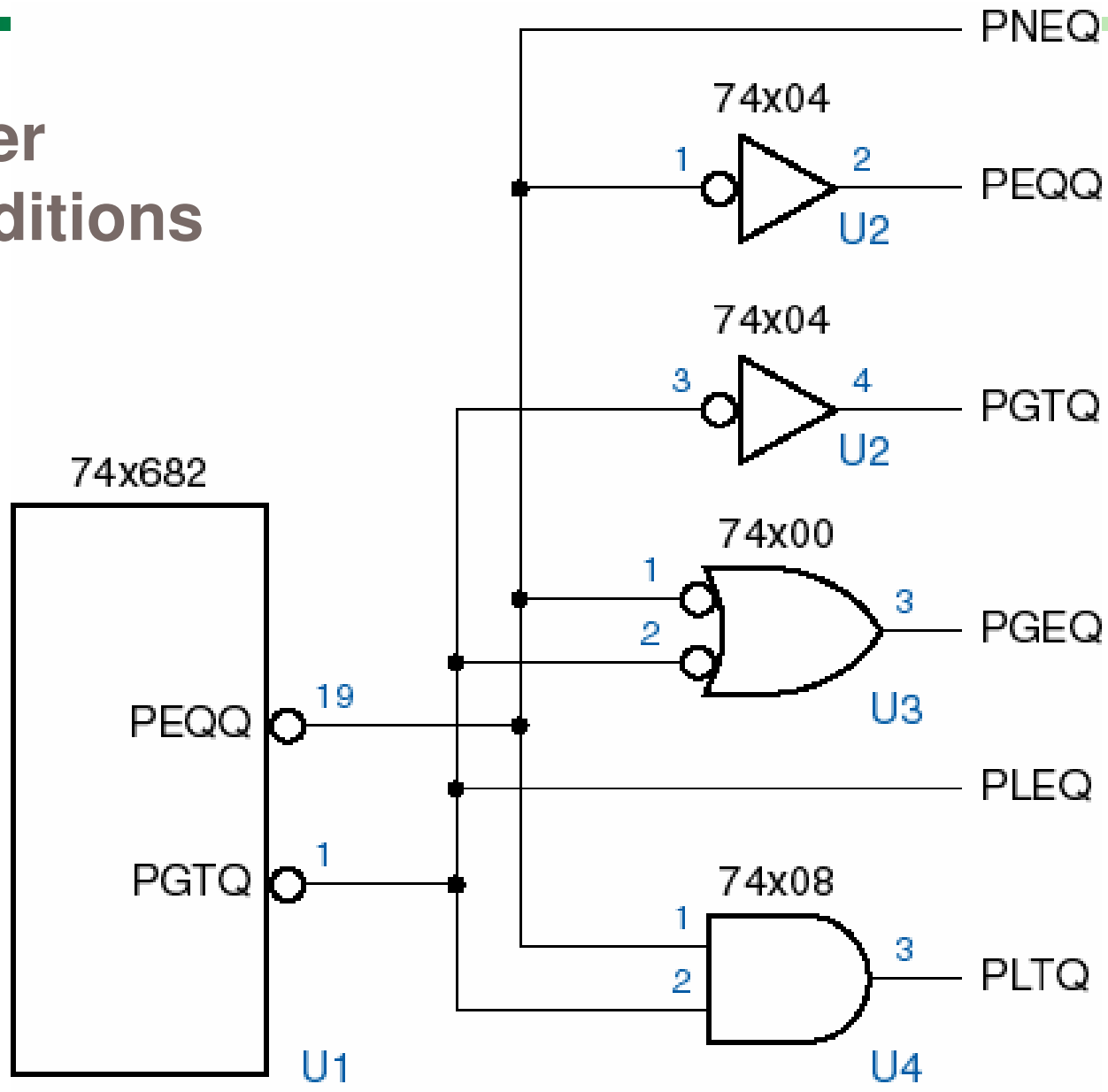


8-bit Magnitude Comparator





Other conditions



Adders

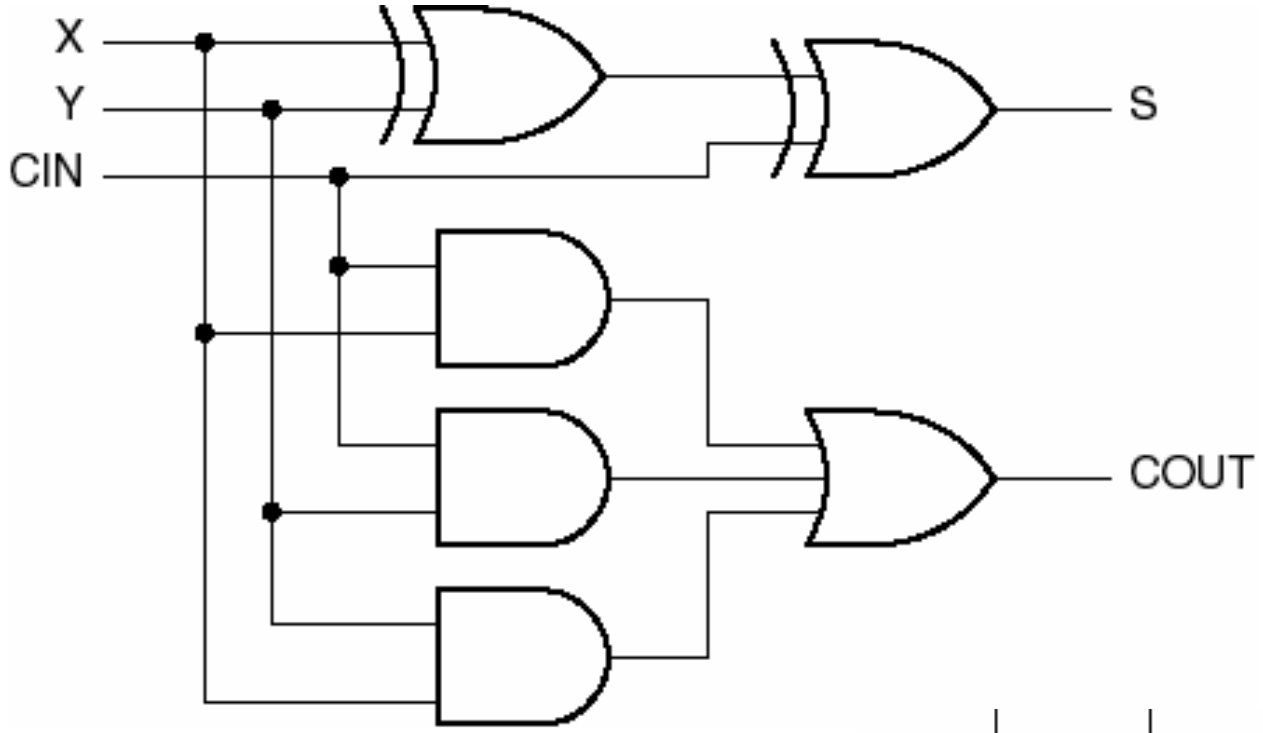
Basic building block is “full adder”

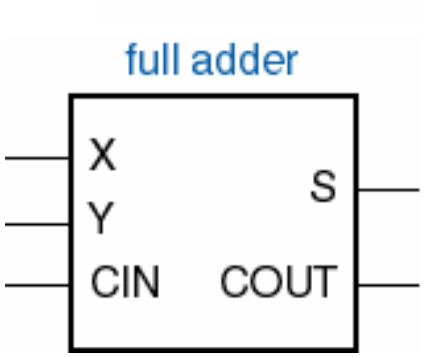
- 1-bit-wide adder, produces sum and carry outputs

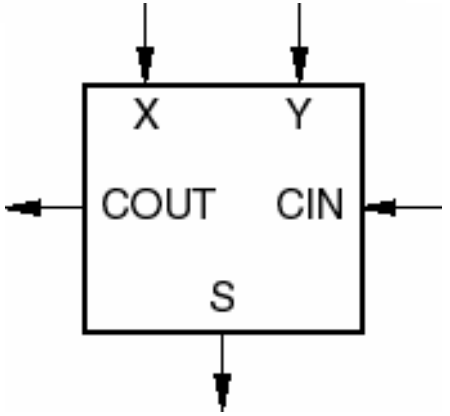
Truth table:

