

Inspired on course 6.111 from MIT
Some support material taken from 6.111 (thank you)

4×(2T+3L)	Introdução com exercícios
3×5L	Trabalho #1
7×5L	Projecto Final

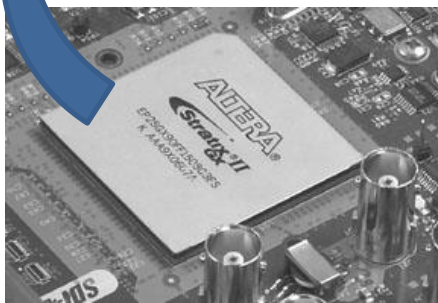
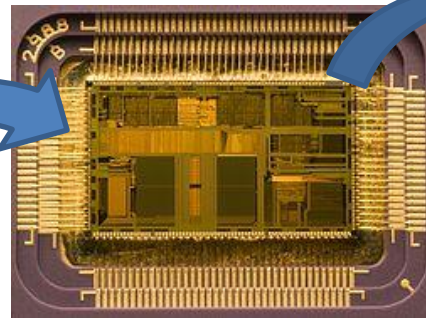
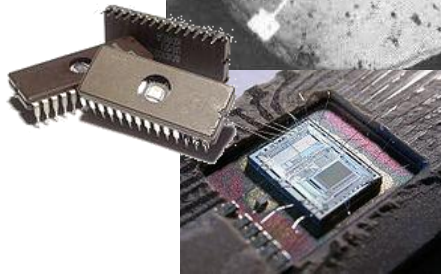
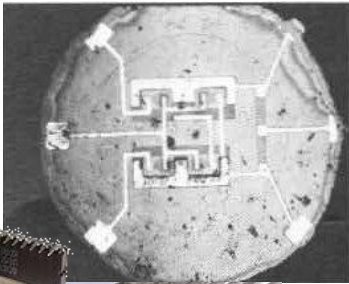
- Introdução
 - FPGAs
 - Laboratório
 - Verilog
 - Lógica combinatória
 - Lógica sequencial
- Máquinas estado
- Advanced verilog (video, etc.)

Refs:

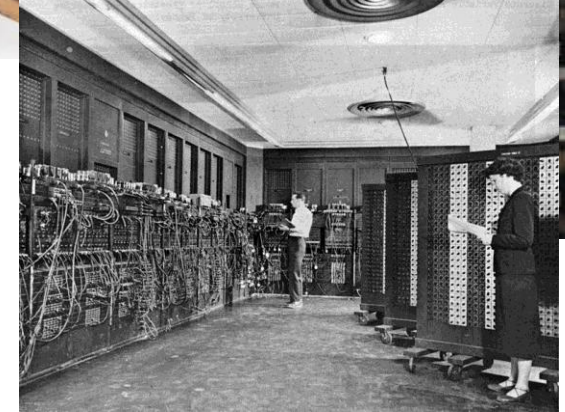
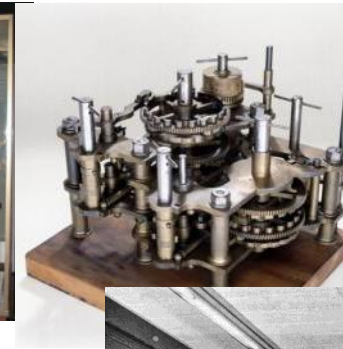
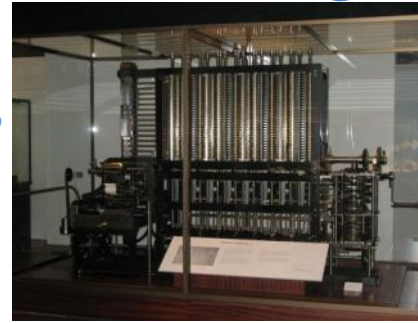
Cyclone II device Handbook, Altera corp.
Quartus II Handbook , Altera corp.
DE2 documentation
Verilog HDL, S. Palnitkar, Prentice Hall

Lógica Digital

Programação



Babbage
difference engine



Lógica Digital de Programação

Tudo funciona ou por magia ou com gnomos...

Programação de microprocessadores (assembly, c++, etc.)

Dar ao “gnomo” uma lista
(consecutiva) de operações a
realizar

1 Máquina executa várias tarefas
consecutivas



Programação de Lógica Digital (FPGAs+HDL)

Dar ao gnomo uma lista de
“objectos” para construir

Várias Máquinas executam várias
tarefas em paralelo



Em linguagens de alto nível por vezes confundem-se



2+2?
Resultado x 2?

A=2+2
Cout << A
A=A*2
Cout << A

A=4
4!
A=8
8!

tempo

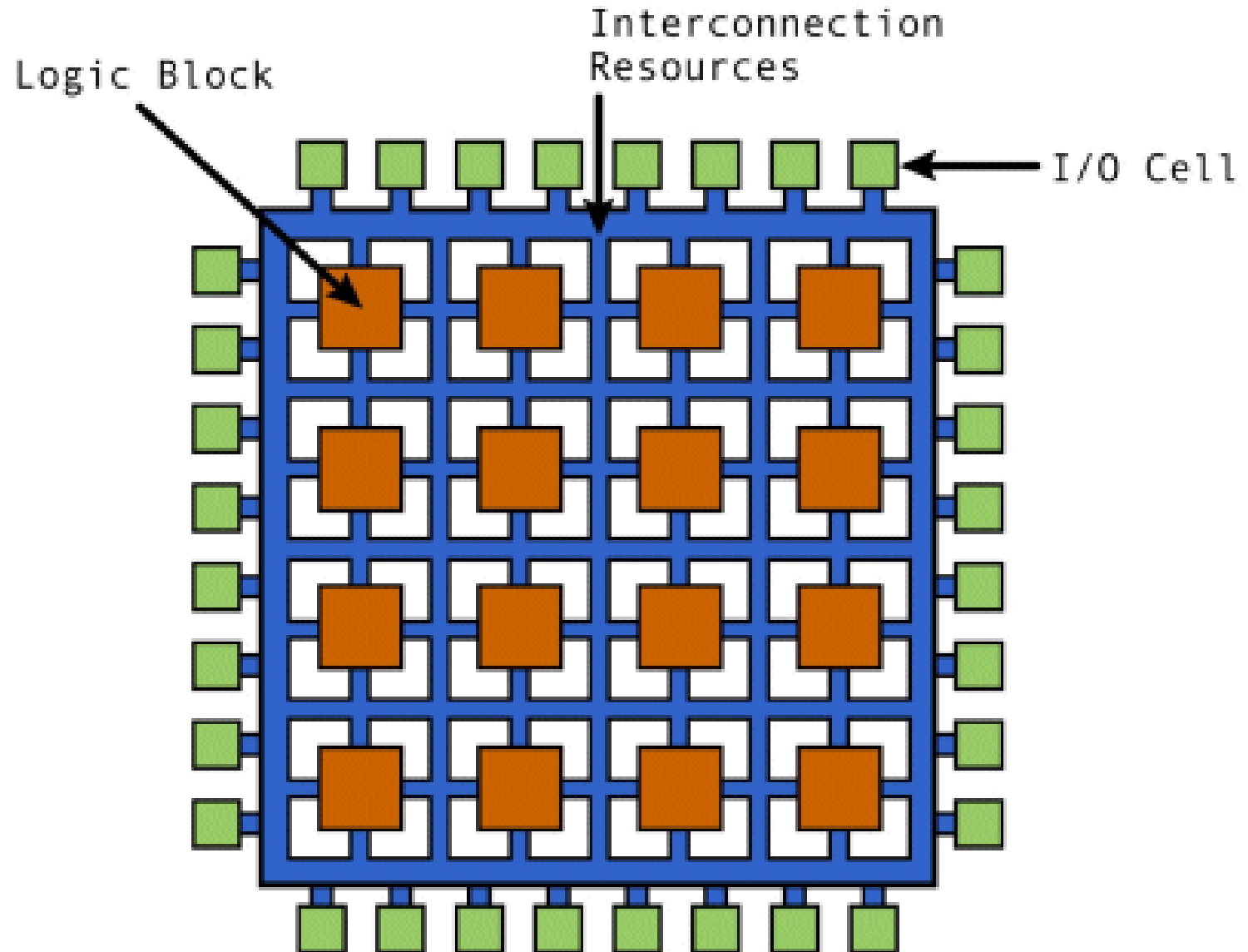
A=2+2
Output A
A=A*2
Output A

Somador: A=4
“ao mesmo tempo”
Multiplicador: A=A*2
Saída ??? (glitches?)

