VHDL
Basic Language Concepts: Synthesis

EL 310
Erkay Savaş
Sabancı University

Preliminaries

- Synthesis is the process of analyzing a VHDL program and inferring digital circuit that implements the behavior implied by this VHDL description.
- The circuit is constructed using a fixed set of hardware primitives.
  - AND, OR, NOT,
  - XOR, XNOR
  - Multiplexor, decoder
- Efficient implementation
  - inference techniques and subsequent optimizations are specific to target hardware primitives.

Preliminaries

- The actual circuit also depends on the vendor-specific synthesis compiler
  - FPGA Express
  - Synplify Pro
  - Synopsys FPGA_Complier2 (FC2)
- Here, we are interested in hardware inference from concurrent signal assignment (CSA) statements.
- For the following class of CSA, we will show the inferred circuits for both gate level and FPGA
  - simple concurrent assignments
  - conditional concurrent assignments
  - selected concurrent assignments

Motivation

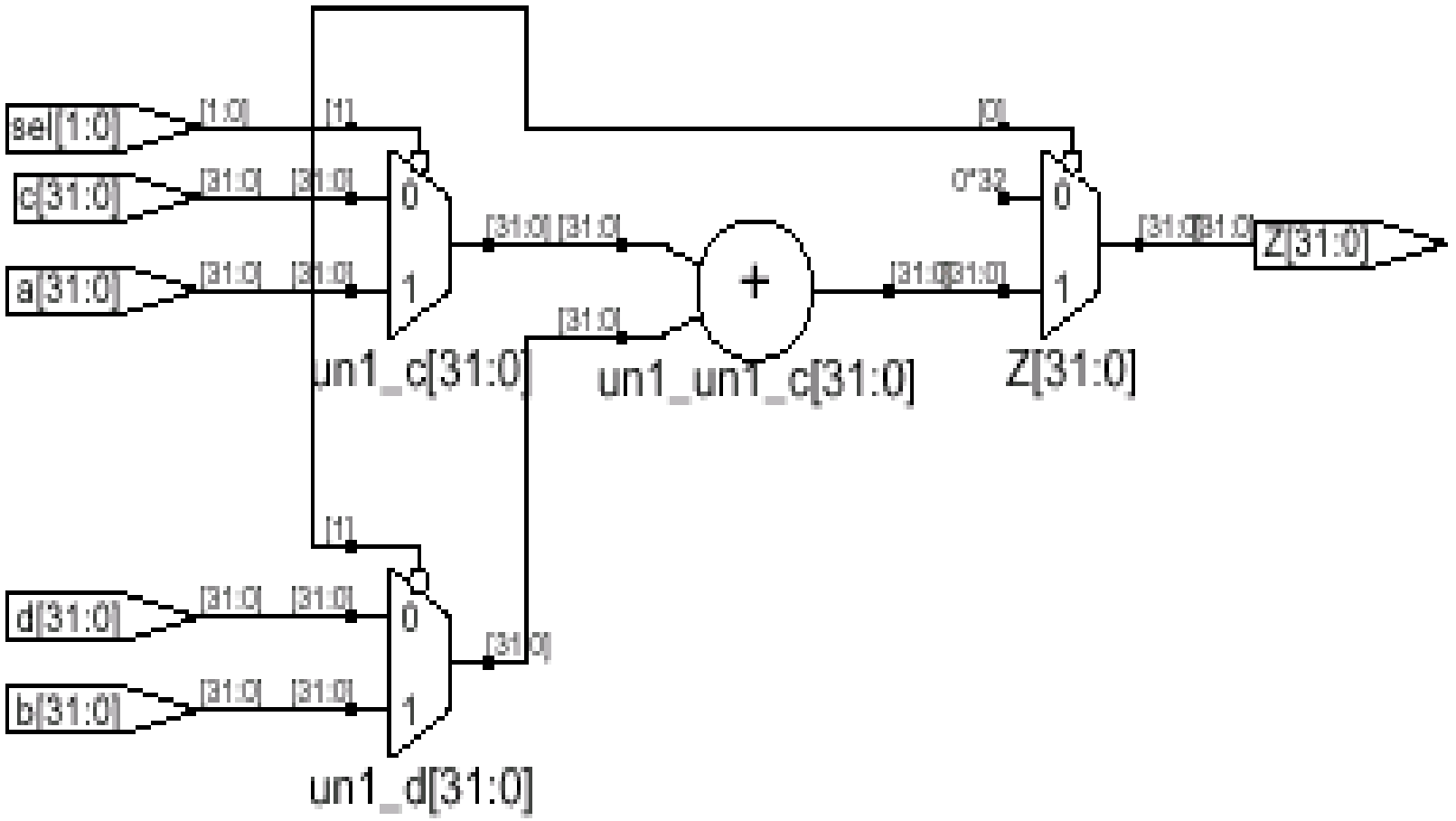
- To study each language constructs in order to anticipate the hardware that are going to be inferred for each construct.
  - There are more than one way to model a hardware component.
  - For example, how many bits are required to represent a signal?
  - Order in conditional and selected CSA will affect the inferred logic.
- If we can know what is going to be inferred from the code we write, we model our digital system in VHDL in such a way that the most effective circuit is produced.

Compiler's Job

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

entity synth is
port (a, b, c ,d: in integer;
      sel: in std_logic_vector(1 downto 0);
      Z: out integer);
end entity synth;

