



Tackling circuit complexity (2)



Paolo PRINETTO
 Politecnico di Torino (Italy)
 University of Illinois at Chicago, IL (USA)

Paolo.Prinetto@polito.it
 Prinetto@uic.edu
 www.testgroup.polito.it

Goal

- This lecture is the latter part of a group of 2 lectures aiming at presenting methods used in manual combinational synthesis to overcome the limitations of the purely manual approach presented in lectures 6.1-6.3

6.5

2

Prerequisites

- Lectures 5.1 and 6.4

6.5

3

Homework

- Students are warmly encouraged to solve the proposed exercises

6.5

4

Further readings

- No particular suggestion

6.5

5

Outline

- Partitioning based techniques
- RT level minimizations
- Synthesis by iterating basic cells

6.5

6

Partitioning-bases techniques

- The system is first partitioned in functional blocks
- Each functional block is then implemented resorting to one of the above mentioned approaches
- The system is eventually designed simply assembling the functional blocks.

6.5

7

Partitioning-bases techniques

- The system is first partitioned in functional blocks
- Each functional block is then implemented resorting to one of the above mentioned approaches
- The system is eventually designed simply assembling the functional blocks.

This step is iterated until each functional block is implemented resorting to elements that are considered to be "basic" or "elementary" for the target design

6.5

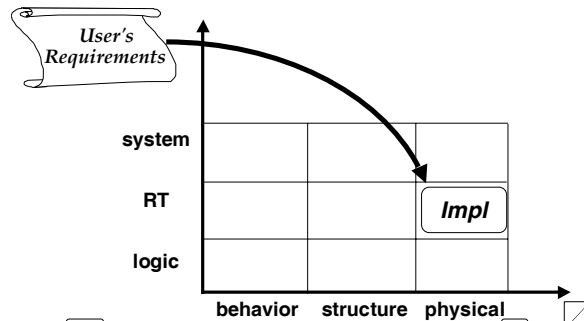
RT Level Synthesis

- A peculiar case is the one in which the "elementary blocks" are the RT level combinational blocks presented in lecture 5.2.

6.5

9

RT Level Synthesis



6.5

10

RT level synthesis

Approaches to manual RT level synthesis can be clustered in 2 major classes:

- *intuitive*
- *systematic (from VHDL RT descriptions)* (presented in module 12).

6.5

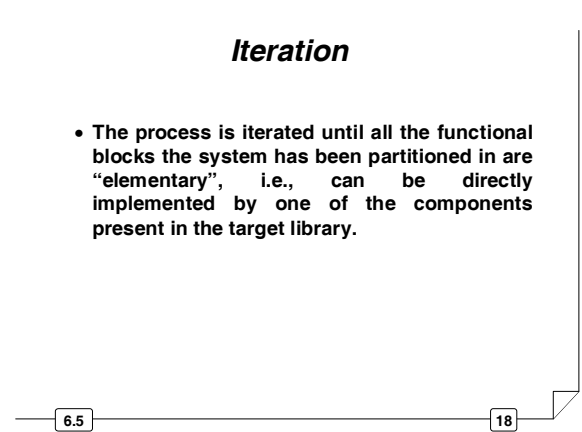
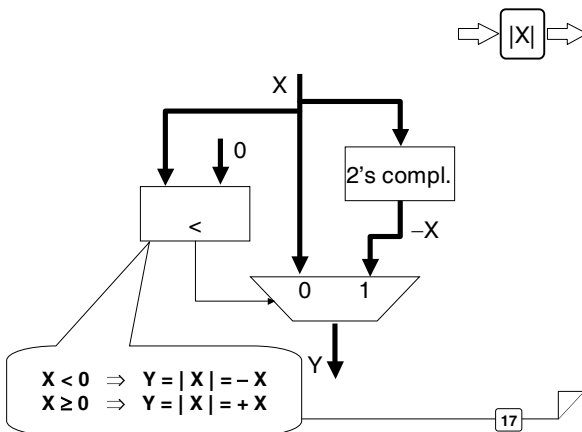
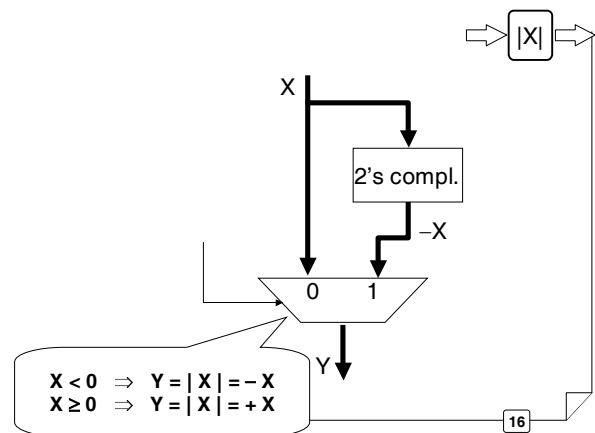
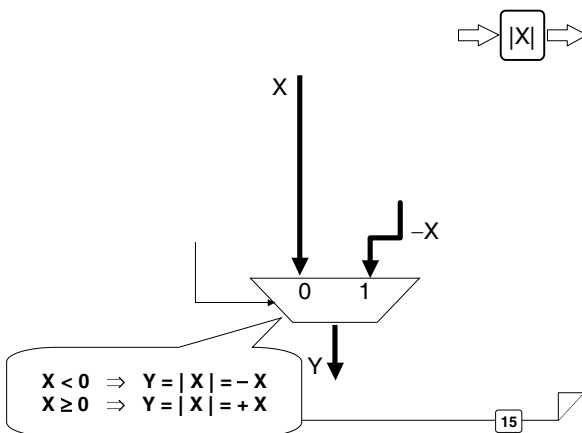
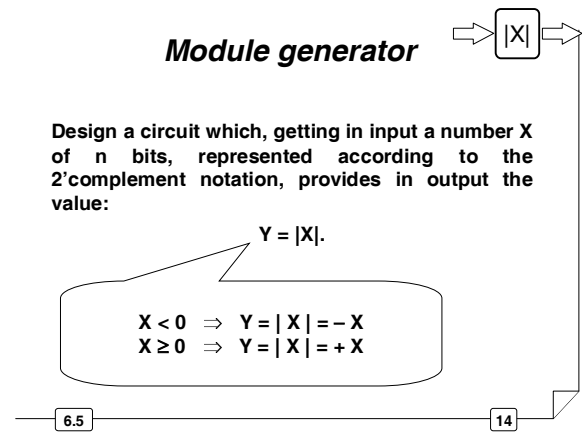
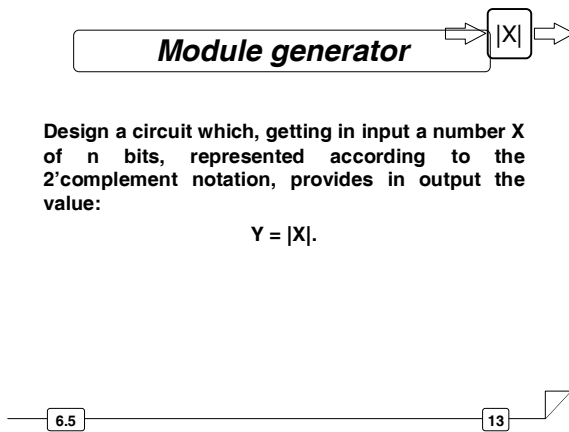
11

Intuitive approach

The intuitive approach will be presented resorting to a couple of examples.