FPGA

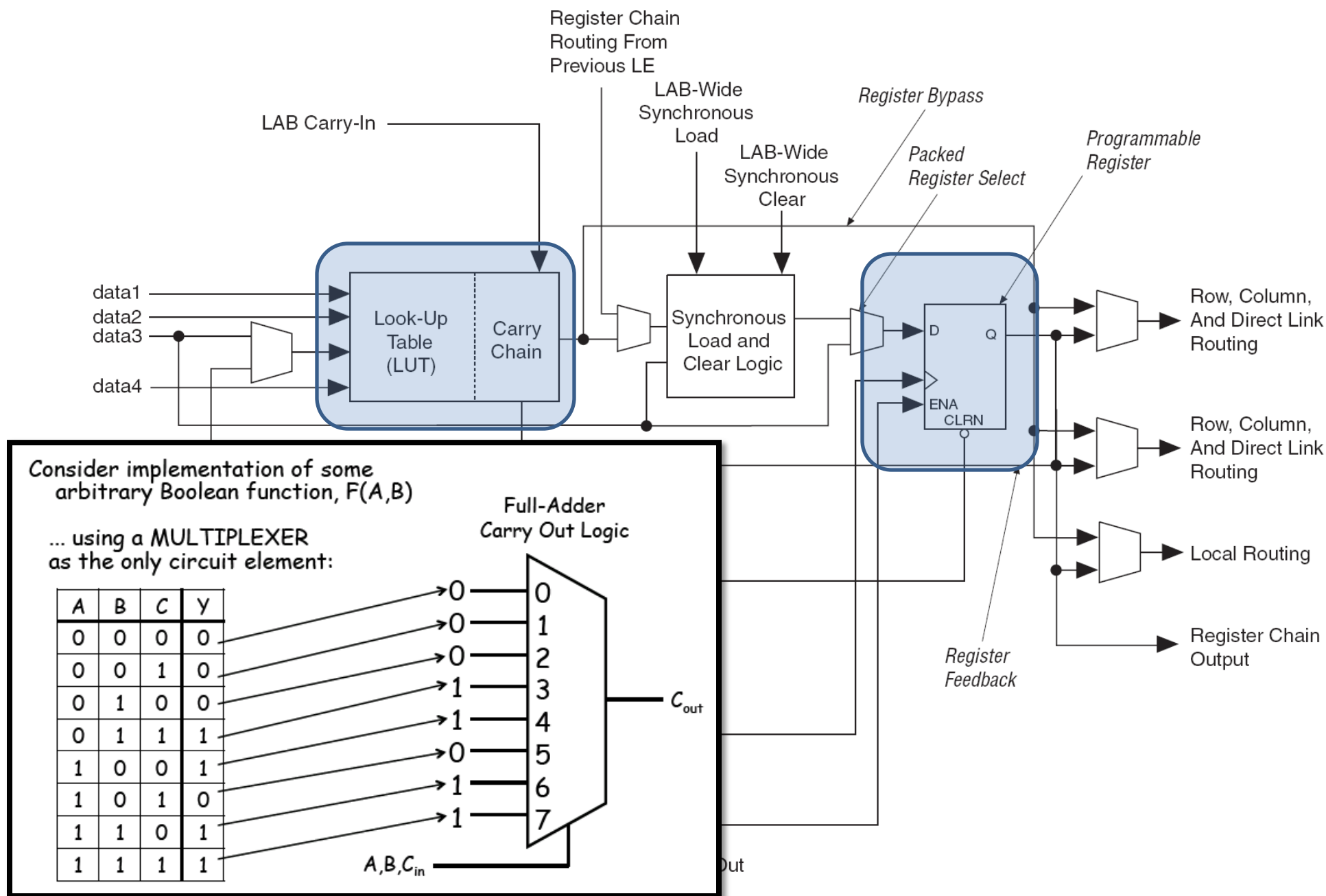
Field Programmable Gate Array:
Set of Chips in a bread board.

Control:
Chips functions
Connections



Logic elements

Figure 2-2. Cyclone II LE



A Tale of Two HDLs

VHDL

ADA-like verbose syntax, lots of redundancy (which can be good!)

Extensible types and simulation engine. Logic representations are not built in and have evolved with time (IEEE-1164).

Design is composed of entities each of which can have multiple architectures. A configuration chooses what architecture is used for a given instance of an entity.

Behavioral, dataflow and structural modeling.
Synthesizable subset...

Harder to learn and use, not technology-specific, DoD mandate

Verilog

C-like concise syntax

Built-in types and logic representations. Oddly, this led to slightly incompatible simulators from different vendors.

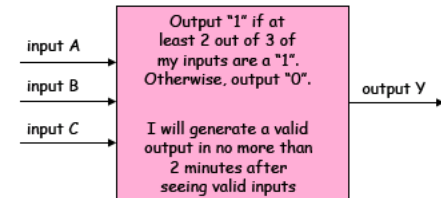
Design is composed of modules.

Behavioral, dataflow and structural modeling.
Synthesizable subset...

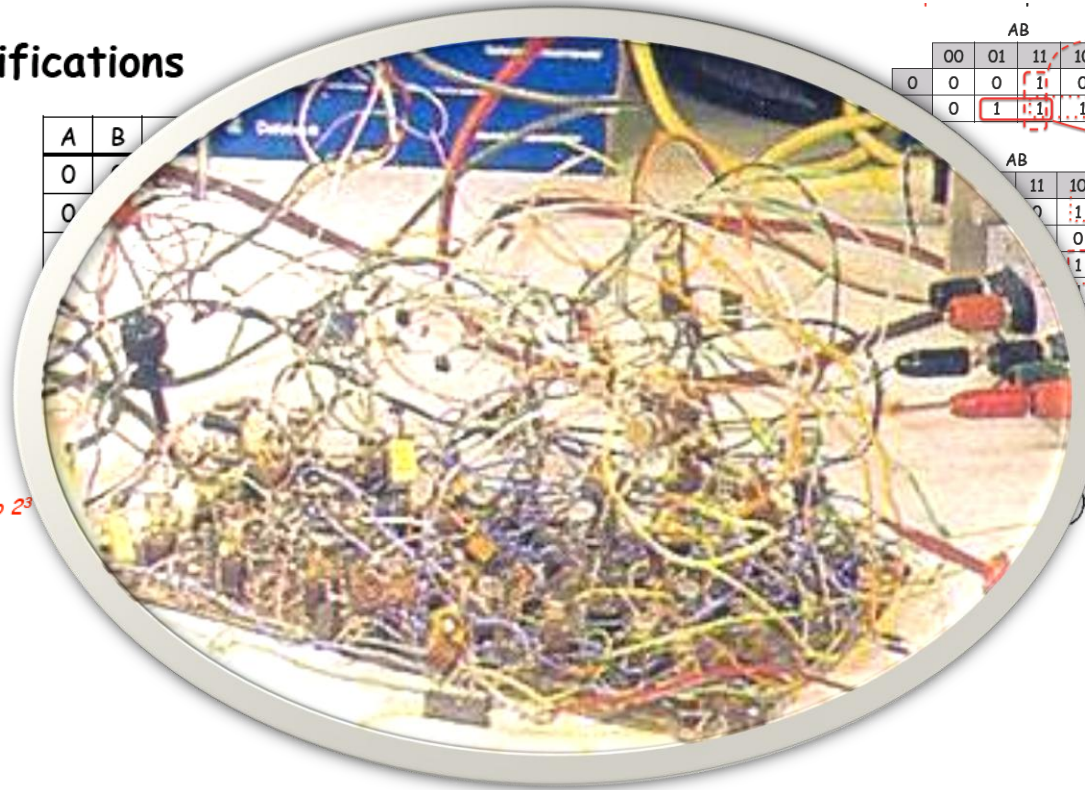
Easy to learn and use, fast simulation, good for hardware design

74LS.... vs FPGAs

Functional Specifications



so 2³



AB		00	01	11	10
0	0	0	0	1	0
0	1	0	1	1	1

$$Y = A \cdot C + B \cdot C + A \cdot B$$

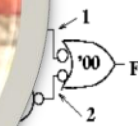
We're done!



AB		11	10
0	0	1	0
1	0	0	1

$$Z = \bar{B} \cdot \bar{D} + \bar{B} \cdot C + \bar{A} \cdot C$$

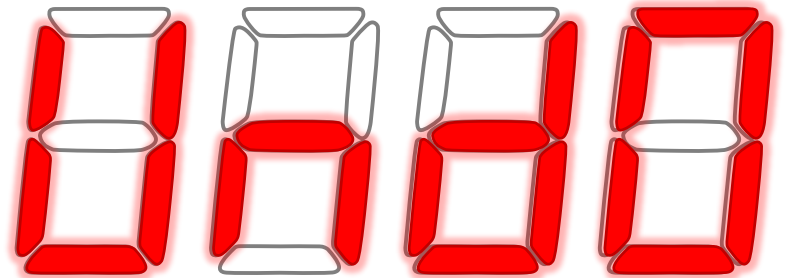
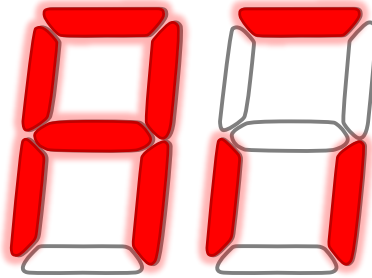
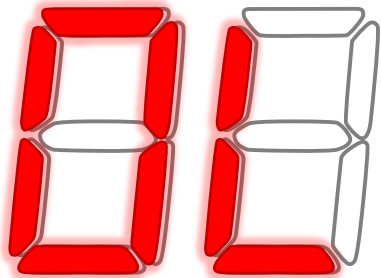
gates:



+

HDL
code

~~HELLO WORLD~~



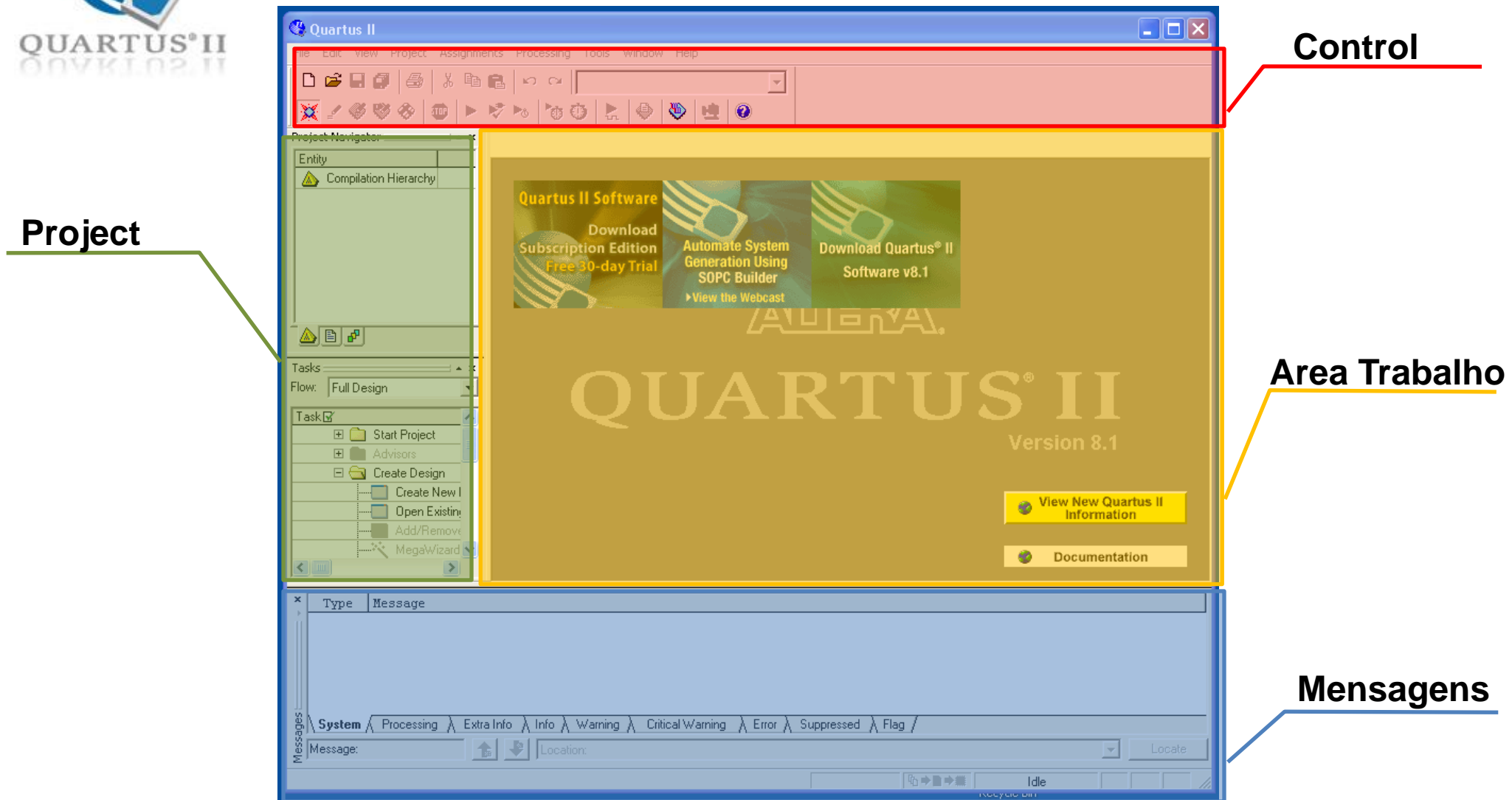
Primeiro exercício...

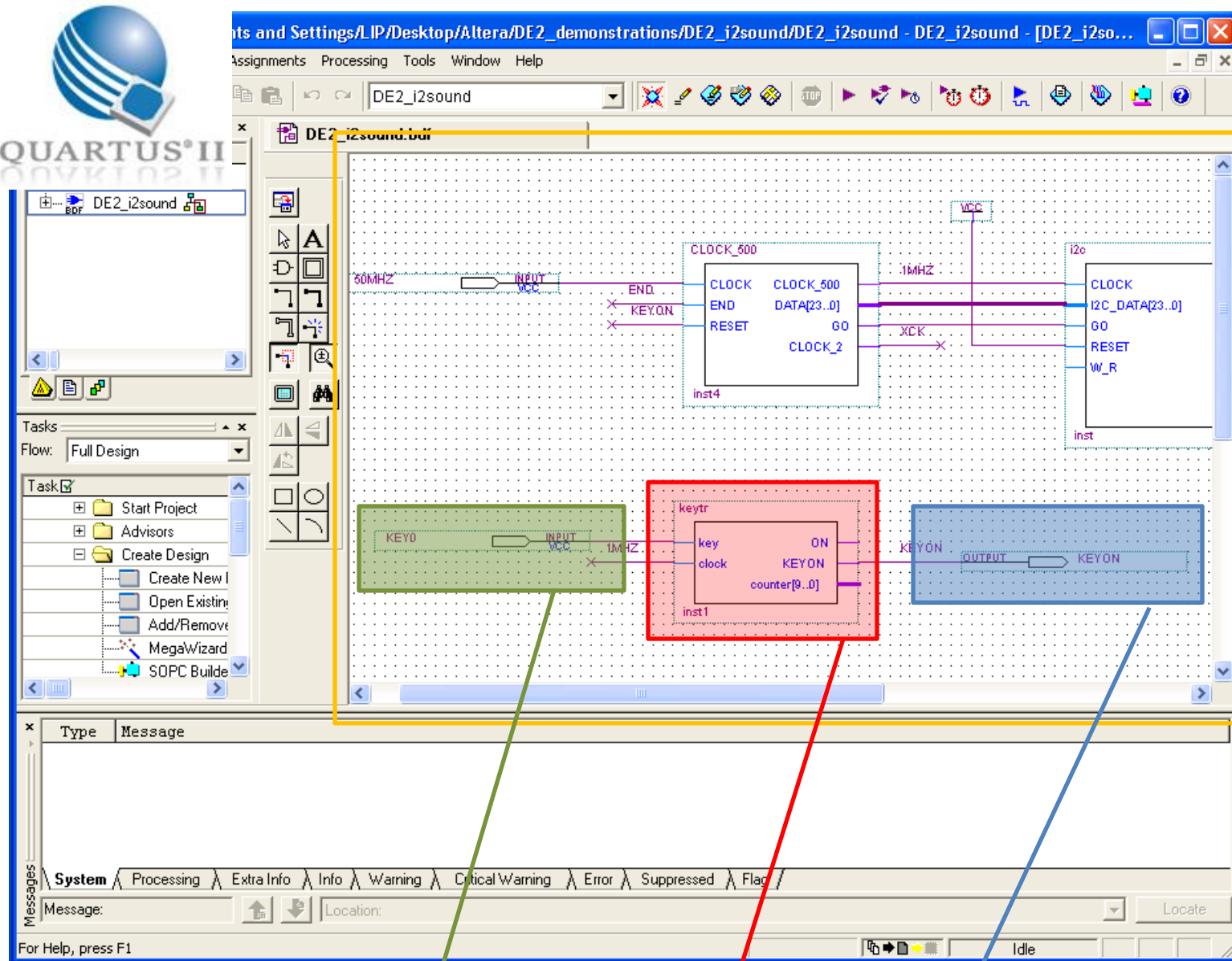


OLA!Undo



HEX3[5]	PIN_U22	Seven Segment Digit 3[5]
HEX3[6]	PIN_W24	Seven Segment Digit 3[6]
HEX4[0]	PIN_U9	Seven Segment Digit 4[0]
HEX4[1]	PIN_U1	Seven Segment Digit 4[1]
HEX4[2]	PIN_U2	Seven Segment Digit 4[2]
HEX4[3]	PIN_T4	Seven Segment Digit 4[3]
HEX4[4]	PIN_R7	Seven Segment Digit 4[4]
HEX4[5]	PIN_R6	Seven Segment Digit 4[5]
HEX4[6]	PIN_T3	Seven Segment Digit 4[6]
HEX5[0]	PIN_T2	Seven Segment Digit 5[0]
HEX5[1]	PIN_P6	Seven Segment Digit 5[1]
HEX5[2]	PIN_P7	Seven Segment Digit 5[2]
HEX5[3]	PIN_T9	Seven Segment Digit 5[3]
HEX5[4]	PIN_R5	Seven Segment Digit 5[4]
HEX5[5]	PIN_R4	Seven Segment Digit 5[5]
HEX5[6]	PIN_R3	Seven Segment Digit 5[6]
HEX6[0]	PIN_R2	Seven Segment Digit 6[0]
HEX6[1]	PIN_P4	Seven Segment Digit 6[1]
HEX6[2]	PIN_P3	Seven Segment Digit 6[2]
HEX6[3]	PIN_M2	Seven Segment Digit 6[3]
HEX6[4]	PIN_M3	Seven Segment Digit 6[4]
HEX6[5]	PIN_M5	Seven Segment Digit 6[5]
HEX6[6]	PIN_M4	Seven Segment Digit 6[6]
HEX7[0]	PIN_L3	Seven Segment Digit 7[0]
HEX7[1]	PIN_L2	Seven Segment Digit 7[1]
HEX7[2]	PIN_L9	Seven Segment Digit 7[2]
HEX7[3]	PIN_L6	Seven Segment Digit 7[3]
HEX7[4]	PIN_L7	Seven Segment Digit 7[4]
HEX7[5]	PIN_P9	Seven Segment Digit 7[5]
HEX7[6]	PIN_N9	Seven Segment Digit 7[6]
Signal Name	FPGA Pin No.	Description