architecture behavioral of synth is
begin

with sel select
Z <= a + b when "00",
     c + d when "10",
     0 when others;
end architecture behavioral;
```

- We need adders; but how many? and how large?,
- how many bits to represent Z?
- Integers must be at least 32 bits.
- What if we deal with smaller integers?

How Compiler Synthesizes



Inference from Declarations 1

- In programming languages, variable and constant declarations mean a memory location
- Inference for signals
  - Wires, latches, and flip-flops
  - In combinational circuits, signals corresponds to wires connecting components.
  - In sequential circuits, flip-flops or latches are inferred for signals when they are stored.
  - Inference depends on the way we write the code
  - Goal is to avoid unnecessary storage elements.

Inference from Declarations 2

- Using integer type for a signal
  - no additional information is needed for simulation purposes
- However, in synthesis
  - compiler needs to know how large it gets.

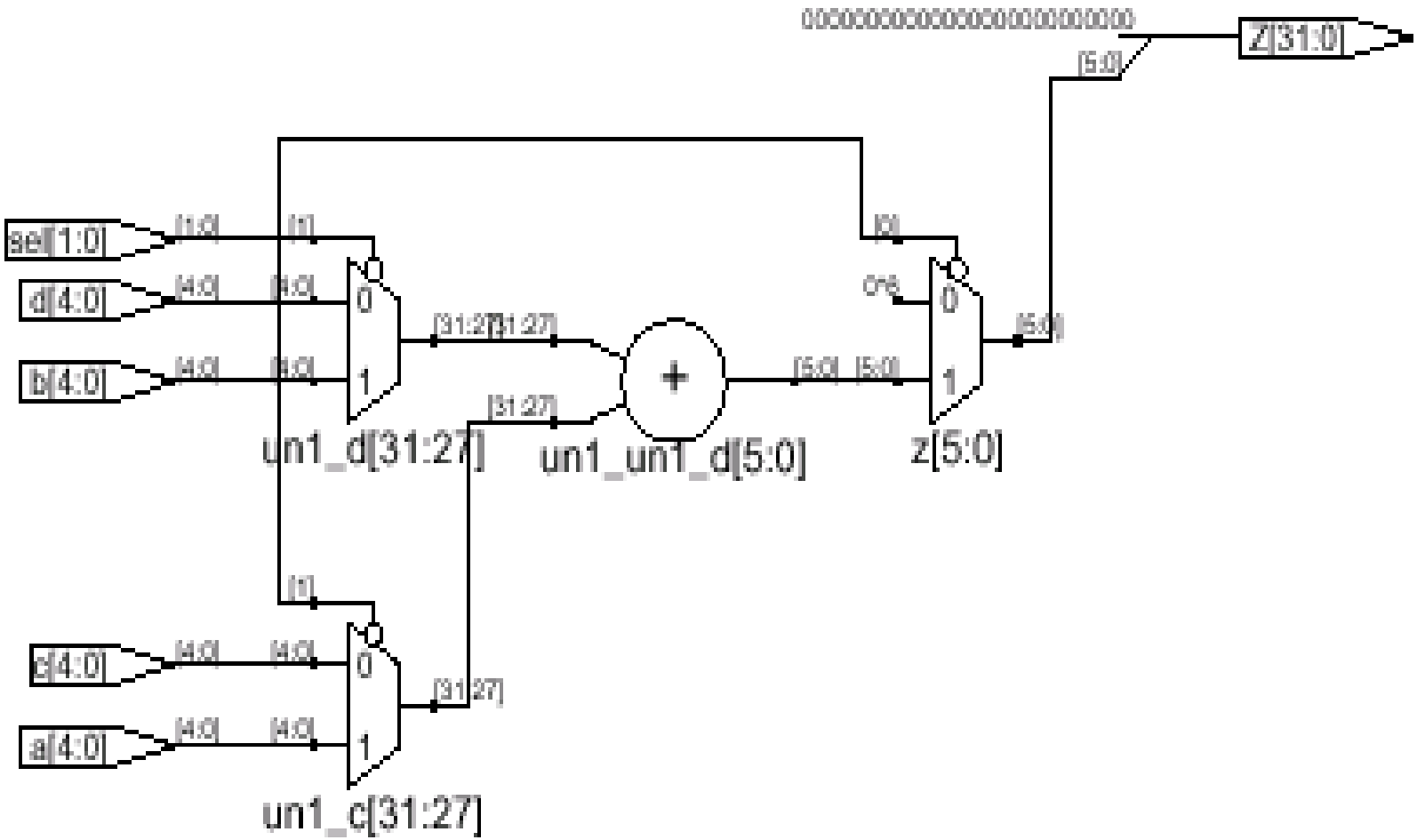
```
...
signal result: std_logic_vector(12 downto 0);
signal count: integer;
signal index: integer range 0 to 18;

type state_type is (state0, state1, state2, state3);
signal next_state:state_type;

...
```

Question: how many bits will a compiler use to implement signals "count" and "index"?

Limiting the Range of an Integer 1

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

entity synth is
port (a, b, c ,d: in integer range 0 to 18;
      sel: in std_logic_vector(1 downto 0);
      Z: out integer);
end entity synth;

architecture behavioral of synth is
begin

with sel select
Z <= a + b when “00”,
     c + d when “10”,
     0 when others;
end architecture behavioral;
```

Limiting the Range of an Integer 2



Inference from Declarations 3

- Compilers may try to optimize
  - even if we do not provide clues such as "range"
  - But never count on that!
- What about

```
type state_type is (state0, state1, state2, state3);
signal next_state:state_type;
```

- We define a new type (enumerated type)
  - A signal of the new type form can take at most 4 distinct values
  - Compiler needs only 2 bits to encode all the states.
  - We may want to have a control over the encoding (recall One-Hot State Assignment for FPGA)

Inference from Simple CSA

- Simple CSA corresponds to combinational logic
  - Value of a signal is computed from other signals
- Operator inferencing
  - Compiler infers logical operations from VHDL CSA statements
  - Utilize gate primitives to implement these operations
- Interconnection between gates are determined by the dependencies between signals
- The way compiler extract these dependencies are not always obvious

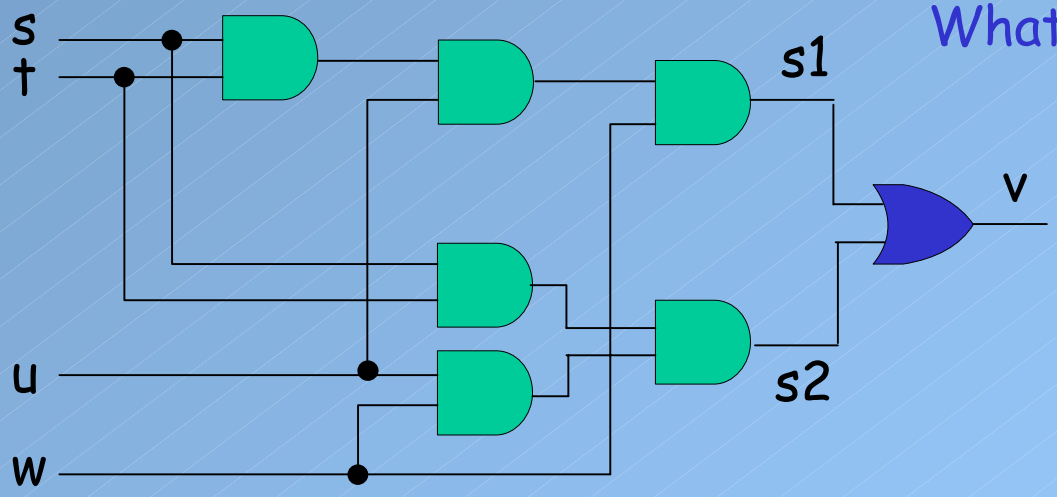
Operator Inferencing: Example

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

entity concurrent is
port(s,t,u,w: in std_logic;
     v: out std_logic);
end entity concurrent;