6.5

12

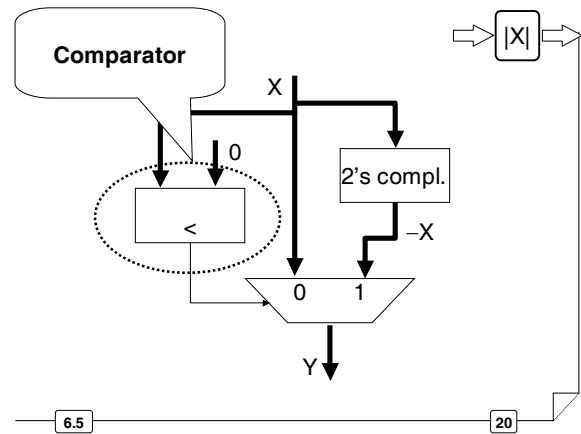


Iteration

- The process is iterated until all the functional blocks the system has been partitioned in are “elementary”, i.e., can be directly implemented by one of components present in the target library.
- Let's assume the target library be the *RT level combinational blocks* presented in lecture 5.1

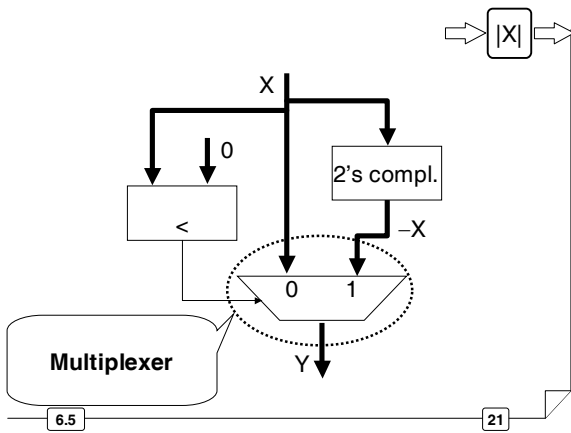
6.5

19



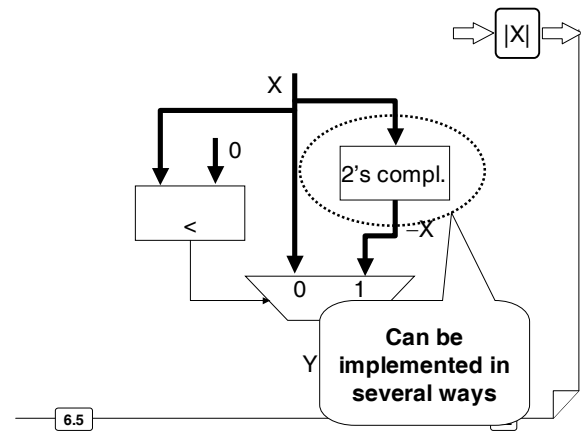
6.5

20



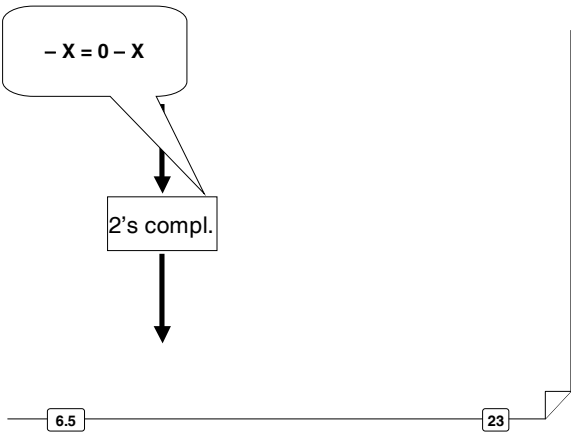
6.5

21



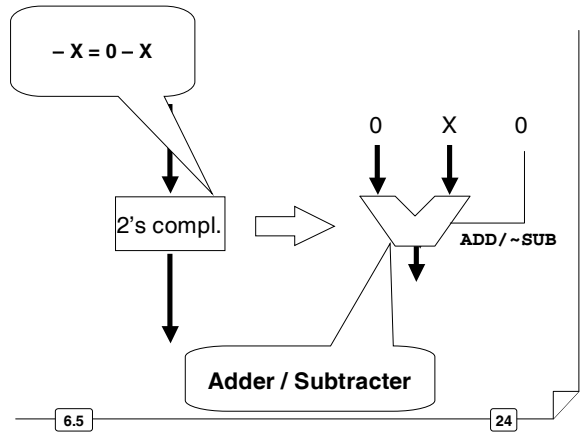
6.5

22



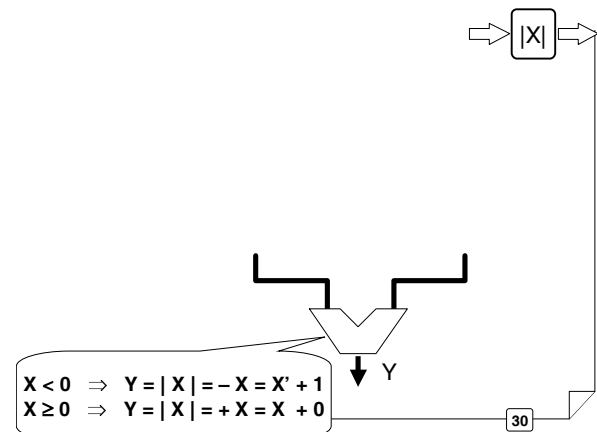
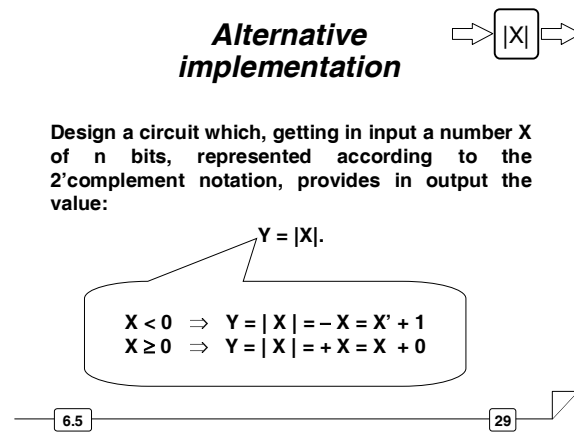
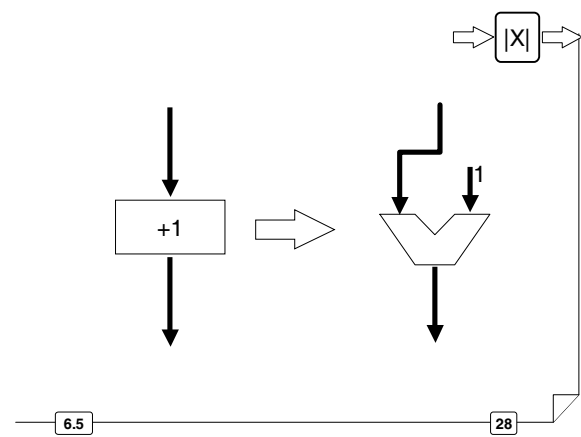
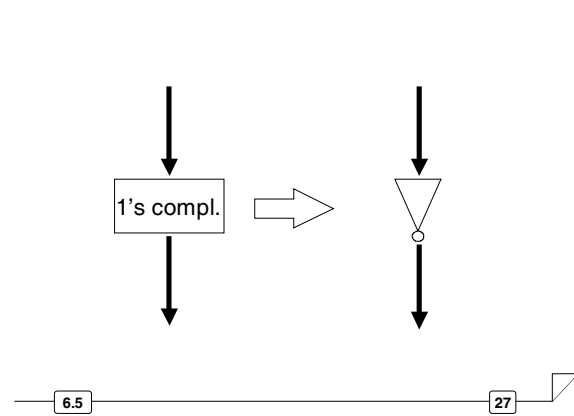
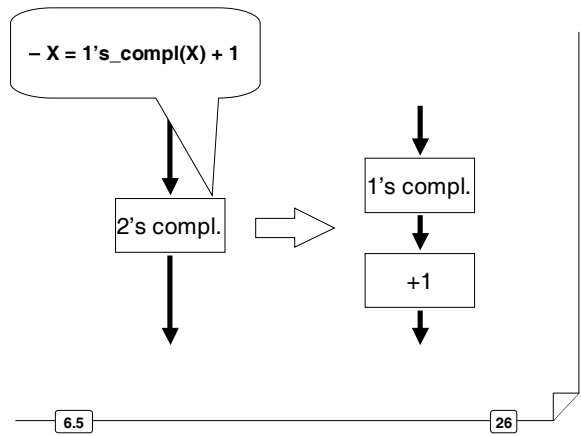
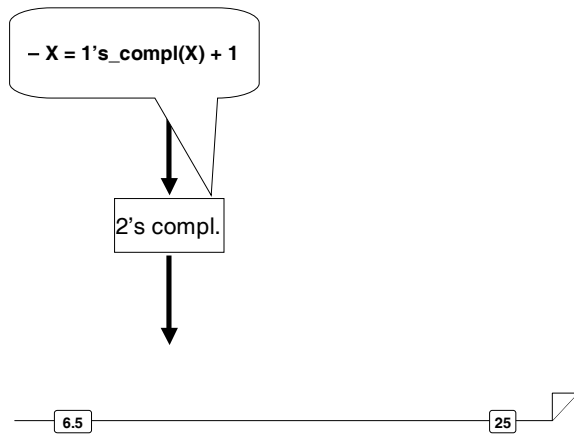
6.5