X	Y	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

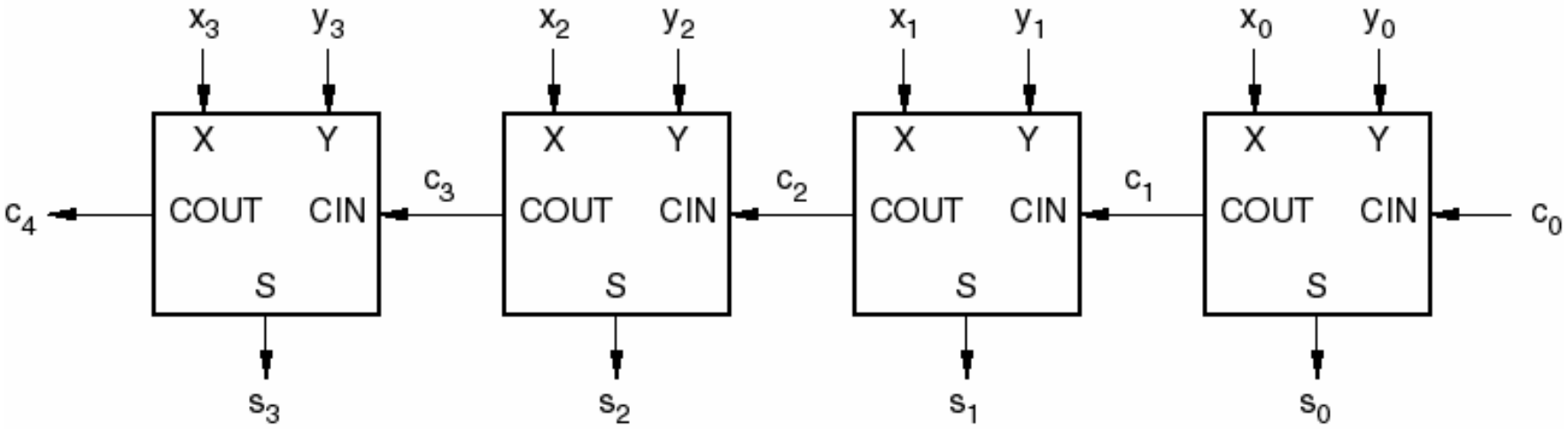
Full-adder circuit







Ripple adder

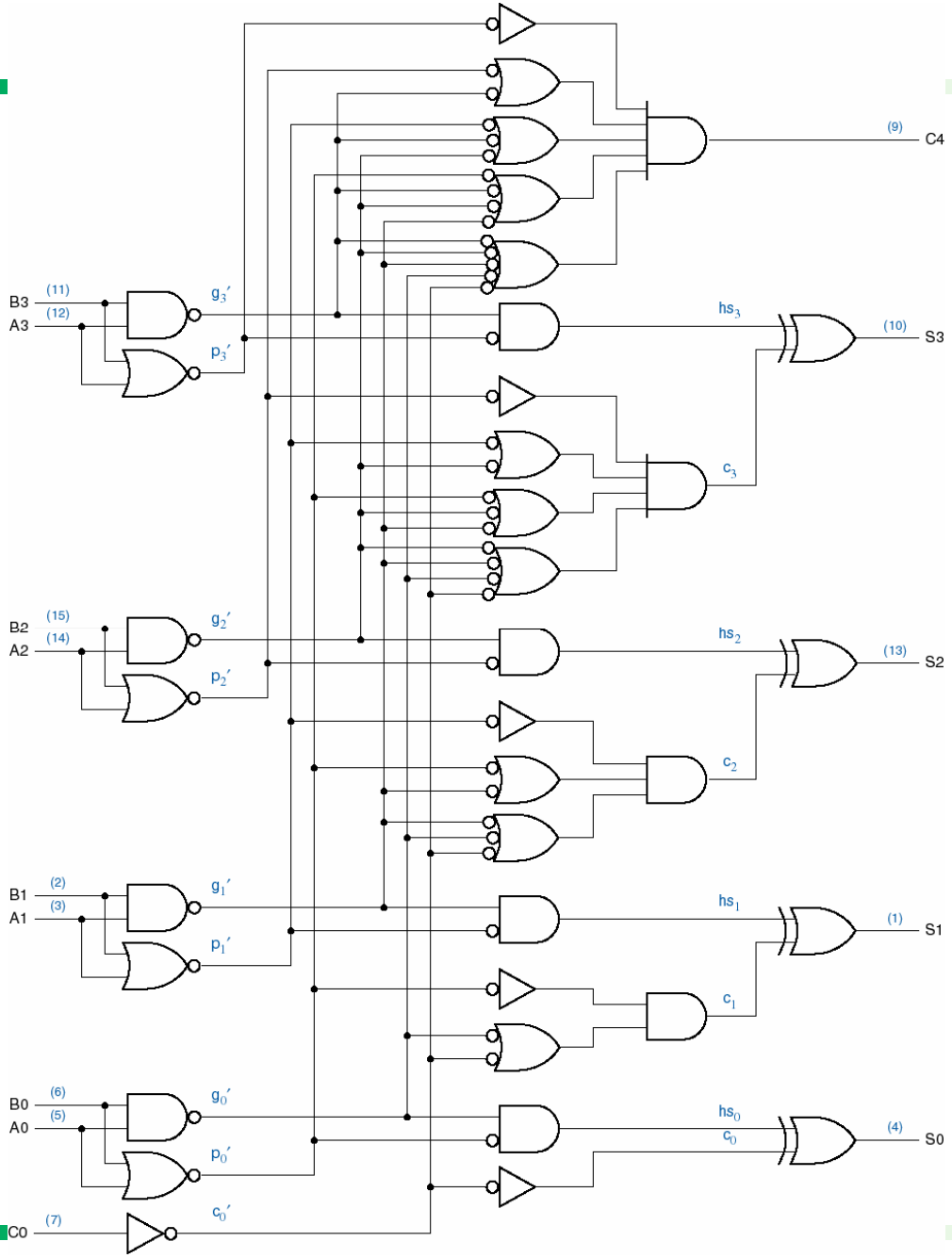