architecture dataflow of concurrent is
signal s1, s2: std_logic;
begin
L1: s1 <= s and t and u and w;
L2: s2 <= (s and t) and (u and w);
L3: v <= s1 or s2;
end architecture dataflow;
```

- Delay information is redundant for synthesis
- Post-synthesis simulation uses the known delay values of components
- Therefore, compilers usually ignores delay values specified in VHDL source.

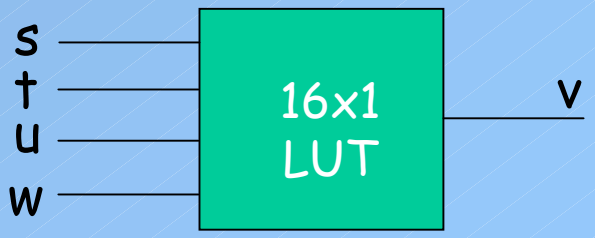
Operator Inferencing: Example

- L1 and L2 are functionally equivalent
- However, they lead to different implementations
- Both implementation have three AND gates.
- However, L2 leads to a faster circuit. Why?
- L2 exploits the concurrency in logical expression.
- Use of parentheses may be useful.



synthesized gate-level implementation

What is the precedence for s1?



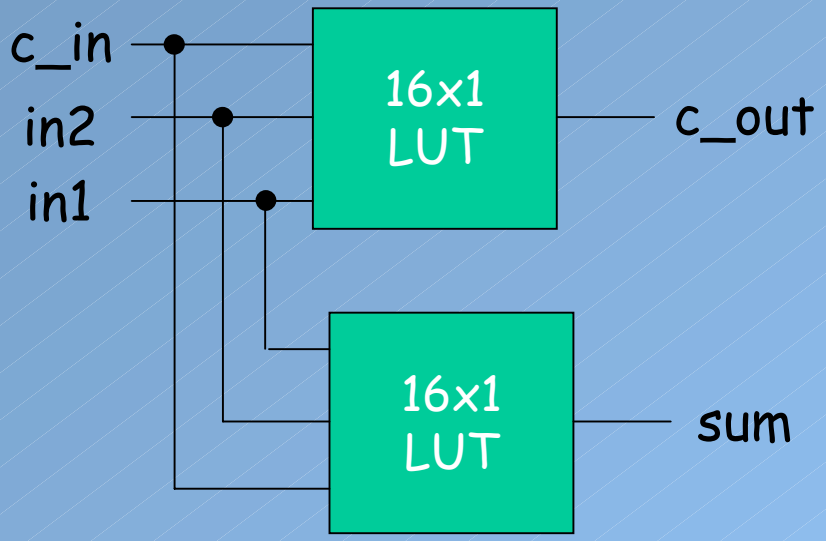
FPGA implementation
Xilinx XC4000E

Full Adder 1

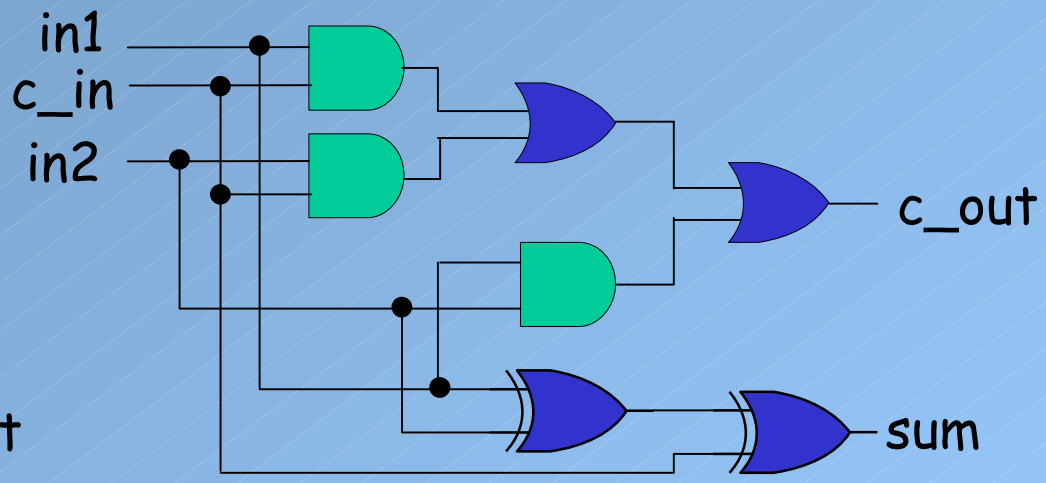
```
library IEEE;
use IEEE.std_logic_1164.all;

entity full_adder is
port(in1,in2,c_in: in std_logic;
     sum, c_out: out std_logic);
end entity full_adder;

architecture dataflow of full_adder is
signal s1, s2, s3: std_logic;
begin
sum <= in1 xor in2 xor c_in;
c_out <= (in1 and in2) or (in1 and c_in) or (in2 and c_in);
end architecture dataflow;
```

- we need two Xilinx XC4000E series LUTs to implement two Boolean functions with three variables (i.e. sum and c_out)

Full Adder 1



FPGA implementation
Xilinx XC4000E



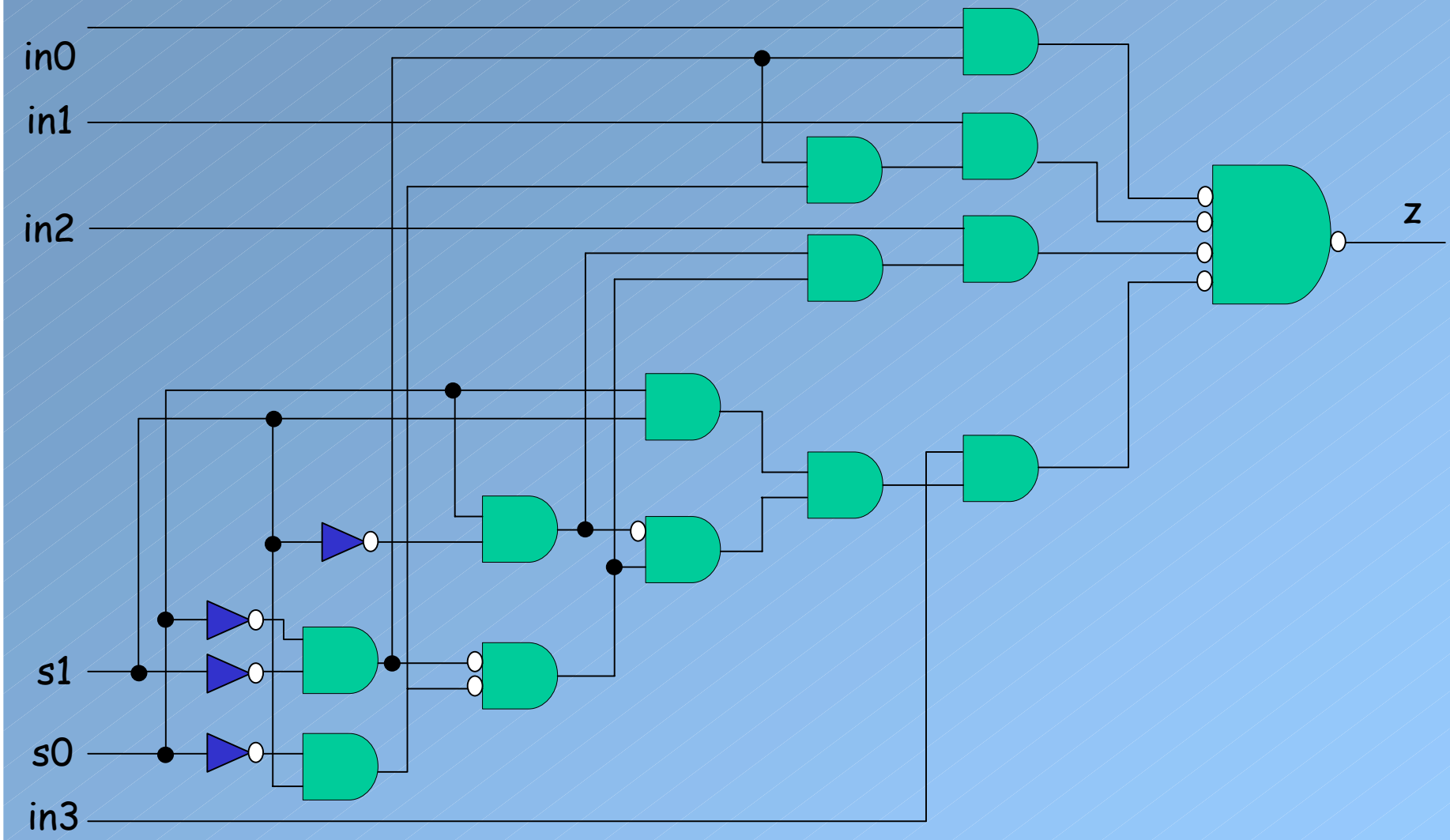
synthesized gate-level implementation

Inference from Conditional Signal Assignment

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

entity mux4 is
port(in0, in1, in2, in3: in std_logic;
     s0, s1: in std_logic;
     z: out std_logic);
end entity mux4;

architecture behavioral of mux4 is
begin
z <= in0 when s0 = '0' and s1 = '0' else
     in1 when s0 = '0' and s1 = '1' else
     in2 when s0 = '1' and s1 = '0' else
     in3 when s0 = '1' and s1 = '1' else
     '0';
end architecture behavioral;
```