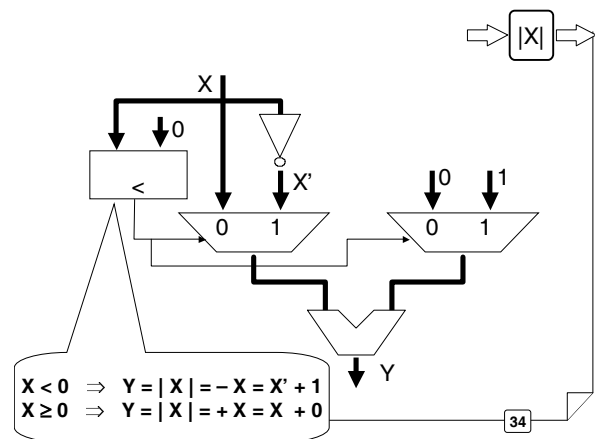
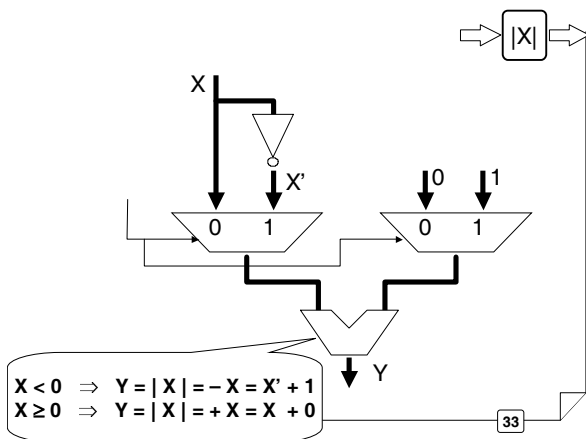
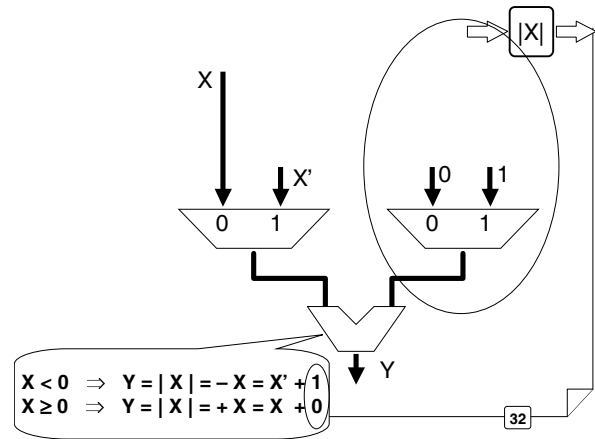
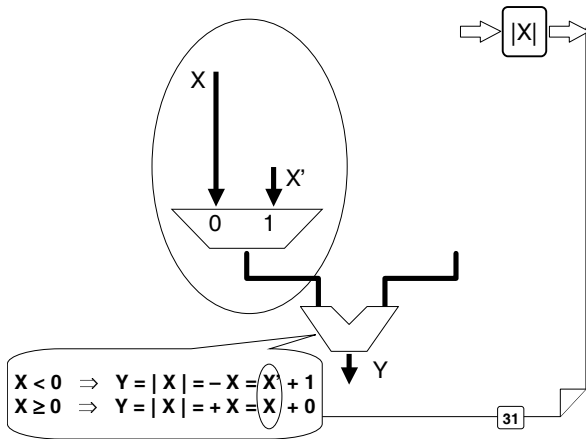
23



6.5

24





Programmable Divider

Design a circuit which, getting in input:

- An 8-bit number X coded 2C
- A 2-bit number D coded 2C

provides on the output Y (coded 2C) the value

$$Y = X / D$$

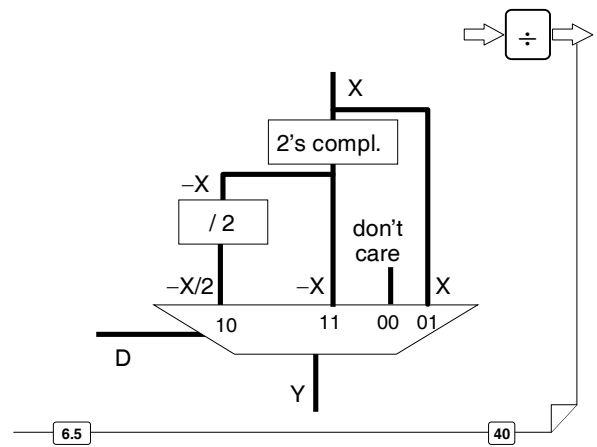
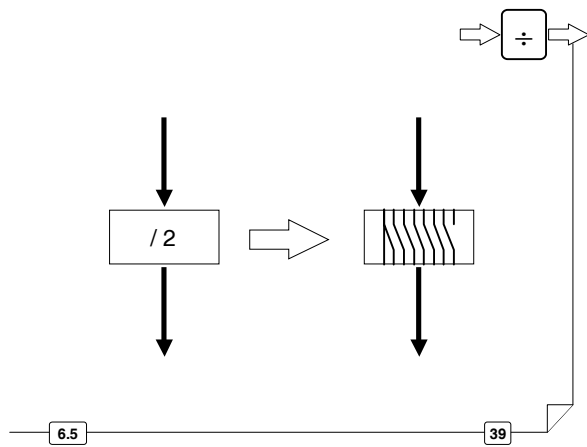
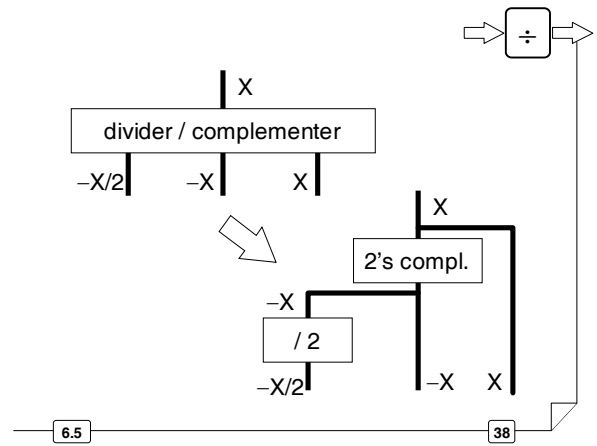
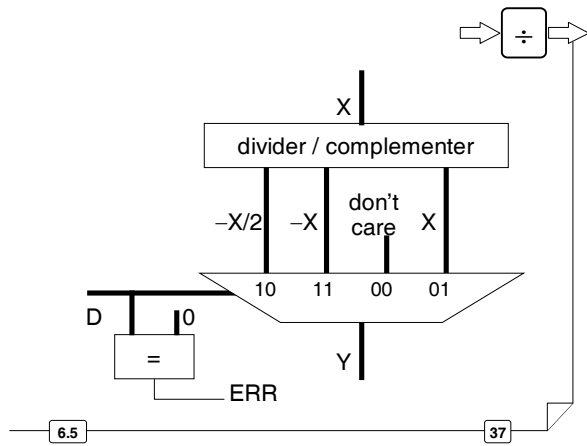
6.5 35

Solution

The divider D can get the following 4 values, only:

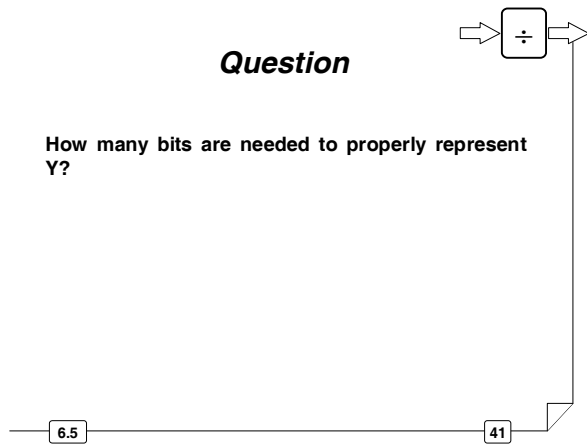
- D=-2 Y = -X/2
- D=-1 Y = -X
- D=0 error
- D=1 Y = X