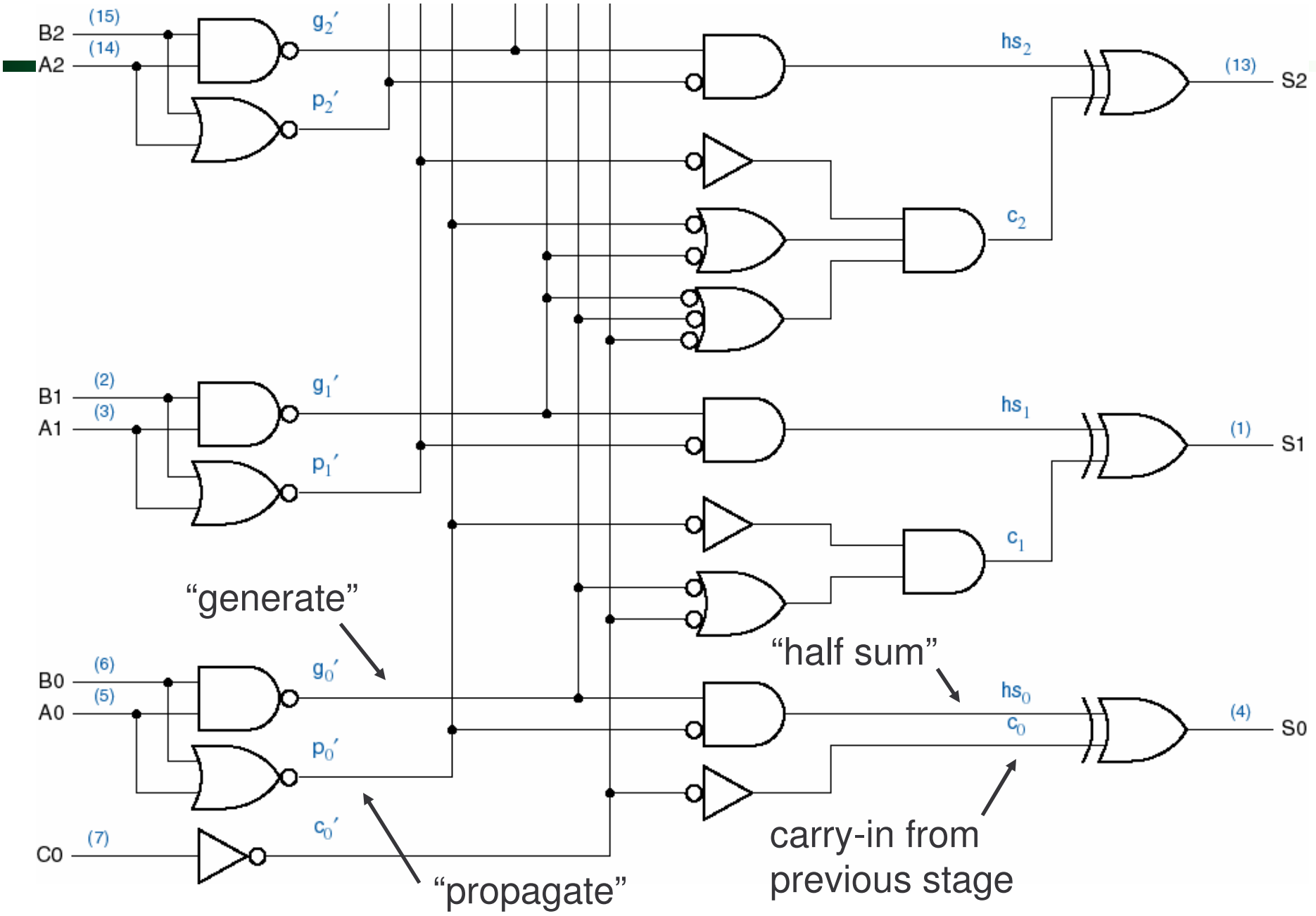
Faster adders eliminate or limit carry chain

- 2-level AND-OR logic ==> 2^n product terms
- 3 or 4 levels of logic, carry lookahead