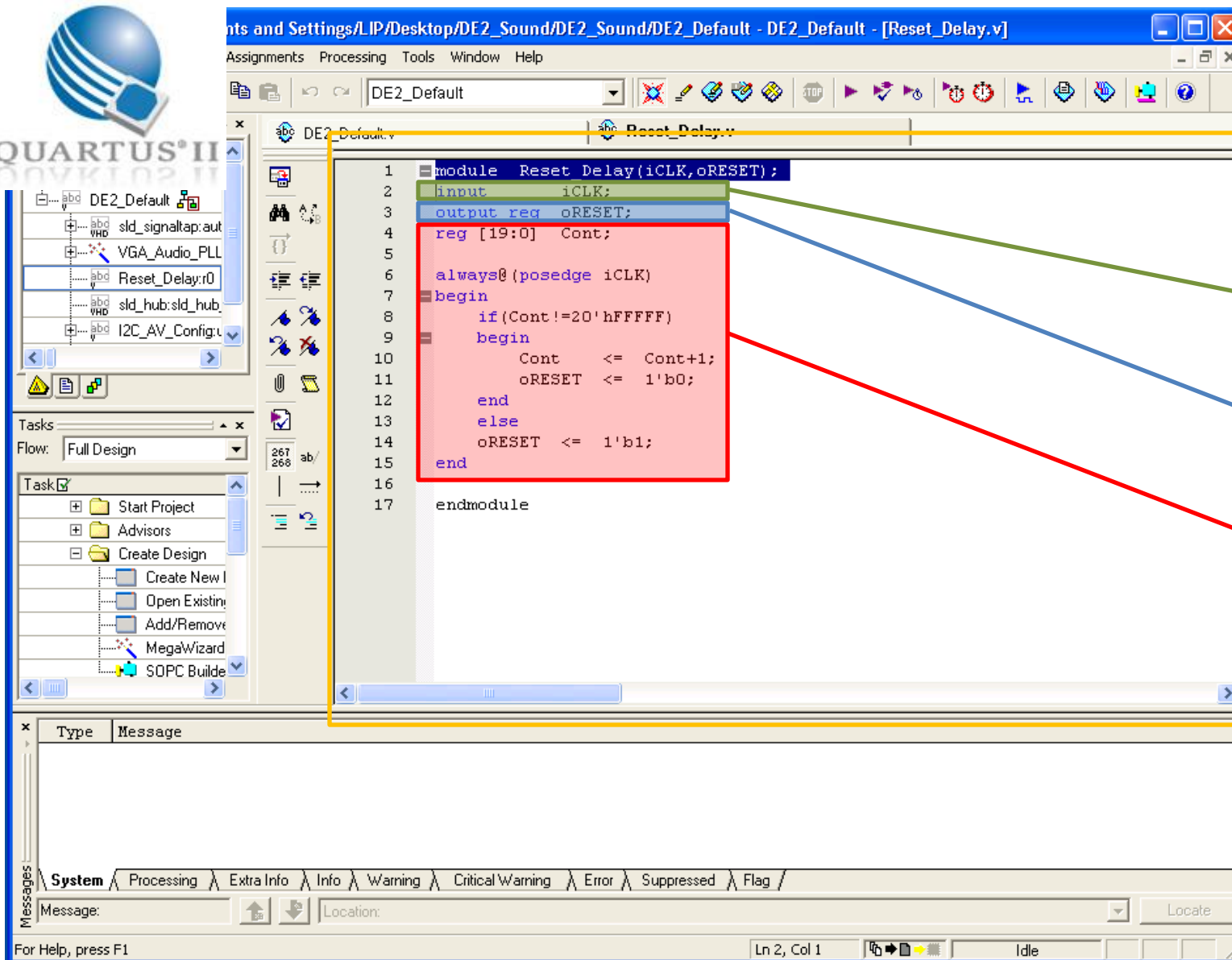




Input

Lógica

Output



Programação
esquemático

Input

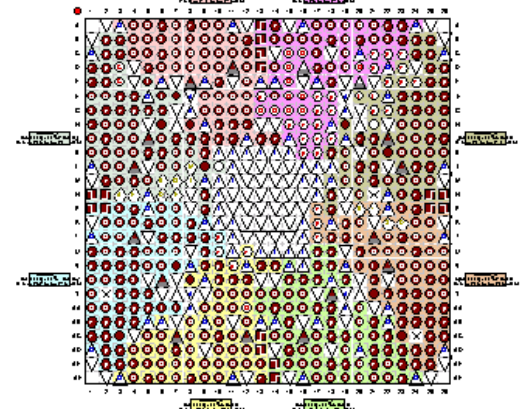
Output

Lógica

Quartus II - C:/Documents and Settings/ALP/Desktop/DE2_Sound/DE2_Sound/DE2_Default - DE2_Default - [Pin Planner]

File Edit View Processing Tools Window

Top View - Wire Bond
Cyclone II - EP2C35F672C6



Named: [] Edit: [X] [✓] Filter: Pins: all

	Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard	Reserved
1	AUD_ADCDAT	Input	PIN_B5	3	B3_N1	3.3-V LVTTTL	
2	AUD_ADCLRCK	Output	PIN_C5	3	B3_N1	3.3-V LVTTTL	
3	AUD_BCLK	Output	PIN_B4	3	B3_N1	3.3-V LVTTTL	
4	AUD_DACDAT	Output	PIN_A4	3	B3_N1	3.3-V LVTTTL	
5	AUD_DACLCK	Output	PIN_C6	3	B3_N1	3.3-V LVTTTL	
6	AUD_XCK	Output	PIN_A5	3	B3_N1	3.3-V LVTTTL	
7	CLOCK_27	Input	PIN_D13	3	B3_N0	3.3-V LVTTTL (default)	
8	CLOCK_50	Input	PIN_N2	2	B2_N1	3.3-V LVTTTL (default)	
9	DRAM_ADDR[11]	Output	PIN_V5	1	B1_N1	3.3-V LVTTTL (default)	
10	DRAM_ADDR[10]	Output	PIN_Y1	1	B1_N1	3.3-V LVTTTL (default)	
11	DRAM_ADDR[9]	Output	PIN_W3	1	B1_N1	3.3-V LVTTTL (default)	
12	DRAM_ADDR[8]	Output	PIN_W4	1	B1_N1	3.3-V LVTTTL (default)	
13	DRAM_ADDR[7]	Output	PIN_U5	1	B1_N1	3.3-V LVTTTL (default)	
14	DRAM_ADDR[6]	Output	PIN_U7	1	B1_N1	3.3-V LVTTTL (default)	
15	DRAM_ADDR[5]	Output	PIN_U6	1	B1_N1	3.3-V LVTTTL (default)	
16	DRAM_ADDR[4]	Output	PIN_W1	1	B1_N0	3.3-V LVTTTL (default)	

All Pins

For Help, press F1

File→Open Project→DE2_top

Quartus II - C:/Documents and Settings/LIP/Desktop/Altera/DE2_demonstrations/DE2_Top/DE2_TOP - DE2_TOP - [DE2_TOP.v]

File Edit View Project Assignments Processing Tools Window Help

DE2_TOP

Entity

Cyclone II: EP2C35F672C6

DE2_TOP

Tasks

Flow: Full Design

Task

- Start Project
- Advisors
- Create Design
- Create New I
- Open Existing
- Add/Remove
- MegaWizard
- SOPC Builder
- Assign Constraint

267 ab/

DE2_TOP.v

```
42 // V1.2 :| Johnny Chen :| 05/11/16 :| Fixed ISP1362 INT/
43 // -----
44
45 module DE2_TOP
46
47 // Clock Input
48 CLOCK_27, // 27 MHz
49 CLOCK_50, // 50 MHz
50 EXT_CLOCK, // External Clock
51 // Push Button
52 KEY, // Pushbutton[3:0]
53 // DPDT Switch
54 SW, // Toggle Switch[17:0]
55 // 7-SEG Display
56 HEX0, // Seven Segment Digit 0
57 HEX1, // Seven Segment Digit 1
58 HEX2, // Seven Segment Digit 2
59 HEX3, // Seven Segment Digit 3
60 HEX4, // Seven Segment Digit 4
61 HEX5, // Seven Segment Digit 5
62 HEX6, // Seven Segment Digit 6
63 HEX7, // Seven Segment Digit 7
64 // LED
65 LEDG, // LED Green[8:0]
66 LEDR, // LED Red[17:0]
67 // UART
68 UART_TXD, // UART Transmitter
69 UART_RXD, // UART Receiver
```

Type Message

System Processing Extra Info Info Warning Critical Warning Error Suppressed Flag

Message:

Location:

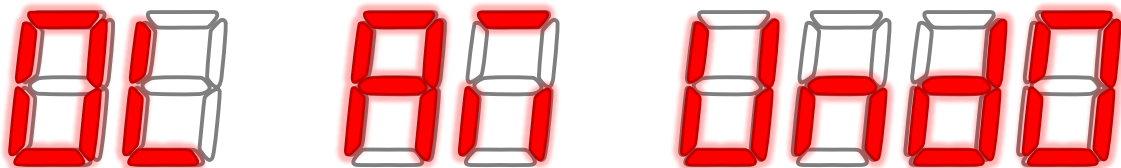
For Help, press F1

start

Quartus II - C:/Docu...

```
174 // Clock Input
175 input CLOCK_27; // 27 MHz
176 input CLOCK_50; // 50 MHz
177 input EXT_CLOCK; // External Clock
178 // Push Button
179 input [3:0] KEY; // Pushbutton[3:0]
180 // DPDT Switch
181 input [17:0] SW; // Toggle Switch[17:0]
182 // 7-SEG Display
183 output [6:0] HEX0; // Seven Segment Digit 0
184 output [6:0] HEX1; // Seven Segment Digit 1
185 output [6:0] HEX2; // Seven Segment Digit 2
186 output [6:0] HEX3; // Seven Segment Digit 3
187 output [6:0] HEX4; // Seven Segment Digit 4
188 output [6:0] HEX5; // Seven Segment Digit 5
189 output [6:0] HEX6; // Seven Segment Digit 6
190 output [6:0] HEX7; // Seven Segment Digit 7
191 // LED
192 output [8:0] LEDG; // LED Green[8:0]
193 output [17:0] LEDR; // LED Red[17:0]
194 // UART
195 output UART_TXD; // UART Transmitter
196 input UART_RXD; // UART Receiver
197 // IRDA
198 output IRDA_TXD; // IRDA Transmitter
199 input IRDA_RXD; // IRDA Receiver
200 // SDRAM Interface
201 inout [15:0] DRAM_DQ; // SDRAM Data bus 16 Bits
202 output [11:0] DRAM_ADDR; // SDRAM Address bus 12 Bits
```

```
300 // Turn on all display
301 assign HEX0 = 7'h00;
302 assign HEX1 = 7'h00;
303 assign HEX2 = 7'h00;
304 assign HEX3 = 7'h00;
305 assign HEX4 = 7'h00;
306 assign HEX5 = 7'h00;
307 assign HEX6 = 7'h00;
308 assign HEX7 = 7'h00;
309 assign LEDG = 9'h1FF;
310 assign LEDR = 18'h3FFFF;
311 assign LCD_ON = 1'b1;
312 assign LCD_BLON = 1'b1;
313
314 // All inout port turn to tri-state
315 assign DRAM_DQ = 16'hzzzz;
316 assign FL_DQ = 8'hzz;
317 assign SRAM_DQ = 16'hzzzz;
318 assign OTG_DATA = 16'hzzzz;
319 assign LCD_DATA = 8'hzz;
320 assign SD_DAT = 1'bz;
321 assign I2C_SDAT = 1'bz;
322 assign ENET_DATA = 16'hzzzz;
323 assign AUD_ADCLRCK = 1'bz;
324 assign AUD_DACLK = 1'bz;
325 assign AUD_BCLK = 1'bz;
326 assign GPIO_0 = 36'hzzzzzzzz;
327 assign GPIO_1 = 36'hzzzzzzzz;
328
```



Entrando no Verilog

Structural: modules, instances

Dataflow: continuous assignment

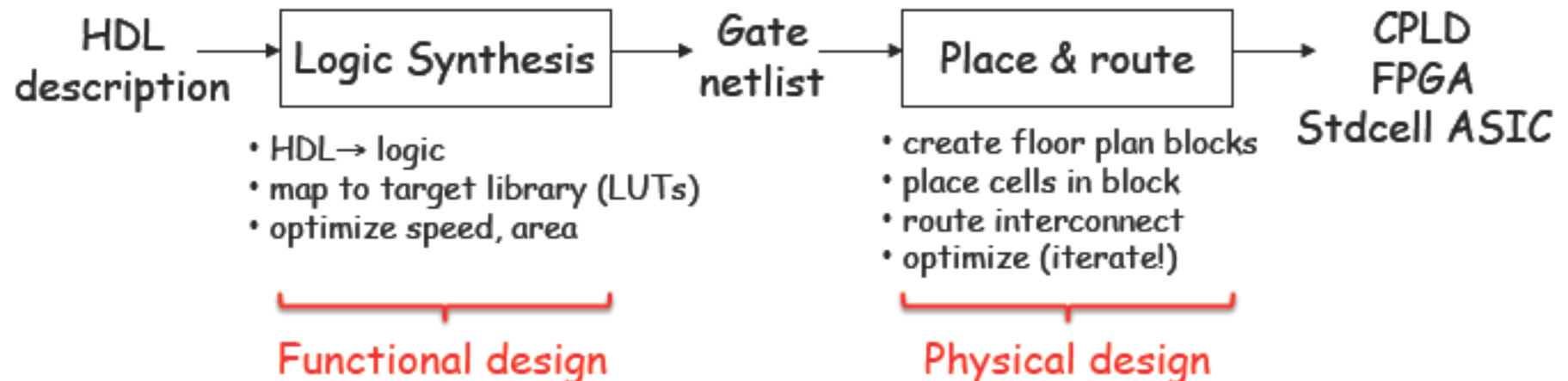
Sequential behaviour: always blocks

Using an HDL description

So, we have an executable functional specification that

- documents exact behavior of all the modules and their interfaces
- can be tested & refined until it does what we want

An HDL description is the first step in a mostly automated process to build an implementation directly from the behavioral model



Verilog data values

Since we're describing hardware, we'll need to represent the values that can appear on wires. Verilog uses a 4-valued logic:

Value	Meaning
0	Logic zero, "low"
1	Logic one, "high"
Z or ?	High impedance (tri-state buses)
X	Unknown value (simulation)

"X" is used by simulators when a wire hasn't been initialized to a known value or when the predicted value is an illegitimate logic value (e.g., due to contention on a tri-state bus).

Verilog also has the notion of "drive strength" but we can safely ignore this feature for our purposes.

Numeric Constants

Constant values can be specified with a specific width and radix:

```
123          // default: decimal radix, unspecified width
'd123        // 'd = decimal radix
'h7B         // 'h = hex radix
'o173        // 'o = octal radix
'b111_1011   // 'b = binary radix, "_" are ignored
'hxx         // can include X, Z or ? in non-decimal constants
16'd5        // 16-bit constant 'b0000_0000_0000_0101
11'h1X?      // 11-bit constant 'b001_XXXX_ZZZZ
```

By default constants are unsigned and will be extended with 0's on left if need be (if high-order bit is X or Z, the extended bits will be X or Z too). You can specify a signed constant as follows:

```
8'shFF       // 8-bit twos-complement representation of -1
```

To be absolutely clear in your intent it's usually best to explicitly specify the width and radix.

Wires

We have to provide declarations* for all our named wires (aka "nets"). We can create buses - indexed collections of wires - by specifying the allowable range of indices in the declaration:

```
wire a,b,z;           // three 1-bit wires
wire [31:0] memdata;  // a 32-bit bus
wire [7:0] b1,b2,b3,b4; // four 8-bit buses
wire [W-1:0] input;   // parameterized bus
```

Note that [0:7] and [7:0] are both legitimate but it pays to develop a convention and stick with it. Common usage is [MSB:LSB] where MSB > LSB; usually LSB is 0. Note that we can use an expression in our index declaration but the expression's value must be able to be determined at compile time. We can also build unnamed buses via concatenation:

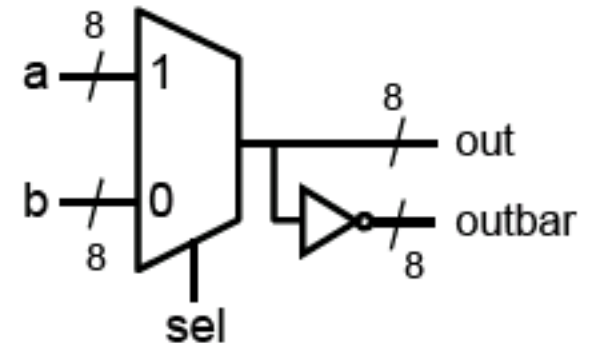
```
{b1,b2,b3,b4} // 32-bit bus, b1 is [31:24], b2 is [23:16], ...
{4{b1[3:0]},16'h0000} // 32-bit bus, 4 copies of b1[3:0], 16 0's
```

* Actually by default undeclared identifiers refer to a 1-bit wire, but this means typos get you into trouble. Specify "`default_nettype none" at the top of your source files to avoid this bogus behavior.

n-bit signals

- Multi-bit signals and buses are easy in Verilog.
- 2-to-1 multiplexer with 8-bit operands:

```
module mux_2_to_1(a, b, out,  
                  outbar, sel);  
    input [7:0] a, b;  
    input sel;  
    output [7:0] out, outbar;  
    reg [7:0] out;  
    always @ (a or b or sel)  
    begin  
        if (sel) out = a;  
        else out = b;  
    end  
    assign outbar = ~out;  
endmodule
```



- {m,n} Concatenate m to n, creating larger vector

```
// if the MSB of a is high, this module  
// concatenates 1111 to the vector. With signed  
// binary numbers, this is called sign extension.
```

```
module sign_extend(a, out);  
    input [3:0] a;  
    output [7:0] out;  
  
    assign out = a[3] ? {4'b1111,a} : {4'b0000,a};  
endmodule
```