6.5 36



Question

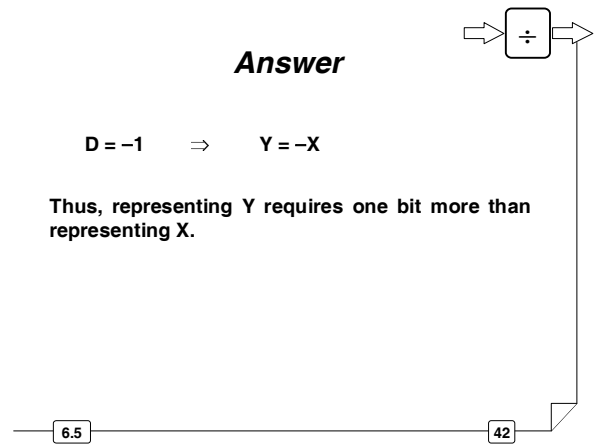
How many bits are needed to properly represent Y?



Answer

$$D = -1 \Rightarrow Y = -X$$

Thus, representing Y requires one bit more than representing X.



Outline

- Partitioning based techniques
 ⇒ *RT level minimizations*
- Synthesis by iterating basic cells

6.5

43

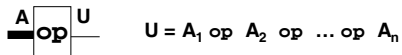
Hints

At the end of an intuitive RT level synthesis, it's highly recommended to perform an optimization step aiming at locally minimizing the Functional Blocks.

6.5

44

Graphic notations

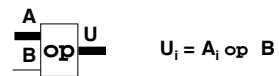
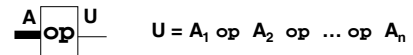


where op = and, or, xor, nand, nor, exnor

6.5

45

Graphic notations

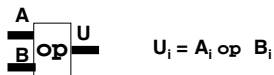
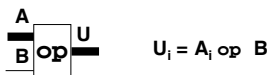
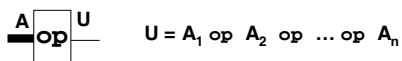


where op = and, or, xor, nand, nor, exnor

6.5

46

Graphic notations



where op = and, or, xor, nand, nor, exnor

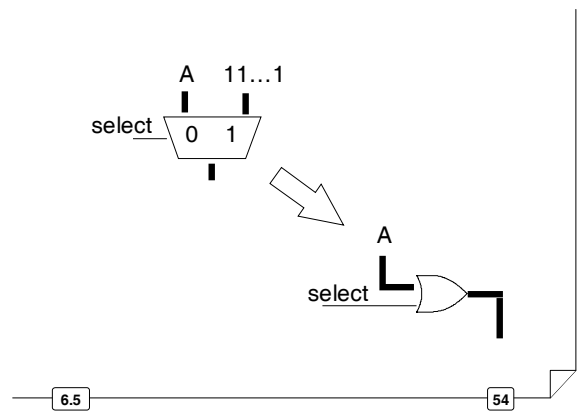
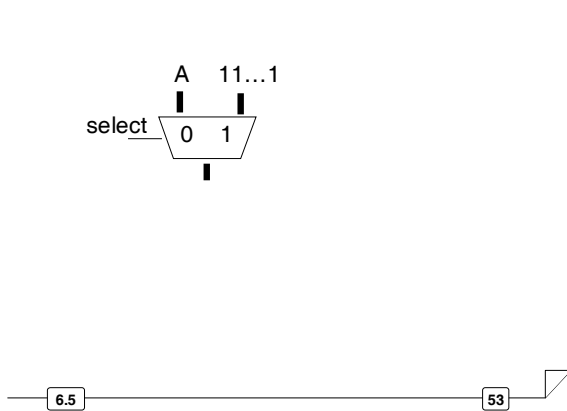
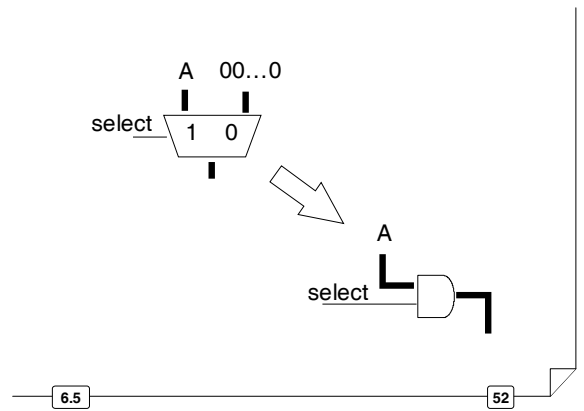
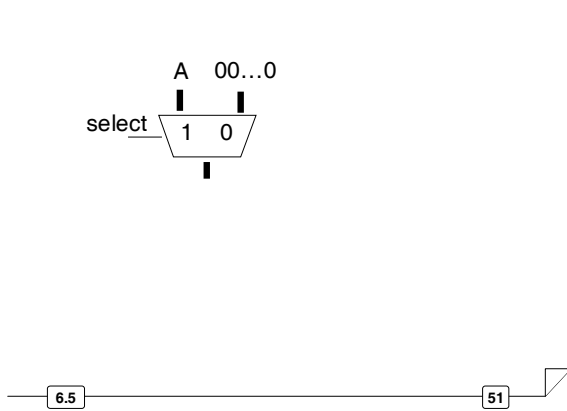
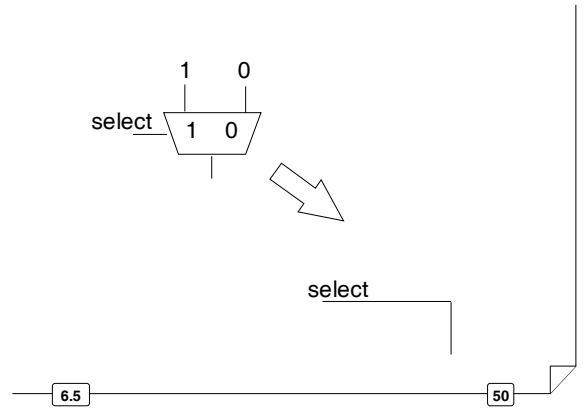
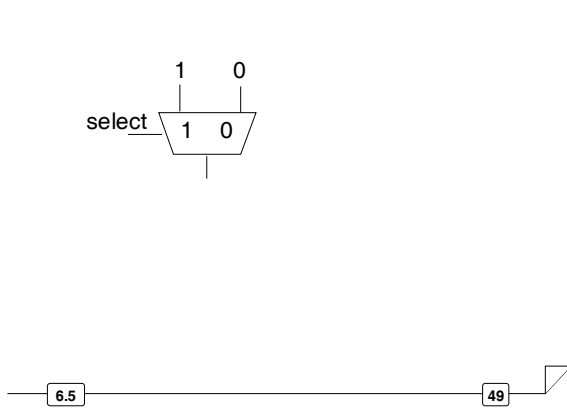
6.5