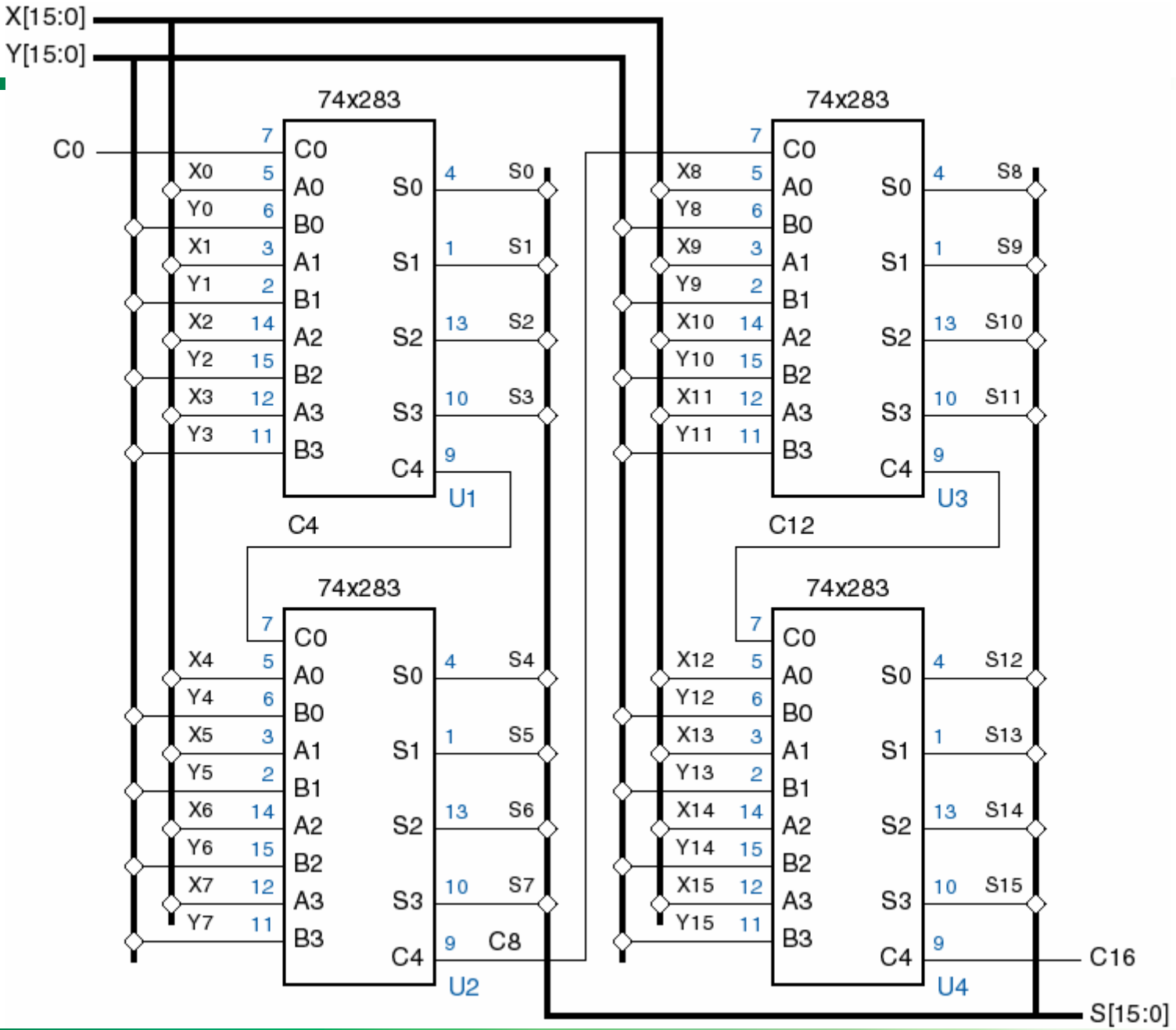
74x283
4-bit adder

Uses carry lookahead internally

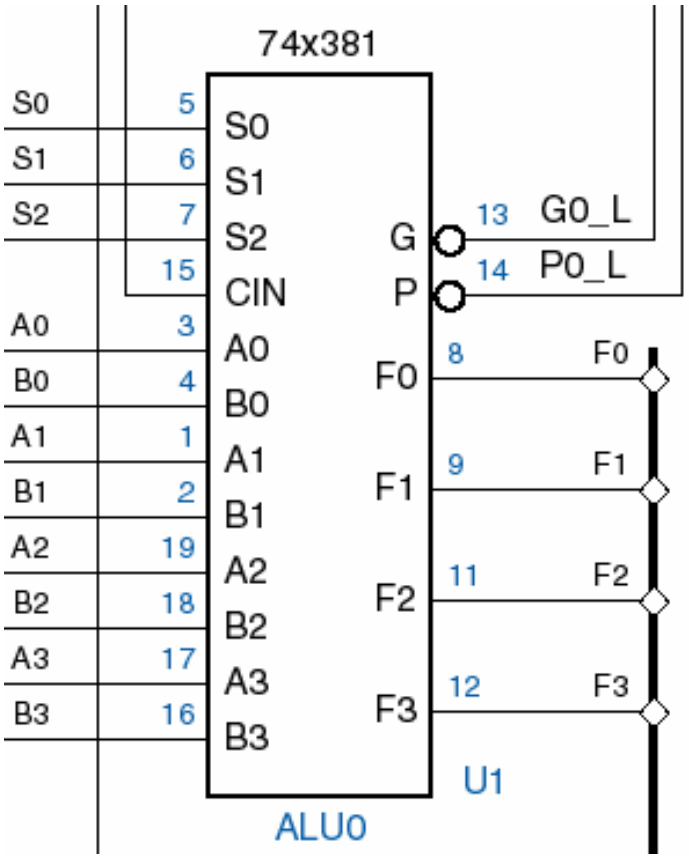


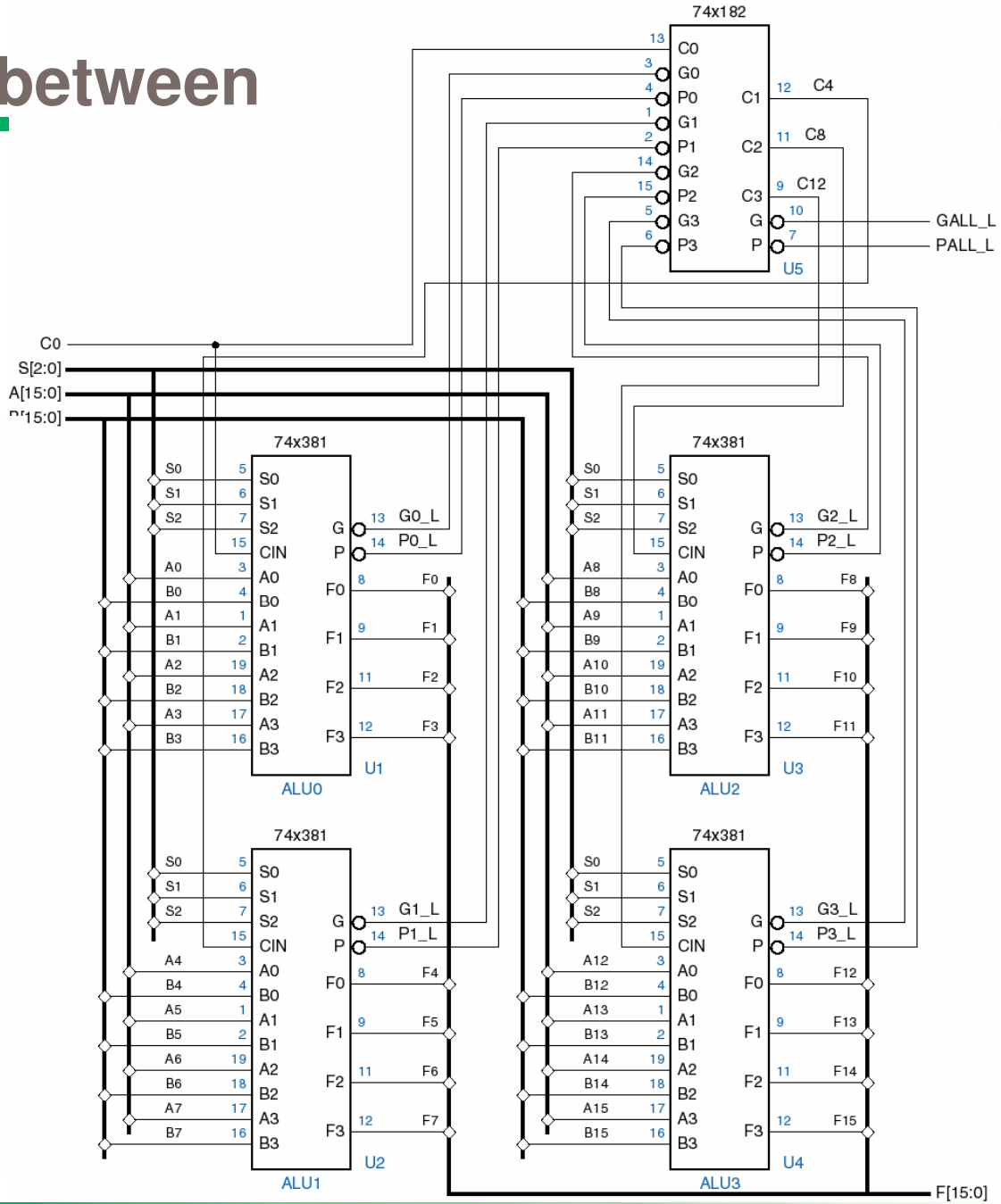

B2
(15)
A2
(14)
g_2'
p_2'
hs_2
(13)
S2
c_2
B1
(2)
A1
(3)
g_1'
p_1'
hs_1
(1)
S1
c_1
"generate"
"half sum"
B0
(6)
A0
(5)
g_0'
p_0'
hs_0
c_0
(4)
S0
C0
(7)