- Purely combinational
- Priority order implied must be preserved in the text when the conditions are not mutually exclusive.
- When conditions are mutually exclusive, synthesis compiler usually eliminates the priority logic

MUX4

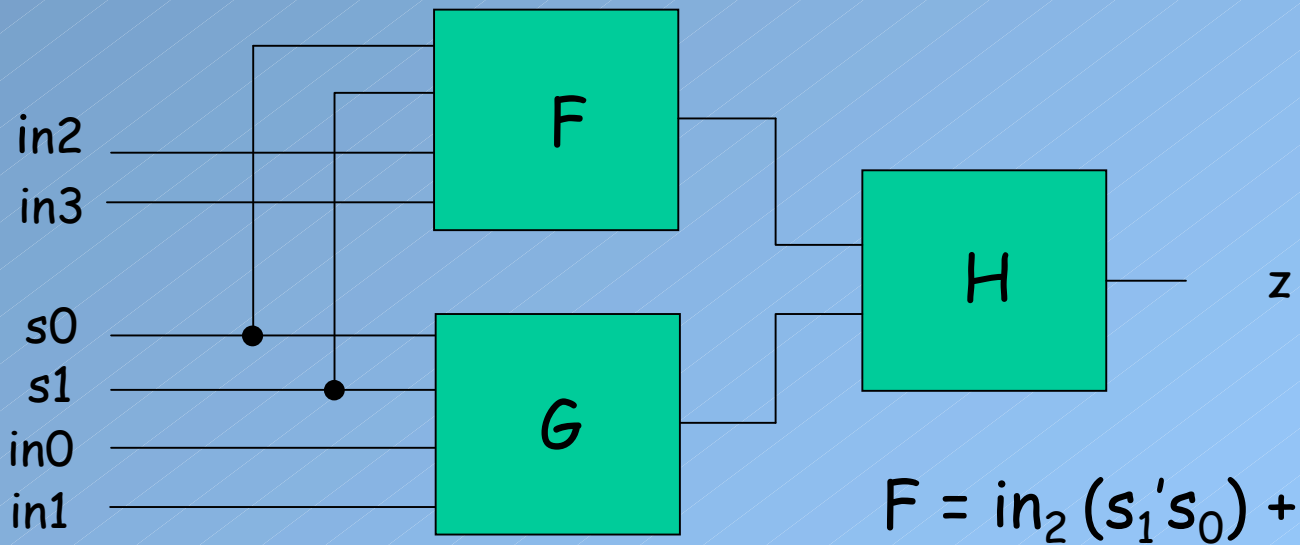


Synthesized Gate-Level Implementation
(priority encoding preserved)

MUX4

- It implements

$$z = in_0 (s_1's_0') + in_1 (s_1s_0') (s_1's_0')' + in_2 (s_1's_0) (s_1s_0')'(s_1's_0')' + in_3 (s_1s_0) (s_1's_0)' (s_1s_0')' (s_1's_0')'$$

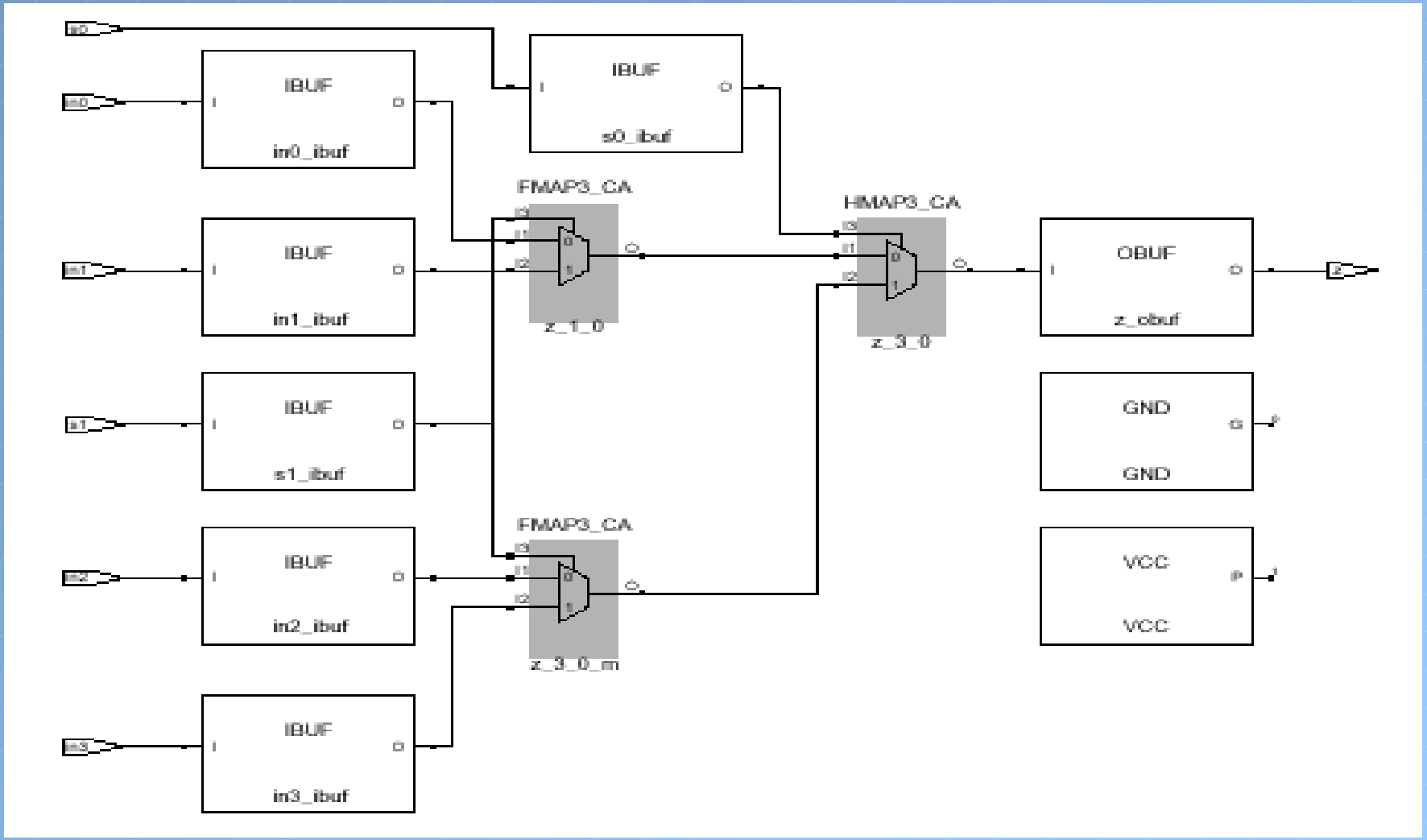


FPGA implementation
Xilinx XC4000E

It eliminates the priority logic

$$F = in_2 (s_1's_0) + in_3 (s_1s_0)$$
$$G = in_0 (s_1's_0') + in_1 (s_1s_0')$$
$$H = F + G$$