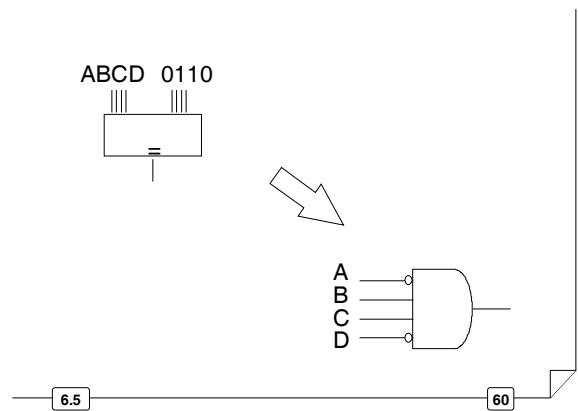
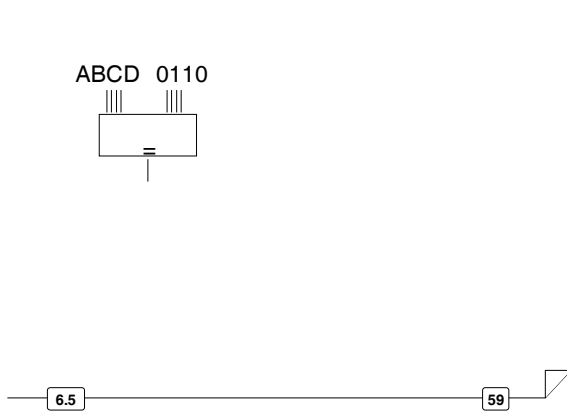
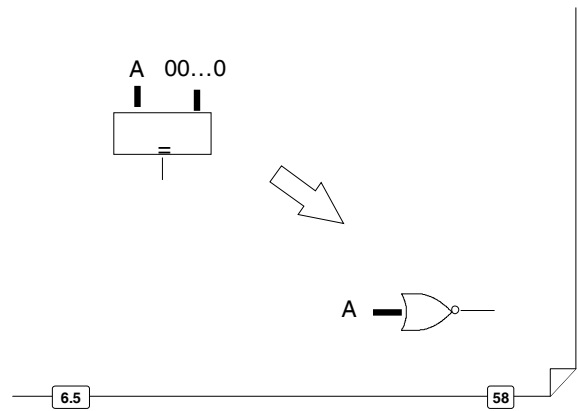
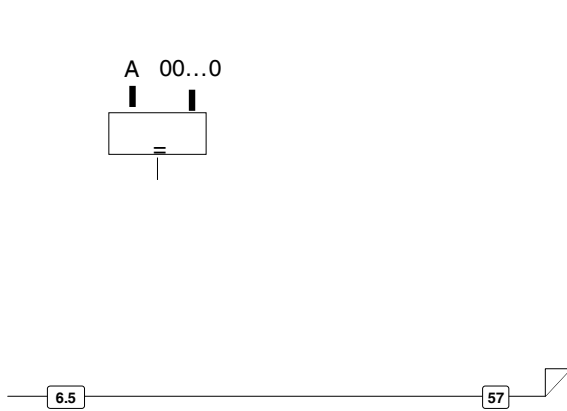
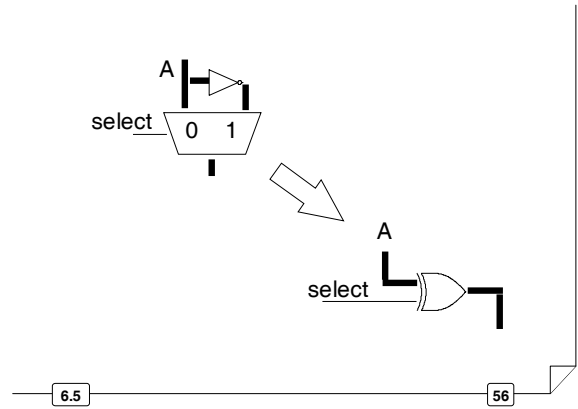
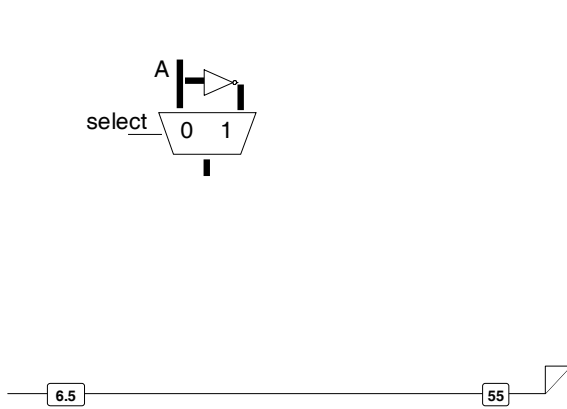
47

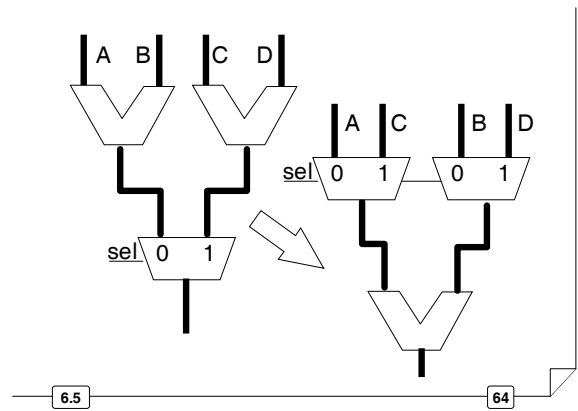
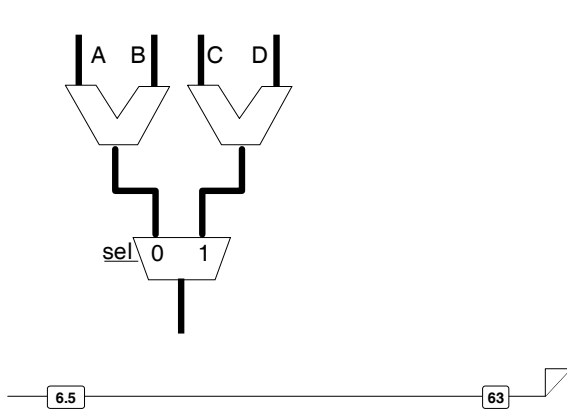
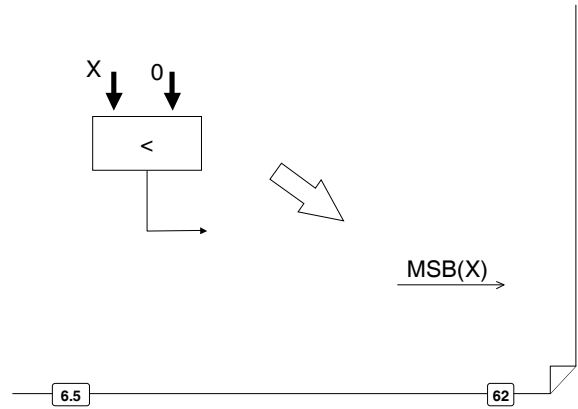
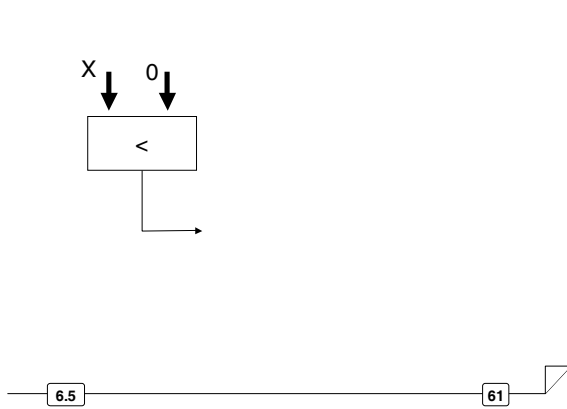
Some optimizations...

6.5

48

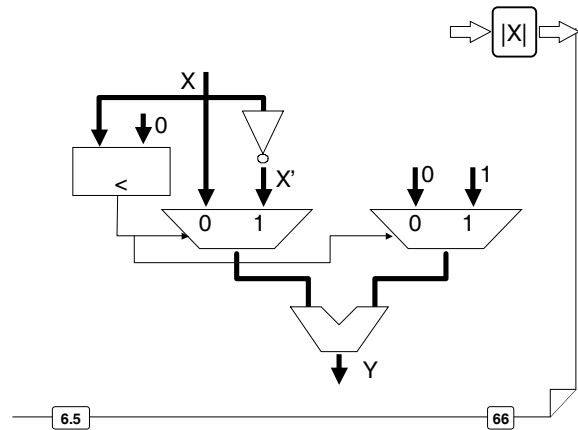
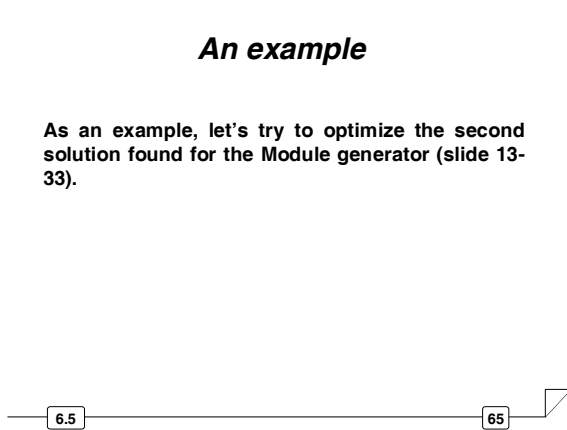