c_0'
"propagate"
carry-in from
previous stage

Ripple carry between groups



Lookahead carry between groups





Subtraction

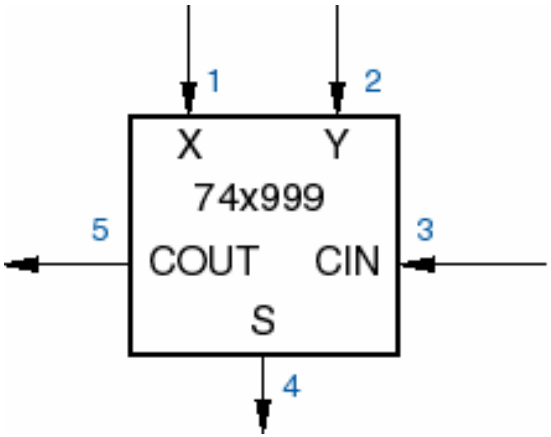
Subtraction is the same as addition of the two's complement.

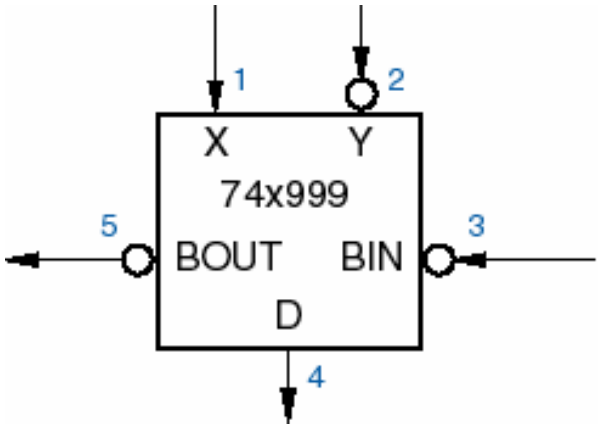
The two's complement is the bit-by-bit complement plus 1.