Integer Arithmetic

- Verilog's built-in arithmetic makes a 32-bit adder easy:

```
module add32(a, b, sum);  
    input[31:0] a,b;  
    output[31:0] sum;  
    assign sum = a + b;  
endmodule
```

- A 32-bit adder with carry-in and carry-out:

```
module add32_carry(a, b, cin, sum, cout);  
    input[31:0] a,b;  
    input cin;  
    output[31:0] sum;  
    output cout;  
    assign {cout, sum} = a + b + cin;  
endmodule
```


Basic building block: modules

In Verilog we design modules, one of which will be identified as our top-level module. Modules usually have named, directional ports (specified as `input`, `output` or `inout`) which are used to communicate with the module.

```
// 2-to-1 multiplexer with dual-polarity outputs
module mux2(input a,b,sel, output z,zbar);
  wire selbar,z1,z2;    // wires internal to the module
  // order doesn't matter - all statements are
  // executed concurrently!
  not i1(selbar,sel);    // inverter, name is "i1"
  and a1(z1,a,selbar);   // port order is (out,in1,in2,...)
  and a2(z2,b,sel);
  or  o1(z,z1,z2);
  not i2(zbar,z);
endmodule
```

Don't forget this ";"



In this example the module's behavior is specified using Verilog's built-in Boolean modules: `not`, `buf`, `and`, `nand`, `or`, `nor`, `xor`, `xnor`. *Just say no!* We want to specify behavior, not implementation!

- Verilog designs consist of interconnected **modules**.
- A module can be an element or collection of lower level design blocks.
- A simple module with combinational logic might look like this:

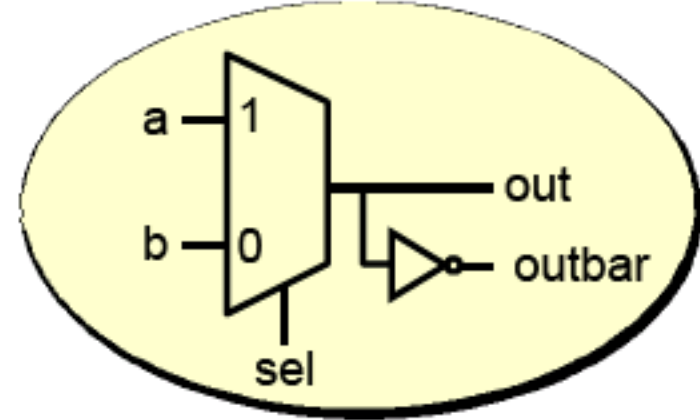
```
module mux_2_to_1(a, b, out,
                  outbar, sel);
```

```
// This is 2:1 multiplexor
```

```
input a, b, sel;
output out, outbar;
```

```
assign out = sel ? a : b;
assign outbar = ~out;
```

```
endmodule
```



$$\text{Out} = \text{sel} \bullet a + \overline{\text{sel}} \bullet b$$

2-to-1 multiplexer with inverted output

Declare and name a module; list its ports. Don't forget that semicolon.

Comment starts with //
Verilog skips from // to end of the line

Specify each port as input, output, or inout

Express the module's behavior. Each statement executes in parallel; order does not matter.

Conclude the module code.

Continuous assignments

If we want to specify a behavior equivalent to combinational logic, use Verilog's operators and continuous assignment statements:

```
// 2-to-1 multiplexer with dual-polarity outputs
module mux2(input a,b,sel, output z,zbar);
    // again order doesn't matter (concurrent execution!)
    // syntax is "assign LHS = RHS" where LHS is a wire/bus
    // and RHS is an expression
    assign z = sel ? b : a;
    assign zbar = ~z;
endmodule
```

Conceptually `assign`'s are evaluated continuously, so whenever a value used in the RHS changes, the RHS is re-evaluated and the value of the wire/bus specified on the LHS is updated.

This type of execution model is called "dataflow" since evaluations are triggered by data values flowing through the network of wires and operators.

```

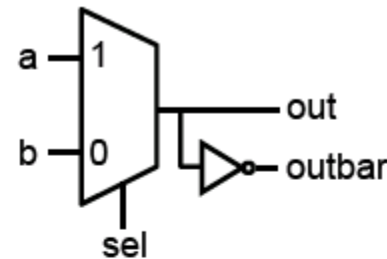
module mux_2_to_1(a, b, out,
                  outbar, sel);

    input a, b, sel;
    output out, outbar;

    assign out = sel ? a : b;
    assign outbar = ~out;

endmodule

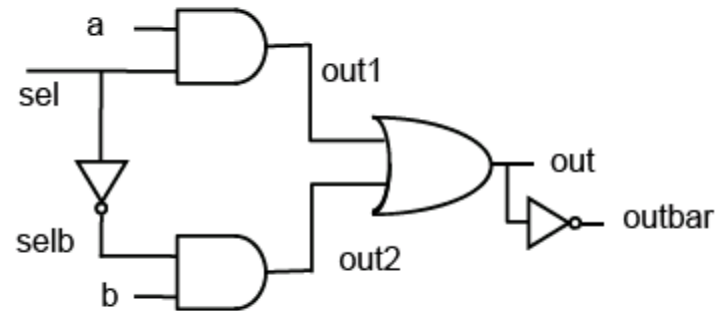
```



- Continuous assignments use the `assign` keyword
- A simple and natural way to represent combinational logic
- Conceptually, the right-hand expression is continuously evaluated as a function of arbitrarily-changing inputs...just like dataflow
- The target of a continuous assignment is a net driven by combinational logic
- Left side of the assignment must be a scalar or vector net or a concatenation of scalar and vector nets. It can't be a scalar or vector register (*discussed later*). Right side can be register or nets
- Dataflow operators are fairly low-level:
 - Conditional assignment: `(conditional_expression) ? (value-if-true) : (value-if-false);`
 - Boolean logic: `~, &, |`
 - Arithmetic: `+, -, *`
- Nested conditional operator (4:1 mux)
 - `assign out = s1 ? (s0 ? 13 : 12) : (s0 ? 11 : 10);`

Gate Level Description

```
module muxgate (a, b, out,  
outbar, sel);  
input a, b, sel;  
output out, outbar;  
wire out1, out2, selb;  
and a1 (out1, a, sel);  
not i1 (selb, sel);  
and a2 (out2, b, selb);  
or o1 (out, out1, out2);  
assign outbar = ~out;  
endmodule
```



- Verilog supports basic logic gates as primitives
 - and, nand, or, nor, xor, xnor, not, buf
 - can be extended to multiple inputs: e.g., nand nand3in (out, in1, in2, in3);
 - bufif1 and bufif0 are tri-state buffers
- Net represents connections between hardware elements. Nets are declared with the keyword `wire`.

Boolean operators

- **Bitwise operators** perform bit-oriented operations on vectors
 - $\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = 4'b1010$
 - $4'b0101 \& 4'b0011 = \{0\&0, 1\&0, 0\&1, 1\&1\} = 4'b0001$
- **Reduction operators** act on each bit of a single input vector
 - $\&(4'b0101) = 0 \& 1 \& 0 \& 1 = 1'b0$
- **Logical operators** return one-bit (true/false) results
 - $!(4'b0101) = 1'b0$

Bitwise

$\sim a$	NOT
$a \& b$	AND
$a b$	OR
$a \wedge b$	XOR
$a \sim\wedge b$ $a \wedge\sim b$	XNOR

Reduction

$\&a$	AND
$\sim\&a$	NAND
$ a$	OR
$\sim a$	NOR
$\wedge a$	XOR
$\sim\wedge a$ $\wedge\sim a$	XNOR

Logical

$!a$	NOT
$a \&\& b$	AND
$a b$	OR
$a == b$ $a != b$	[in]equality returns x when x or z in bits. Else returns 0 or 1
$a === b$ $a !== b$	case [in]equality returns 0 or 1 based on bit by bit comparison

*Note distinction between $\sim a$ and $!a$
when operating on multi-bit values*

Other operators

Conditional

$a ? b : c$	If a then b else c
-------------	--------------------

Relational

$a > b$	greater than
$a \geq b$	greater than or equal
$a < b$	Less than
$a \leq b$	Less than or equal

Arithmetic

$-a$	negate
$a + b$	add
$a - b$	subtract
$a * b$	multiply
a / b	divide
$a \% b$	modulus
$a ** b$	exponentiate
$a \ll b$	logical left shift
$a \gg b$	logical right shift
$a \lll b$	arithmetic left shift
$a \ggg b$	arithmetic right shift

Hierarchy: module instances

Our descriptions are often hierarchical, where a module's behavior is specified by a circuit of module instances:

```
// 4-to-1 multiplexer
module mux4(input d0,d1,d2,d3, input [1:0] sel, output z);
  wire z1,z2;
  // instances must have unique names within current module.
  // connections are made using .portname(expression) syntax.
  // once again order doesn't matter...
  mux2 m1(.sel(sel[0]),.a(d0),.b(d1),.z(z1)); // not using zbar
  mux2 m2(.sel(sel[0]),.a(d2),.b(d3),.z(z2));
  mux2 m3(.sel(sel[1]),.a(z1),.b(z2),.z(z));
  // could also write "mux2 m3(z1,z2,sel[1],z,)" NOT A GOOD IDEA!
endmodule
```

Connections to a module's ports are made using a syntax that specifies both the port name and the wire(s) that connects to it, so ordering of the ports doesn't have to be remembered.