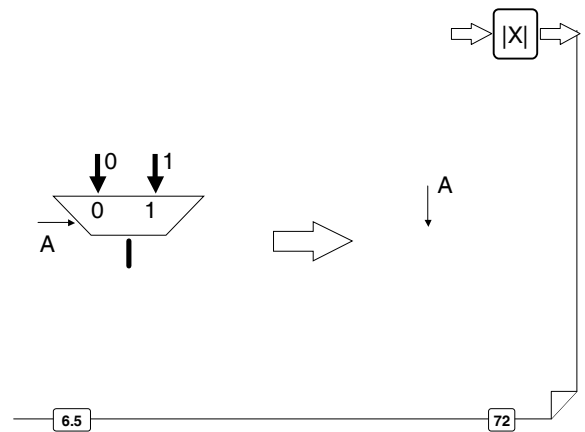
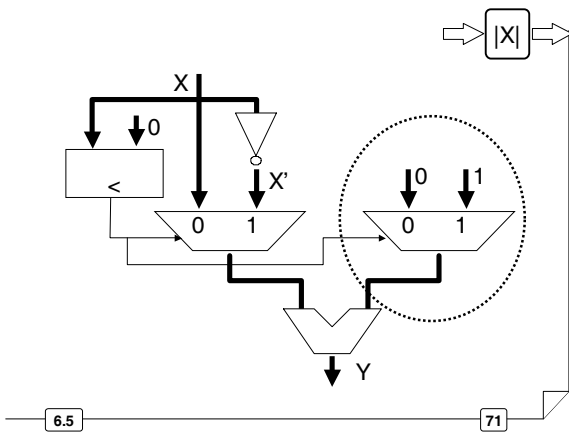
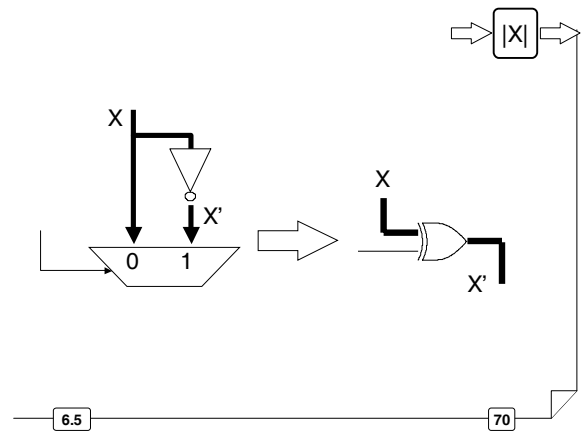
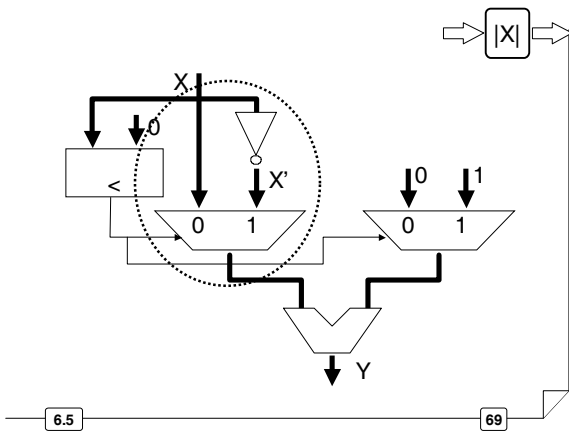
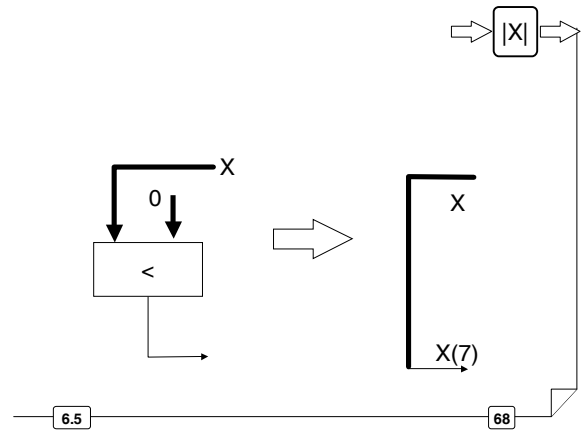
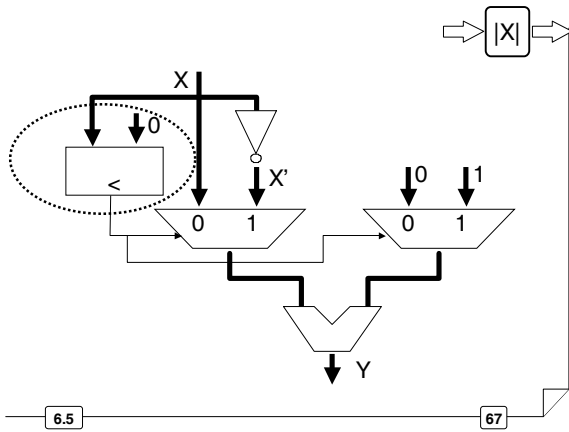




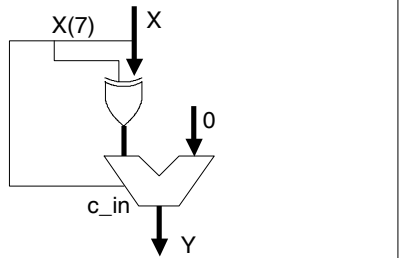
An example

As an example, let's try to optimize the second solution found for the Module generator (slide 13-33).





Optimized Implementation



6.5

73

Outline

- Partitioning based techniques
- RT level minimizations
- ⇒ *Synthesis by iterating basic cells*

6.5

74

Synthesis by iterating basic cells

In particular cases, the target architecture can be *sliced* in a set of smaller functional blocks, all each other equal, properly interconnected.

6.5

75

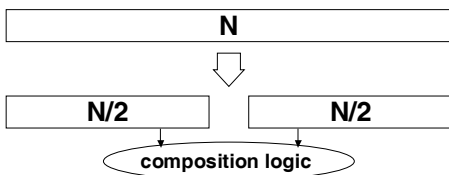
Basic idea



6.5

76

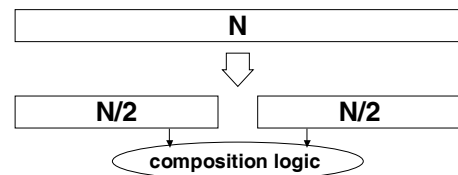
Basic idea



6.5

77

Basic idea



6.5

78

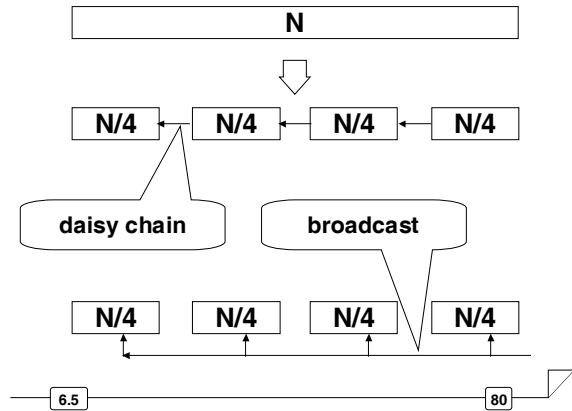
Signal propagation

Two kinds of signals can usually be identified:

- *broadcast*
- *daisy chain*.