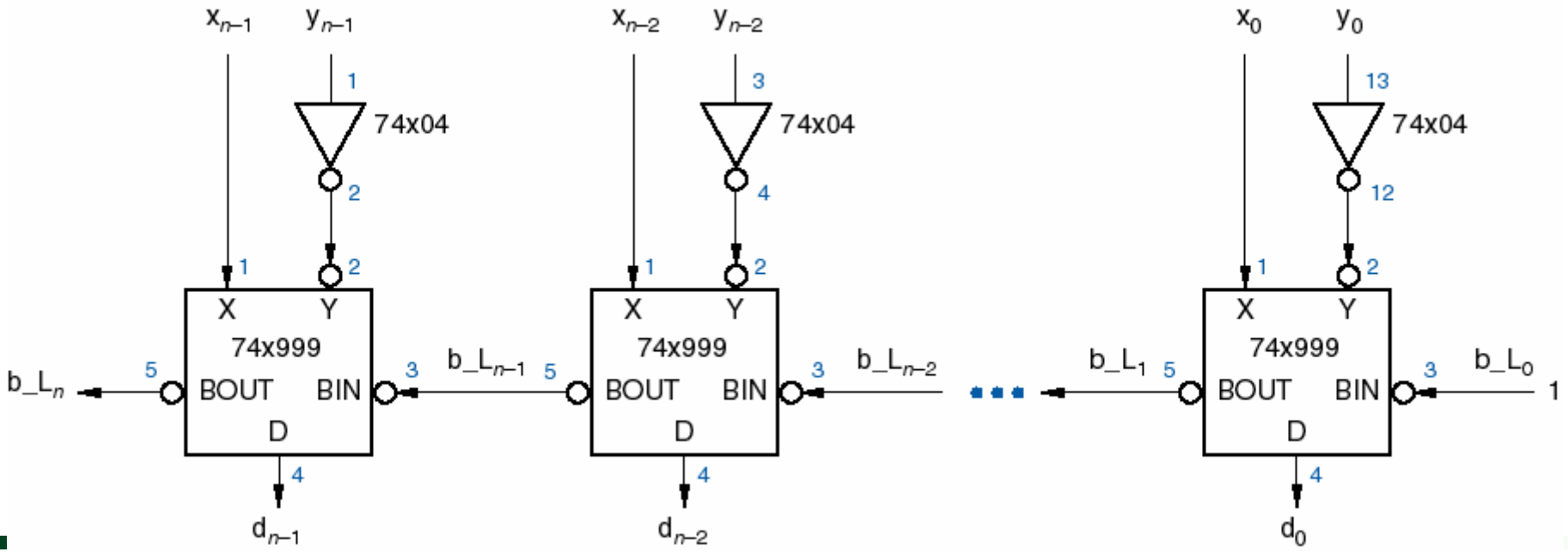
Therefore, $X - Y = X + \overline{Y} + 1$.

- Complement Y inputs to adder, set C_{in} to 1.
- For a borrow, set C_{in} to 0.