This type of hierarchical behavioral model is called "structural" since we're building up a structure of instances connected by wires. We often mix dataflow and structural modeling when describing a module's behavior.

Parameterized modules

```
// 2-to-1 multiplexer, W-bit data
module mux2 #(parameter W=1) // data width, default 1 bit
    (input [W-1:0] a,b,
     input sel,
     output [W-1:0] z);
    assign z = sel ? b : a;
    assign zbar = ~z;
endmodule
```

```
// 4-to-1 multiplexer, W-bit data
module mux4 #(parameter W=1) // data width, default 1 bit
    (input [W-1:0] d0,d1,d2,d3,
     input [1:0] sel,
     output [W-1:0] z);
    wire [W-1:0] z1,z2;

    mux2 #(.W(W)) m1(.sel(sel[0]),.a(d0),.b(d1),.z(z1));
    mux2 #(.W(W)) m2(.sel(sel[0]),.a(d2),.b(d3),.z(z2));
    mux2 #(.W(W)) m3(.sel(sel[1]),.a(z1),.b(z2),.z(z));
endmodule
```



*could be an expression evaluable at compile time;
if parameter not specified, default value is used*

Sequential behaviors

There are times when we'd like to use sequential semantics and more powerful control structures - these are available inside sequential `always` blocks:

```
// 4-to-1 multiplexer
module mux4(input a,b,c,d, input [1:0] sel, output reg z,zbar);
  always @(*) begin
    if (sel == 2'b00) z = a;
    else if (sel == 2'b01) z = b;
    else if (sel == 2'b10) z = c;
    else if (sel == 2'b11) z = d;
    else z = 1'bx; // when sel is X or Z
    // statement order matters inside always blocks
    // so the following assignment happens *after* the
    // if statement has been evaluated
    zbar = ~z;
  end
endmodule
```

`always` `@(*)` blocks are evaluated whenever any value used inside changes. Equivalently we could have written

```
always @(a, b, c, d, sel) begin ... end // careful, prone to error!
```


reg vs wire

We've been using `wire` declarations when naming nets (ports are declared as wires by default). However nets appearing on the LHS of assignment statements inside of `always` blocks *must* be declared as type `reg`.

I don't why Verilog has this rule! I think it's because traditionally `always` blocks were used for sequential logic (the topic of next lecture) which led to the synthesis of hardware registers instead of simply wires. So this seemingly unnecessary rule really supports historical usage - the declaration would help the reader distinguish registered values from combinational values.

We can add the `reg` keyword to `output` or `inout` ports (we wouldn't be assigning values to `input` ports!), or we can declare nets using `reg` instead of `wire`.

```
output reg [15:0] result    // 16-bit output bus assigned in always block
reg flipflop;               // declaration of 1-bit net of type reg
```

Mix Assignments

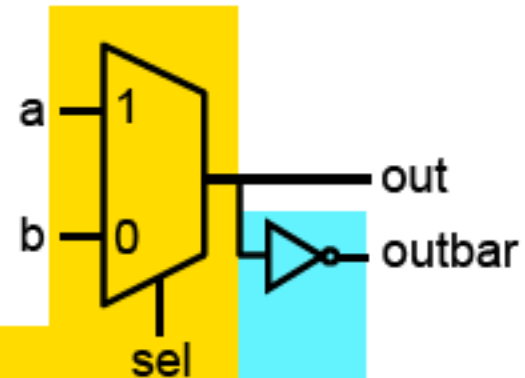
- Procedural and continuous assignments can (and often do) co-exist within a module
- Procedural assignments update the value of `reg`. The value will remain unchanged till another procedural assignment updates the variable. This is the main difference with continuous assignments in which the right hand expression is constantly placed on the left-side

```
module mux_2_to_1(a, b, out,  
                  outbar, sel);  
    input a, b, sel;  
    output out, outbar;  
    reg out;
```

```
    always @ (a or b or sel)  
    begin  
        if (sel) out = a;  
        else out = b;  
    end
```

```
    assign outbar = ~out;
```

```
endmodule
```



*procedural
description*

*continuous
description*

Case statements

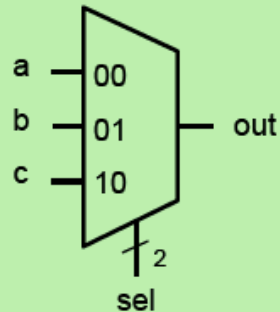
Chains of if-then-else statements aren't the best way to indicate the intent to provide an alternative action for every possible control value. Instead use `case`:

```
// 4-to-1 multiplexer
module mux4(input a,b,c,d, input [1:0] sel, output reg z,zbar);
  always @(*) begin
    case (sel)
      2'b00: z = a;
      2'b01: z = b;
      2'b10: z = c;
      2'b11: z = d;
      default: z = 1'bx; // in case sel is X or Z
    endcase
    zbar = ~z;
  end
endmodule
```

`case` looks for an exact bit-by-bit match of the value of the case expression (e.g., `sel`) against each case item, working through the items in the specified order. `casex/casez` statements treat X/Z values in the selectors as don't cares when doing the matching that determines which clause will be executed.

Danger

Goal:



Proposed Verilog Code:

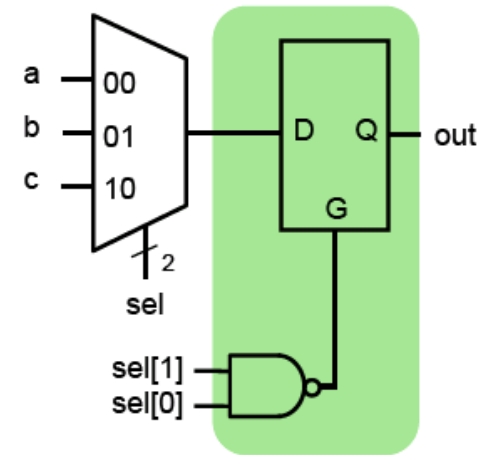
```
module maybe_mux_3to1(a, b, c, sel, out)
    input [1:0] sel;
    input a,b,c;
    output out;
    reg out;

    always @(a or b or c or sel)
    begin
        case (sel)
            2'b00: out = a;
            2'b01: out = b;
            2'b10: out = c;
            // 2'b11: out = ?;
        endcase
    end
endmodule
```

If out is not assigned during any pass through the always block, then the **previous value must be retained**

LATCH

Synthesized Result:



- Latch memory “latches” old data when G=0 (we will discuss latches later)
- In practice, we almost *never* intend this

Solution:

- Precede all conditionals with a default assignment for all signals assigned within them...

```
always @(a or b or c or sel)
begin
    out = 1'bx;
    case (sel)
        2'b00: out = a;
        2'b01: out = b;
        2'b10: out = c;
    endcase
end
endmodule
```

```
always @(a or b or c or sel)
begin
    case (sel)
        2'b00: out = a;
        2'b01: out = b;
        2'b10: out = c;
        default: out = 1'bx;
    endcase
end
endmodule
```

- ...or, fully specify all branches of conditionals and assign all signals from all branches

- For each if, include else
- For each case, include default

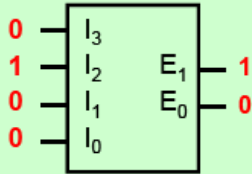
Danger

Goal:

Proposed Verilog Code:

Code: if $i[0]$ is 1, the result is 00 regardless of the other inputs.
 $i[0]$ takes the highest priority.

4-to-2 Binary Encoder



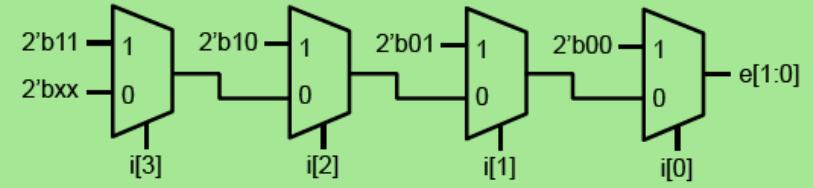
I_3	I_2	I_1	I_0	E_1	E_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1
all others				X	X

```
module binary_encoder(i, e);
    input [3:0] i;
    output [1:0] e;
    reg [1:0] e;

    always @(i)
    begin
        if (i[0]) e = 2'b00;
        else if (i[1]) e = 2'b01;
        else if (i[2]) e = 2'b10;
        else if (i[3]) e = 2'b11;
        else e = 2'bxx;
    end
endmodule
```

```
if (i[0]) e = 2'b00;
else if (i[1]) e = 2'b01;
else if (i[2]) e = 2'b10;
else if (i[3]) e = 2'b11;
else e = 2'bxx;
end
```

Inferred Result:



What is the resulting circuit?