6.5

79



Some examples

In the following we shall examine some significant examples of slicing, mainly referred to the RT level combinational blocks introduced in Lecture 5.1

6.5

81

Adders

We shall consider:

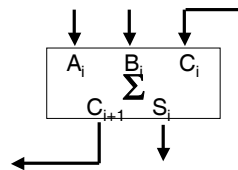
- *full adder*
- *n-bits adders*
 - *Ripple carry adders*
 - *Carry look-ahead adders*

6.5

82

Full adder

Implement a full adder:

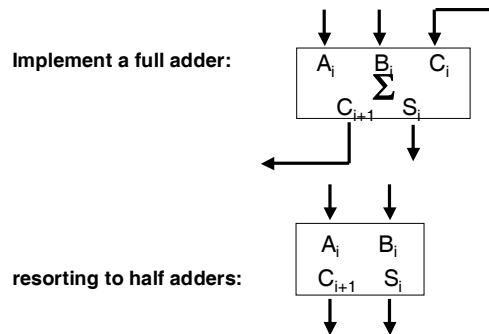


6.5

83

Full adder

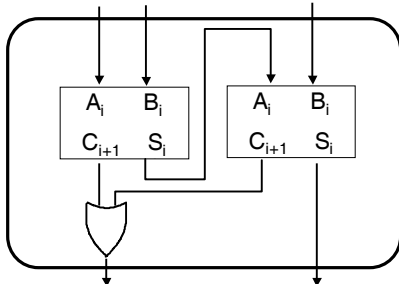
Implement a full adder:



6.5

84

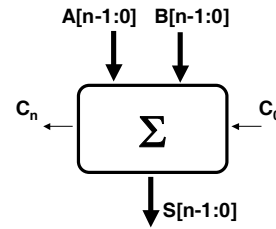
Solution



6.5

85

n-bit adder



6.5

86

n-bit adder

An *n-bit adder* can be implemented resorting to *n* full-adders, connected according to two alternative architectures:

- ripple carry
- look-ahead carry.

6.5

87

Ripple carry adder

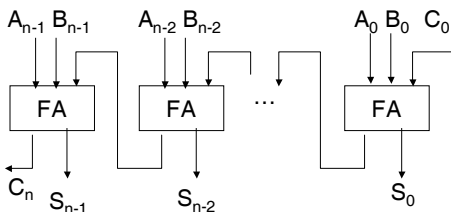
The implementation is based on the same algorithm used when the addition is performed manually by paper and pencil:

the carry input C_i of each cell is fed by the carry output C_{i-1} of the adjacent less significant cell.

6.5

88

Ripple carry adder



6.5

89

Delays

The global delay of the adder is:

$$n \cdot \Delta_{FA}$$

being Δ_{FA} the delay of each full-adder cell.

6.5

90

Look-ahead carry adder

A *look-ahead carry adder* is a faster, but more expensive, implementation.

Each cell generates 2 outputs:

$$P_i = A_i \oplus B_i \quad : \text{propagation function}$$

$$G_i = A_i \cdot B_i \quad : \text{generation function.}$$

6.5

91

Look-ahead carry adder (cont'd)

The output of the *i*-th cell can be expressed as:

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i$$

$$C_{i+1} = A_i B_i + (A_i \oplus B_i) C_i = G_i + P_i C_i$$

6.5

92

Look-ahead carry adder (cont'd)

The output of the *i*-th cell can be expressed as:

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i$$

$$C_{i+1} = A_i B_i + (A_i \oplus B_i) C_i = G_i + P_i C_i$$

Carry signals are evaluated as follows:

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

...

6.5

93

Look-ahead carry adder (cont'd)

The output

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

All carry signals depend on A and B signals, only, and can be easily generated by a combinational 2 level circuit (*carry look-ahead generator*)

Carry signals are evaluated as follows:

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

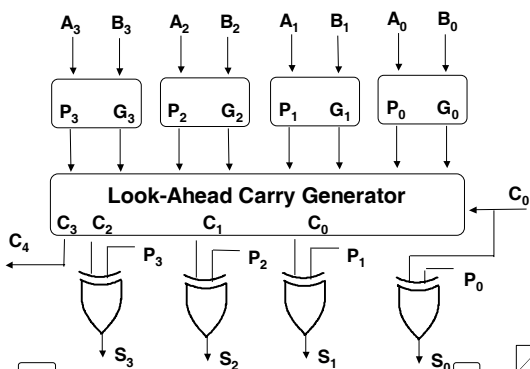
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

...

6.5

94

4-bit look-ahead carry adder



6.5

95

Overflow signal generation

The overflow signal OVFL is generated differently, according whether the internal carry signals are accessible, or not.

6.5

96

Overflow signal generation

The overflow signal OVFL is generated differently, according whether the internal carry signals are accessible, or not.

$$\begin{aligned}
 \text{OVFL} &= [A_{(n-1)} = B_{(n-1)}] \text{ and } [A_{(n-1)} \neq S_{(n-1)}] \\
 &= A_{(n-1)} B_{(n-1)} S_{(n-1)}' + A_{(n-1)}' B_{(n-1)}' S_{(n-1)}
 \end{aligned}$$

6.5

97

Overflow signal generation

The overflow signal OVFL is generated differently, according whether the internal carry signals are accessible, or not.

$$\text{OVFL} = C_{(n-1)} \oplus C_{(n-2)}$$

6.5

98

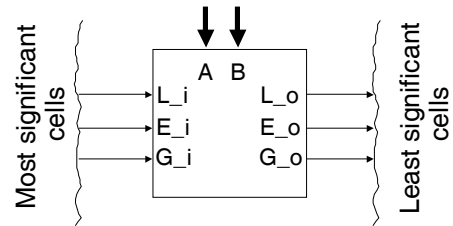
Comparators

When comparators are implemented following a *slicing* strategy, the information concerning the partial results can flow:

- from the most significant cells
- from the least significant cells.

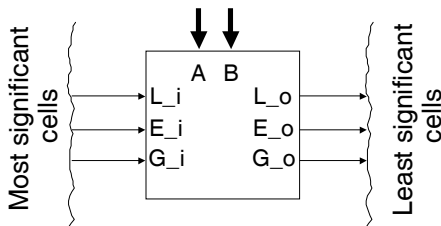
6.5

99



6.5

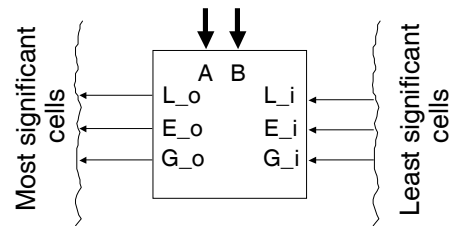
100



$$\begin{aligned}
 L_o &= L_i \vee (E_i \wedge (A < B)) \\
 E_o &= E_i \wedge (A = B) \\
 G_o &= G_i \vee (E_i \wedge (A > B))
 \end{aligned}$$

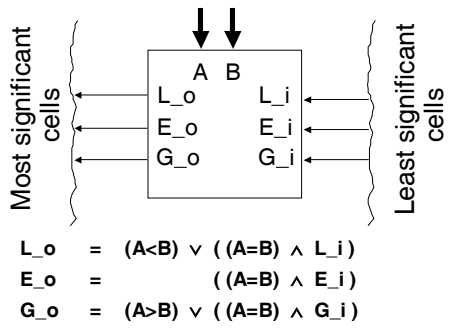
6.5

101



6.5

102



$$L_o = (A < B) \vee ((A = B) \wedge L_i)$$

$$E_o = ((A = B) \wedge E_i)$$

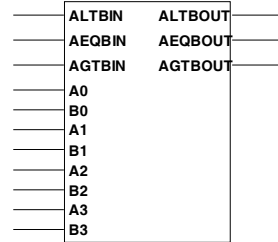
$$G_o = (A > B) \vee ((A = B) \wedge G_i)$$

6.5

103

Example

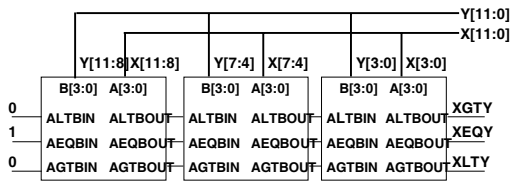
Design a 12-bit comparators, resorting to the following elementary block:



6.5

104

Solution



6.5

105

Comparator for 2C numbers

Are implemented by observing that:

$$A > B \Leftrightarrow (A + C) > (B + C)$$

where “?” is the target comparison operator.

When A and B are n bit numbers, one can choose:

$$C = 2^{n-1}$$

Then, since

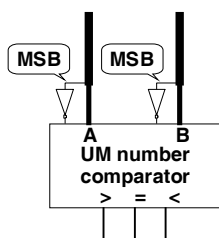
$$- 2^{n-1} \leq A \leq 2^{n-1} - 1$$

adding $C = 2^{n-1}$ is equivalent to complementing the most significant bit.