MUX4



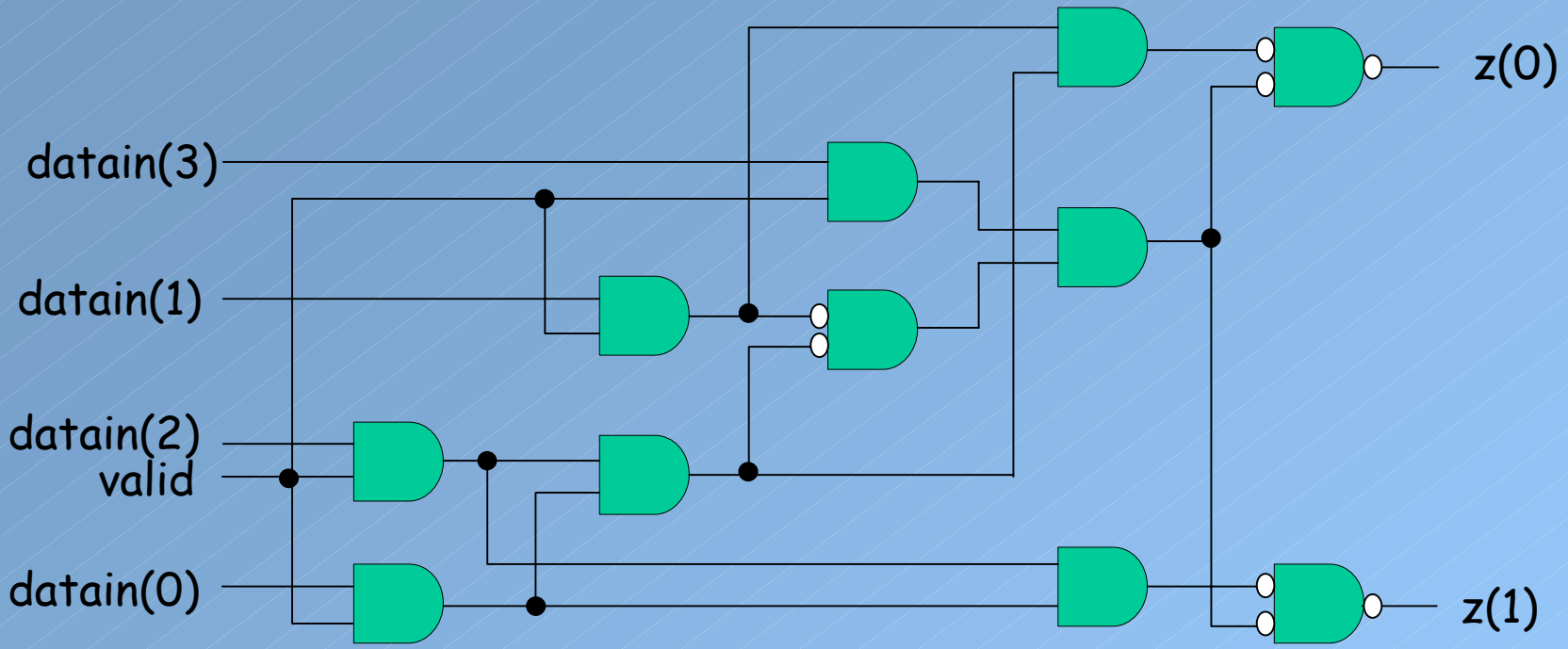
No priority logic indeed!

Priority Encoder

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

entity priority is
port(datain: in std_logic_vector(3 downto 0);
     valid: in std_logic;
     z: out std_logic_vector(1 downto 0));
end entity priority;