Full subtractor = full adder, almost

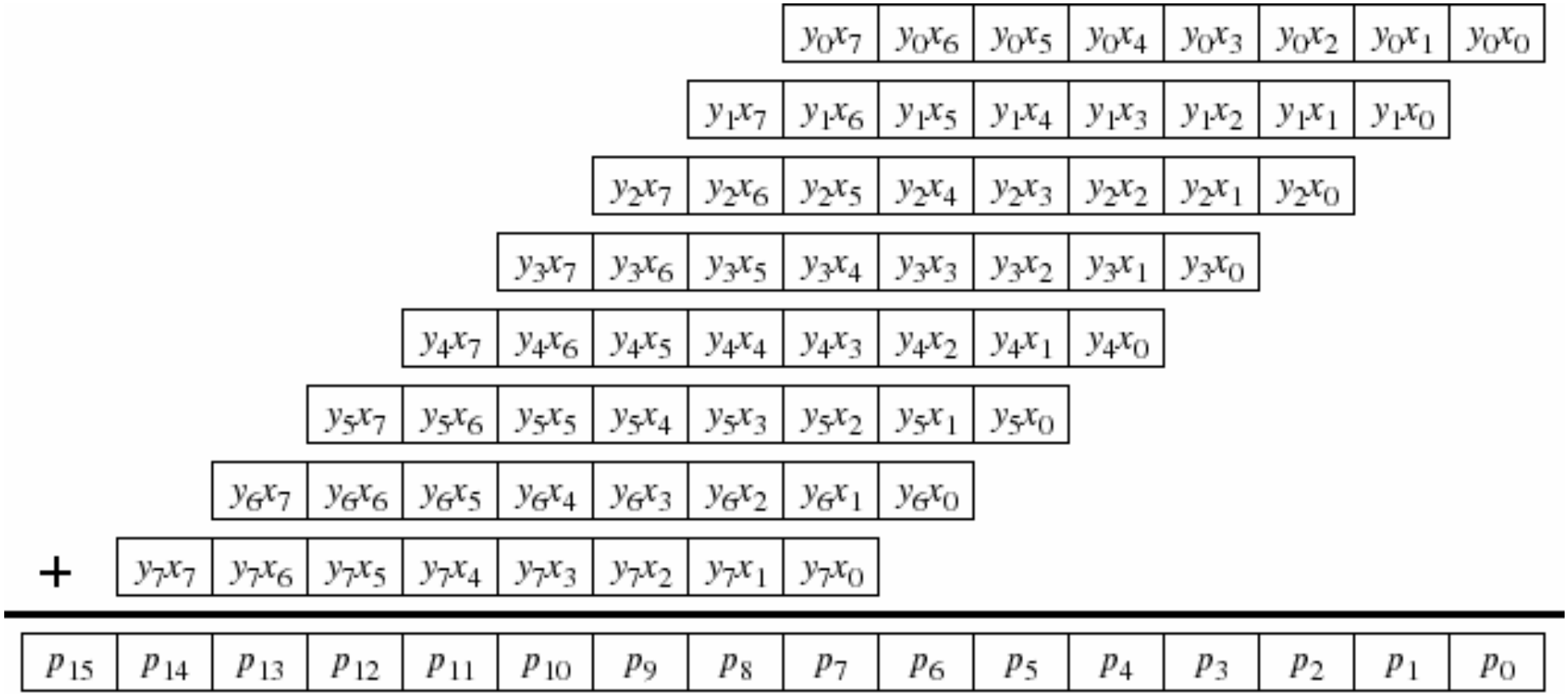




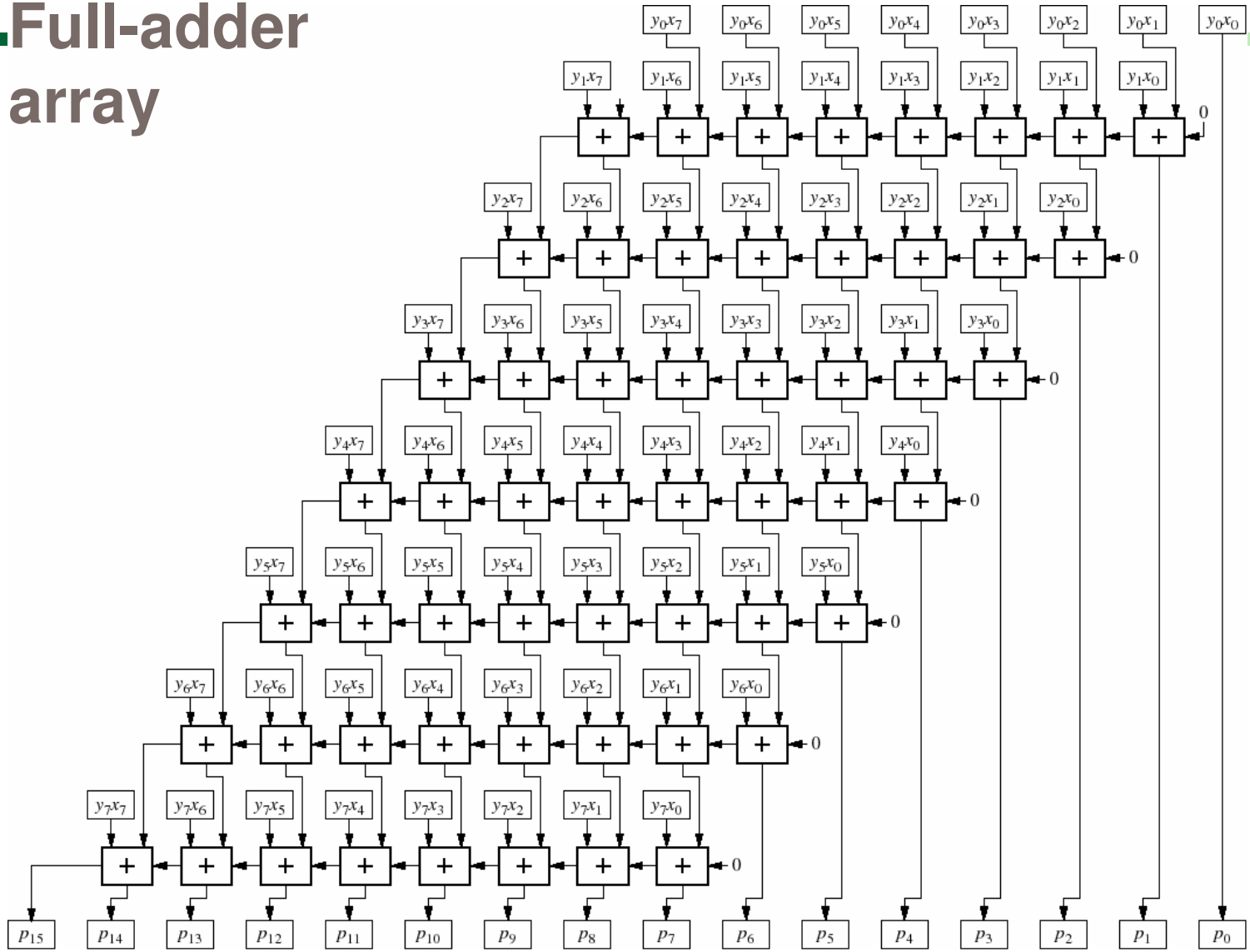


Multipliers

8x8 multiplier



Full-adder array



Faster carry chain