6.5

106

Solution

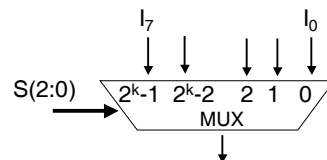


6.5

107

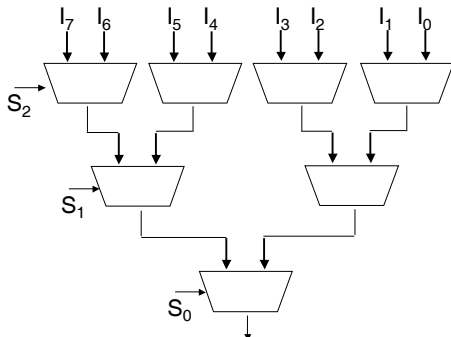
Multiplexers

How to implement an $8 \rightarrow 1$ mux by $2 \rightarrow 1$ mux's ?



6.5

108



6.5

109

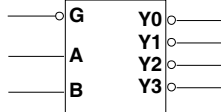
Decoders

How to implement a 3 → 8 decoder by the following 2 → 4 decoder?

6.5

110

Decoder 2 → 4

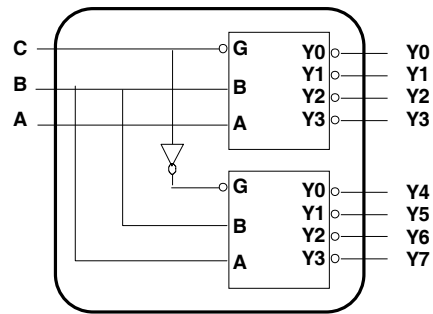


Inputs			Outputs			
G	B	A	Y3	Y2	Y1	Y0
1	X	X	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1

6.5

111

Solution: decoder 3 → 8

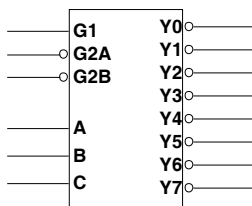


6.5

112

Exercise

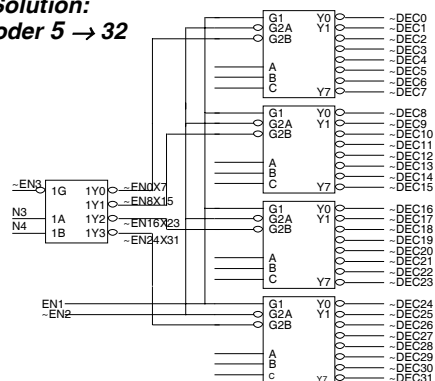
Design a 5 → 32 decoder, resorting to the following 3 → 8 decoder:



6.5

113

Solution: decoder 5 → 32



6.5

114

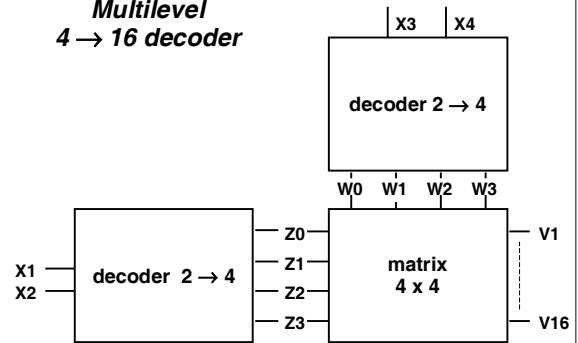
Multi level decoders

An $n \rightarrow 2^n$ decoder can be implemented as a single-level network, composed of 2^n n -inputs AND gates.
Cheaper implementations resort to multiple-level networks.

6.5

115

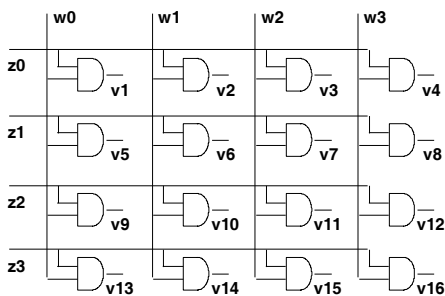
**Multilevel
4 → 16 decoder**



6.5

116

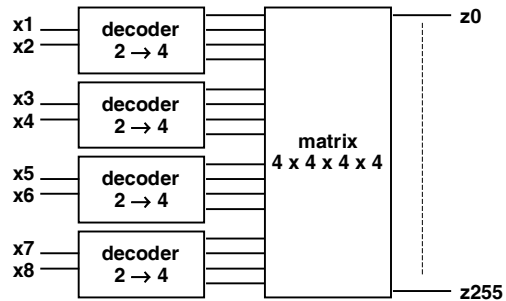
Matrix 4 x 4



6.5

117

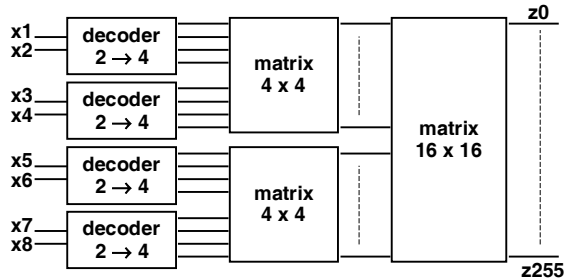
Decoder 8 → 256 (1st implementation)



6.5

118

Decoder 8 → 256 (2nd implementation)



6.5

119

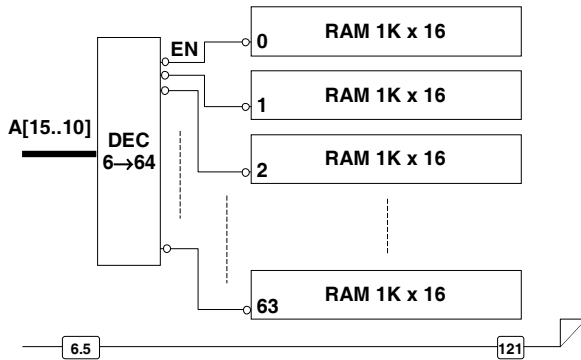
Exercise

Design a 64K x 16 RAM resorting to 1K x 4 RAM modules.

6.5

120

Solution



6.5

121

Multipliers

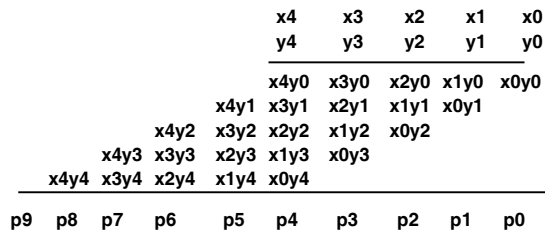
Two architectures will be considered:

- carry propagate
- based on elementary modules.

6.5

122

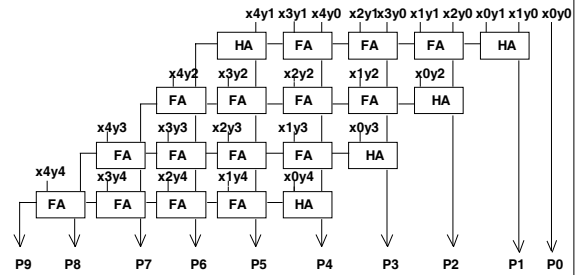
Product of two 5-bit binary numbers



6.5

123

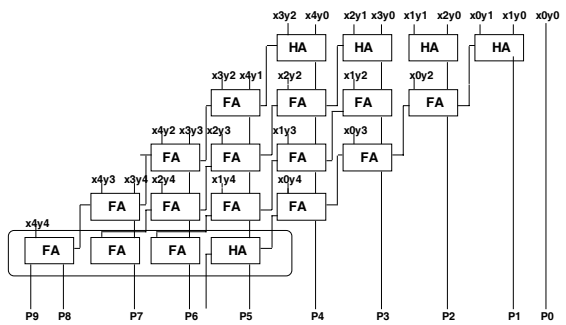
5-bit carry-propagate multiplier



6.5

124

Optimized implementation



6.5

125

“Composed” multipliers

To analyze the basic principle driving “composed” multipliers let’s consider an 8 × 4 multiplier, implemented by 4 × 2 multipliers.

6.5

126