architecture behavioral of priority is
begin
z <= "00" when datain(0) = '1' and valid = '1' else
     "10" when datain(2) = '1' and valid = '1' else
     "01" when datain(1) = '1' and valid = '1' else
     "11" when datain(3) = '1' and valid = '1' else
     "00";
end architecture behavioral;
```

- Not mutually exclusive anymore.
- Priority encoding must be implemented.
- Conditional CSA presents very natural construct for describing a priority logic.

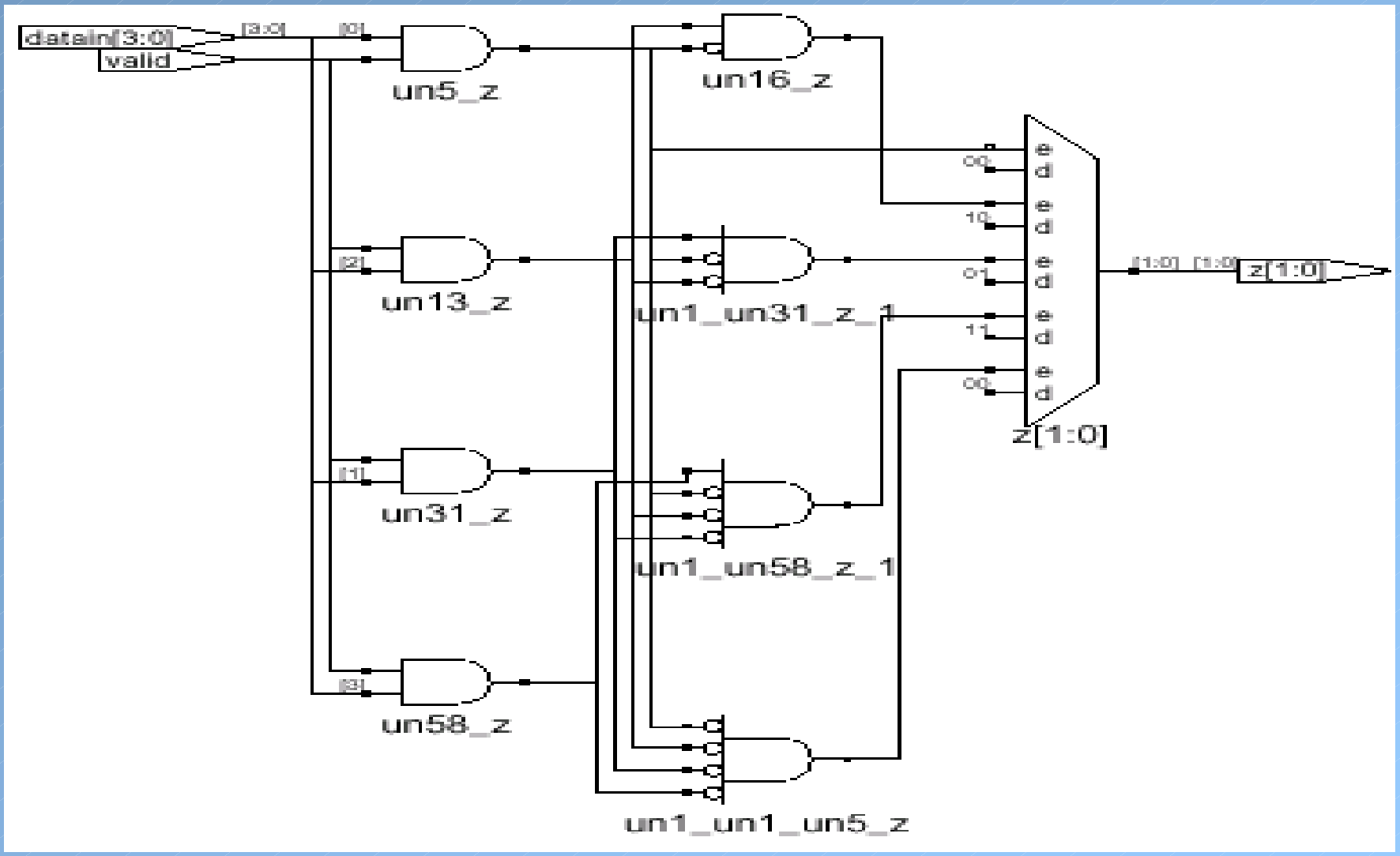
Priority Encoder



z(0) = [valid·datain(1)] [valid·datain(0)]' [valid·datain(2)]' +
[valid·datain(3)] [valid·datain(1)]'[valid·datain(0)]' [valid·datain(2)]'

z(1) = [valid·datain(1)] [valid·datain(0)]' +
[valid·datain(3)] [valid·datain(1)]'[valid·datain(0)]' [valid·datain(2)]'

Priority Encoder



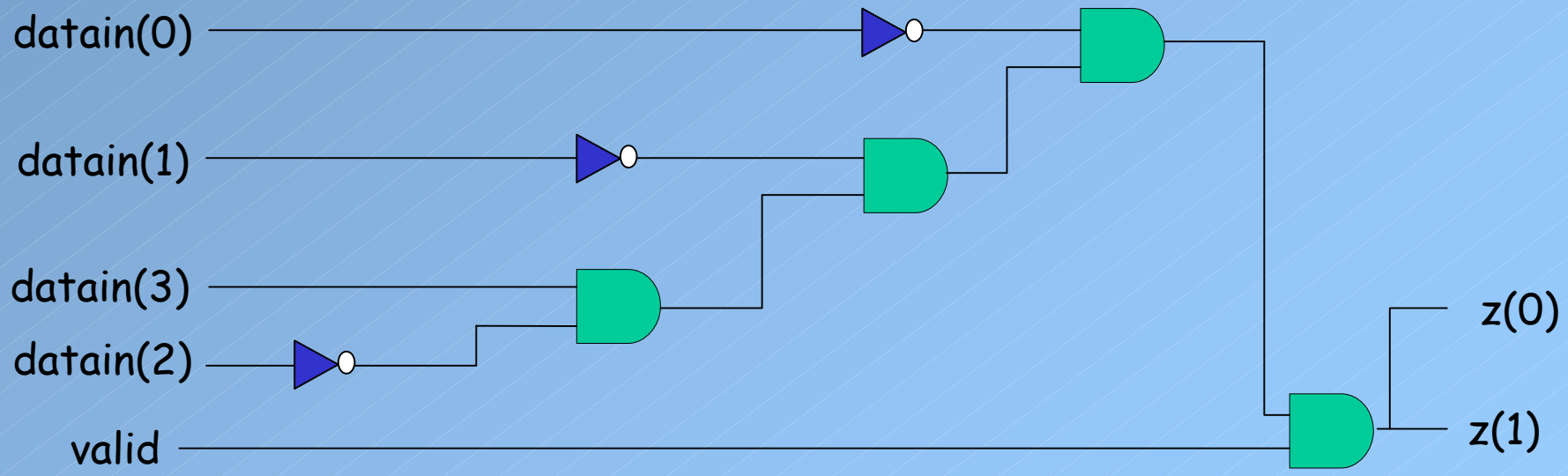
How Don't Cares is Synthesized

- Don't care values cannot be represented in hardware and cannot be compared against 0 or 1.

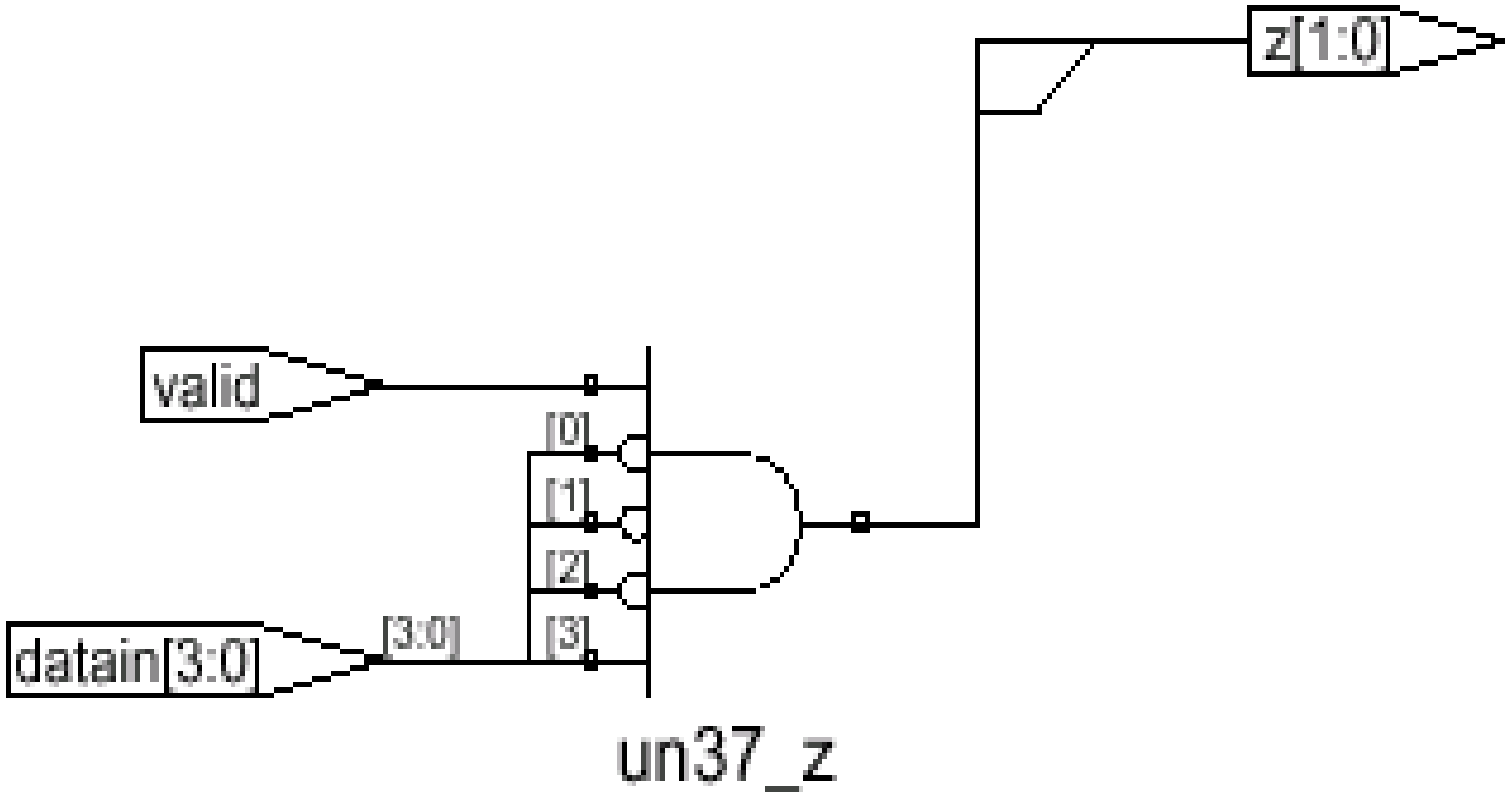
```
library IEEE;
use IEEE.std_logic_1164.all;

entity priority is
port(datain: in std_logic_vector(3 downto 0);
     valid: in std_logic;
     z: out std_logic_vector(1 downto 0));
end entity priority;

architecture behavioral of priority is
begin
z <= "00" when datain = "---1" and valid = '1' else
     "10" when datain = "-100" and valid = '1' else
     "01" when datain = "--10" and valid = '1' else
     "11" when datain = "1000" and valid = '1' else
     "00";
end architecture behavioral;
```

Don't Cares Cannot be Synthesized

- In synthesis, if one of the signals in comparison operation is a don't care, then the comparison always returns FALSE!
  - In the previous example, the output will have value of either 11 or 00.



Don’t Cares is Synthesized Incorrectly



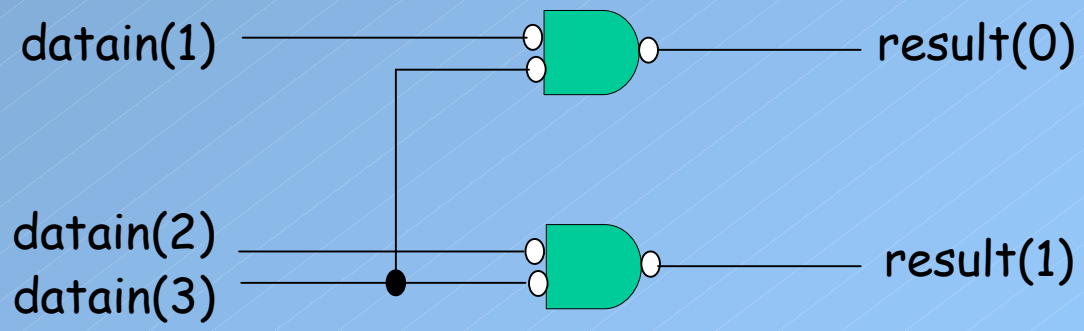
Inference from Selected CSA

- In the selected signal assignment there is no priority ordering among the options
- Consider a 4-to-2 encoder

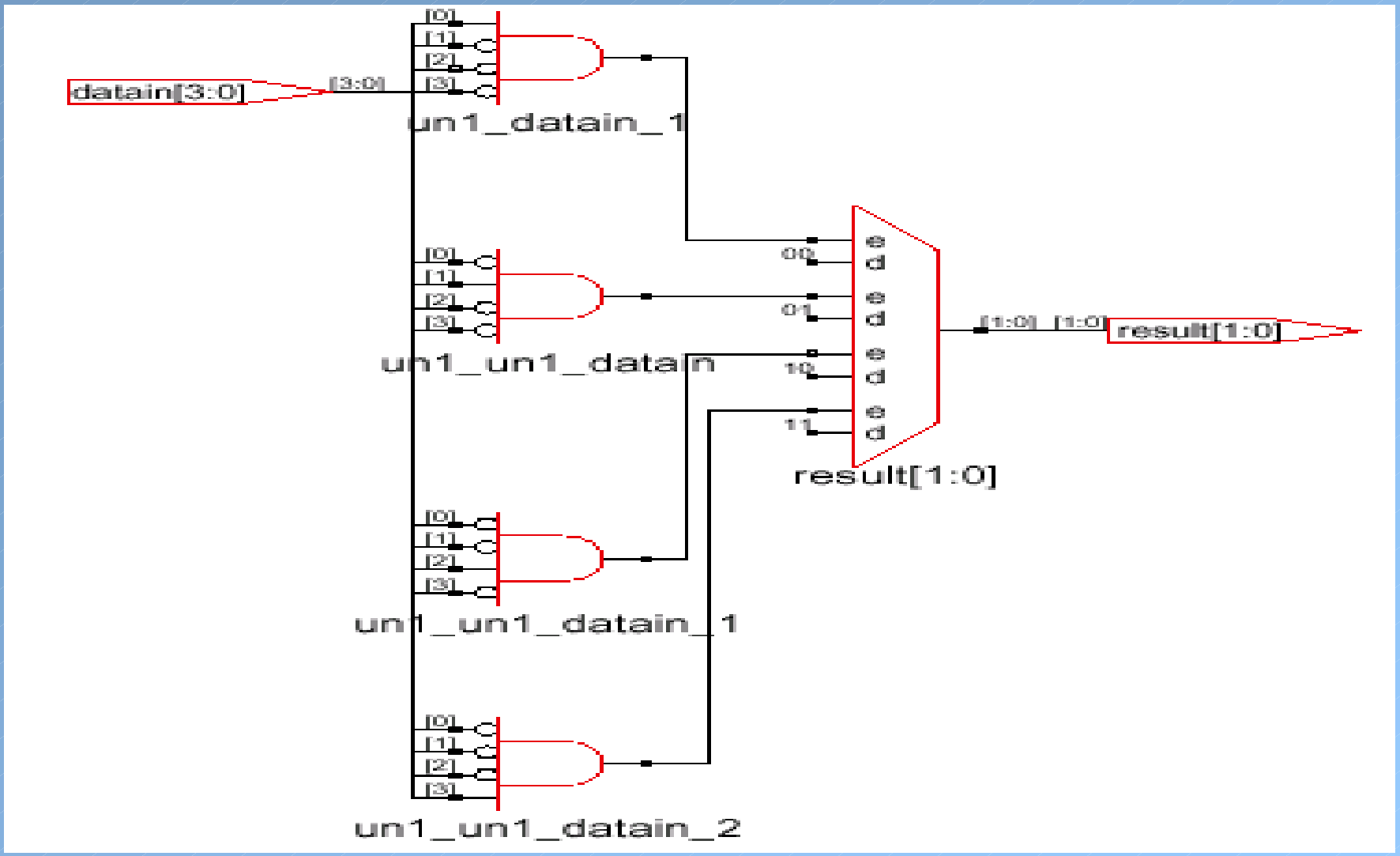
```
library IEEE;
use IEEE.std_logic_1164.all;

entity encoder is
port(datain: in std_logic_vector(3 downto 0);
     result: out std_logic_vector(1 downto 0));
end entity encoder;

architecture behavioral of encoder is
begin
with datain select
result <= "00" when "0001",
          "01" when "0010",
          "10" when "0100",
          "11" when "1000",
          "XX" when others; -- outputs are undefined
end architecture behavioral;
```

A simple 4-to-2 Encoder

- At most one of the input can have value 1 at any given time, otherwise the output of the circuit is undefined
  - result(0) is 1 when datain(1) or datain(3) is 1
  - result(1) is 1 when datain(2) or datain(3) is 1
- This encoder can be realized using two LUTs in a CLB.



A simple 4-to-2 Encoder



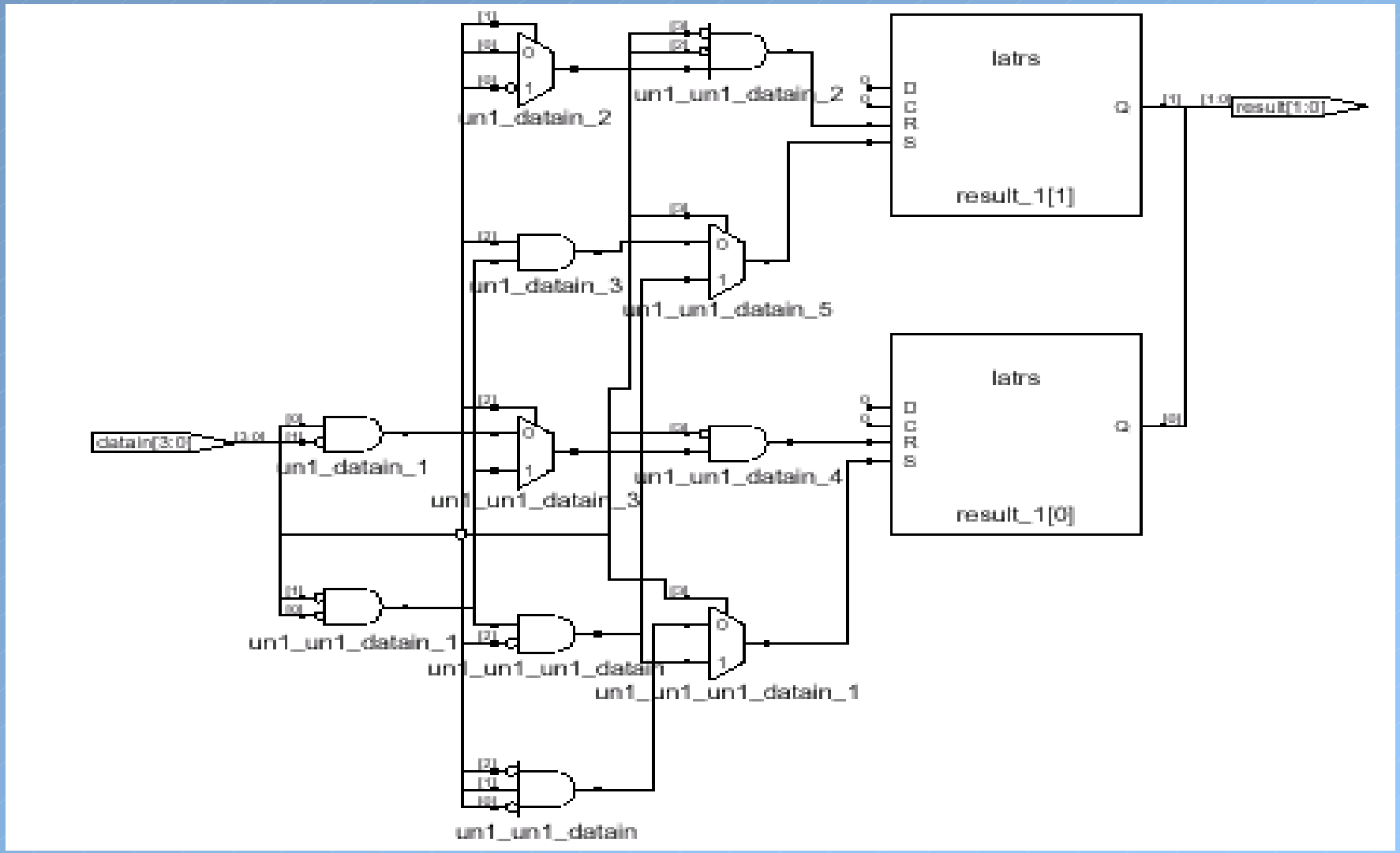
The Keyword unaffected

- unaffected keyword can be used within the select expression (VHDL'87 does not support)
  - Be aware that a latch will be inferred when the keyword unaffected is in the select expression.

```
with datain select
result <= "00" when "0001",
          "01" when "0010",
          "10" when "0100",
          "11" when "1000",
          unaffected when others;
```

  - It has been reported that some synthesis compilers do not support unaffected keyword.

Synthesis with `unaffected` *Keyword*

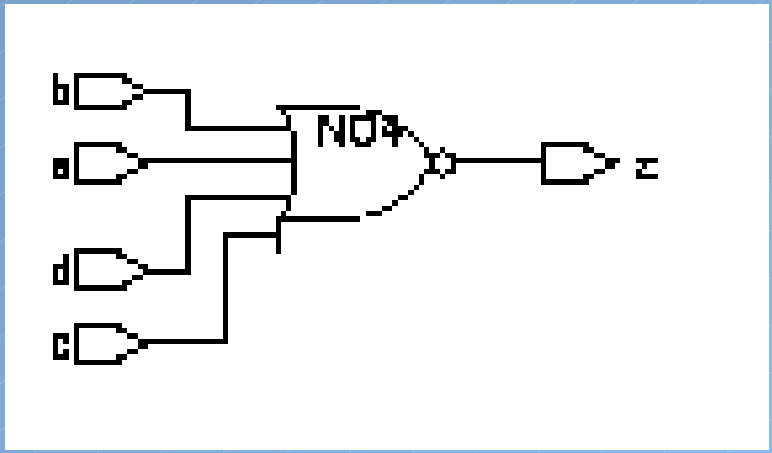


Synthesis of the Resolved Signals

```
library IEEE;
use IEEE.std_logic_1164.all;
entity reso1 is
port (a, b, c, d: in std_logic; Z: out std_logic);
end entity reso1;
architecture behavioral of reso1 is
begin
Z <= not a;
Z <= not b;
Z <= not c;
Z <= not d;
end architecture behavioral;
```

```
library IEEE;
use IEEE.std_logic_1164.all;
entity reso2 is
port (a, b, c, d: in std_logic; Z: out std_logic);
end entity reso2;
architecture behavioral of reso2 is
begin
Z <= a;
Z <= b;
Z <= c;
Z <= d;
end architecture behavioral;
```

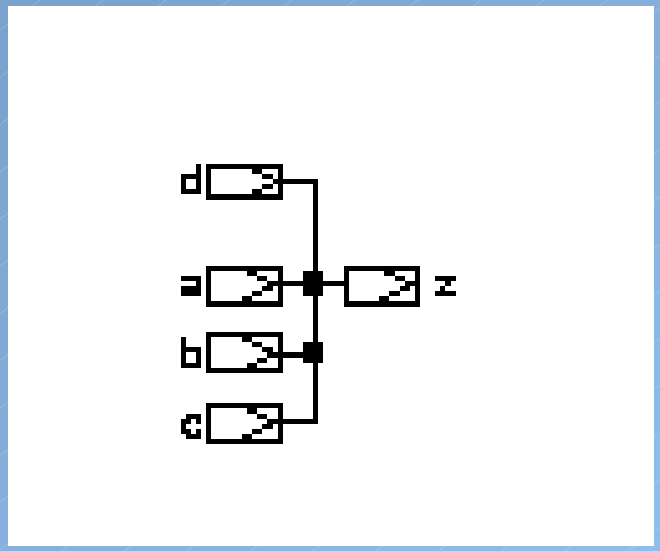
Synthesis of the Resolved Signals



design: resolu	designer: Erkay Savas	date: 10/22/2003
technology: cmx_HRDLIB	company: Sabanci University	sheet: 1 of 1

- Entity reso1

Synthesis of the Resolved Signals



design: resolu	designer: Erkay Savas	date: 10/22/2003
technology:	company: Sabanci University	sheet: 1 of 1

- Entity reso2

Simulation Behavior vs. Synthesis Behavior

- Semantic mismatch
  - Simulation model of the VHDL model and the simulation of the synthesized hardware may not produce an identical behavior

1. Delay Statements: In simulation model we can use any delay value for the components. In synthesis, those values are derived from the specifications of the real components.
2. Comparison Logic: Comparisons to don't cares(-), high impedance (Z), or other literals do not have meaningful hardware counterparts. Equality tests to other than 1/0 values return always FALSE for synthesis. However, recall that we can assign values such as Z or U to the signals in simulation.

Synthesis Hints

1. Do not specify initial values in your declaration of signals.
   - Most synthesis compilers ignore them. If you have to, do it using explicitly under the control of (probably asynchronous) reset signal. Constants can be initialized.
2. Specify the number of the bits necessary for a signal explicitly in a declaration
   - This will tell the compiler exactly how many bits a signal would need. This will lead to less hardware in the form of the widths of signal paths, the number of gates necessary to process the signals, and the number of latches or flip-flops to store signal values.

Synthesis Hints

3. Use of **unaffected** keyword in branches of signal assignments may cause latches
4. Using **don't care** values to cover **when others** case in a selected signal assignments can enable synthesis compiler to optimize the logic and create a smaller logic. Use it consciously, though.
5. Use parentheses to control concurrency that may affect the speed of the logic
6. Use of selected signal assignments will produce less logic since priority among the options is not implied.

Summary

- Process of hardware inference from basic CSA.
  - Every construct has a semantic as to how a construct is synthesized to logic.
  - Simple and conditional signal assignments will lead to the synthesis of combinational circuit.
  - unaffected clause in selected signal assignment can lead to synthesis of a latch since the output value must be "remembered" across execution.
- Inferring signal data widths
- Operator inferencing
- Use of parentheses to control precedence and structure of synthesized circuit.

Summary

- Inference of combinational logic
- Inference of latches in selected signal assignment statements.
- Mismatches between the behavior of the simulation of VHDL code and simulation of the synthesized circuit.