If-else and case statements are interpreted very literally!
Beware of unintended priority logic

Solution:

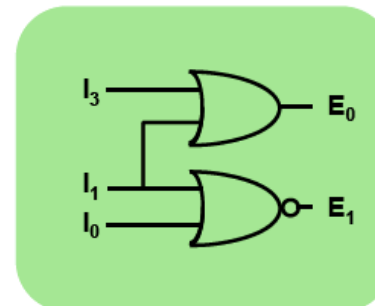
- Make sure that if-else and case statements are *parallel*
 - If **mutually exclusive conditions** are chosen for each branch...
 - ...then synthesis tool can generate a simpler circuit that evaluates the branches in parallel

Parallel Code:

```
module binary_encoder(i, e);
    input [3:0] i;
    output [1:0] e;
    reg [1:0] e;

    always @(i)
    begin
        if (i == 4'b0001) e = 2'b00;
        else if (i == 4'b0010) e = 2'b01;
        else if (i == 4'b0100) e = 2'b10;
        else if (i == 4'b1000) e = 2'b11;
        else e = 2'bxx;
    end
endmodule
```

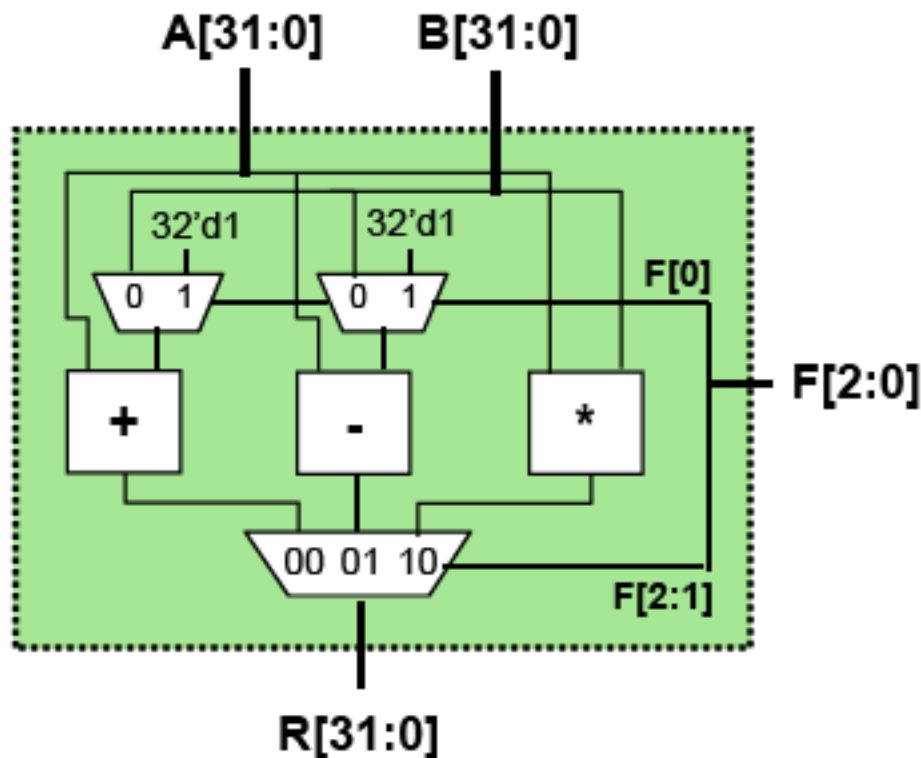
Minimized Result:



Connecting modules to build complex machines

- Modularity is essential to the success of large designs
- A Verilog module may contain submodules that are “wired together”
- High-level primitives enable direct synthesis of behavioral descriptions (functions such as additions, subtractions, shifts (<< and >>), etc.

Example: A 32-bit ALU



Function Table

F2	F1	F0	Function
0	0	0	$A + B$
0	0	1	$A + 1$
0	1	0	$A - B$
0	1	1	$A - 1$
1	0	X	$A * B$

Modules

2-to-1 MUX

```
module mux32two(i0,i1,sel,out);
input [31:0] i0,i1;
input sel;
output [31:0] out;

assign out = sel ? i1 : i0;

endmodule
```

3-to-1 MUX

```
module mux32three(i0,i1,i2,sel,out);
input [31:0] i0,i1,i2;
input [1:0] sel;
output [31:0] out;
reg [31:0] out;

always @ (i0 or i1 or i2 or sel)
begin
    case (sel)
        2'b00: out = i0;
        2'b01: out = i1;
        2'b10: out = i2;
        default: out = 32'bx;
    endcase
end
endmodule
```

32-bit Adder

```
module add32(i0,i1,sum);
input [31:0] i0,i1;
output [31:0] sum;

assign sum = i0 + i1;

endmodule
```

32-bit Subtractor

```
module sub32(i0,i1,diff);
input [31:0] i0,i1;
output [31:0] diff;

assign diff = i0 - i1;

endmodule
```

16-bit Multiplier

```
module mul16(i0,i1,prod);
input [15:0] i0,i1;
output [31:0] prod;

// this is a magnitude multiplier
// signed arithmetic later
assign prod = i0 * i1;

endmodule
```

Top-Level: connect the modules

■ Given submodules:

```
module mux32two(i0,i1,sel,out);  
module mux32three(i0,i1,i2,sel,out);  
module add32(i0,i1,sum);  
module sub32(i0,i1,diff);  
module mul16(i0,i1,prod);
```

■ Declaration of the ALU Module:

```
module alu(a, b, f, r);  
  input [31:0] a, b;  
  input [2:0] f;  
  output [31:0] r;
```

```
  wire [31:0] addmux_out, submux_out;  
  wire [31:0] add_out, sub_out, mul_out;
```

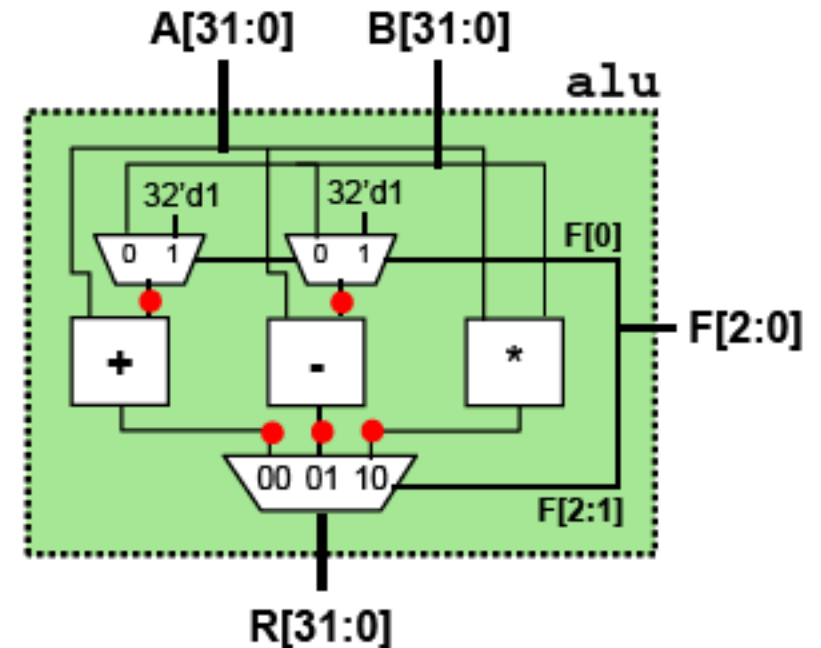
```
  mux32two    adder_mux(b, 32'd1, f[0], addmux_out);  
  mux32two    sub_mux(b, 32'd1, f[0], submux_out);  
  add32       our_adder(a, addmux_out, add_out);  
  sub32       our_subtractor(a, submux_out, sub_out);  
  mul16       our_multiplier(a[15:0], b[15:0], mul_out);  
  mux32three  output_mux(add_out, sub_out, mul_out, f[2:1], r);
```

endmodule

module
names

(unique)
instance
names

corresponding
wires/regs in
module alu



intermediate output nodes ●

More on modules

- **Explicit port naming allows port mappings in arbitrary order: better scaling for large, evolving designs**

Given Submodule Declaration:

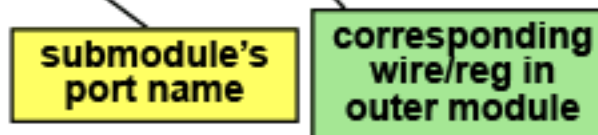
```
module mux32three(i0,i1,i2,sel,out);
```

Module Instantiation with Ordered Ports:

```
mux32three output_mux(add_out, sub_out, mul_out, f[2:1], r);
```

Module Instantiation with Named Ports:

```
mux32three output_mux(.sel(f[2:1]), .out(r), .i0(add_out),  
                        .i1(sub_out), .i2(mul_out));
```



- **Built-in Verilog gate primitives may be instantiated as well**
 - **Instantiations may omit instance name and must be ordered:**

```
and(out, in1,in2,...inN);
```

Other useful verilog features (just for reference)

- Additional control structures: `for`, `while`, `repeat`, `forever`
- Procedure-like constructs: functions, tasks
- One-time-only initialization: `initial` blocks
- Compile-time computations: `generate`, `genvar`
- System tasks to help write simulation test jigs
 - Stop the simulation: `$finish(...)`
 - Print out text, values: `$display(...)`
 - Initialize memory from a file: `$readmemh(...)`, `$readmemb(...)`
 - Capture simulation values: `$dumpfile(...)`, `$dumpvars(...)`
 - Explicit time delays: `#nnn`
- Compiler directives
 - Macro definitions: ``define`
 - Conditional compilation: ``ifdef`, ...
 - Include other source files: ``include`
 - Control simulation time units: ``timescale`
 - No implicit net declarations: ``default_nettype none`

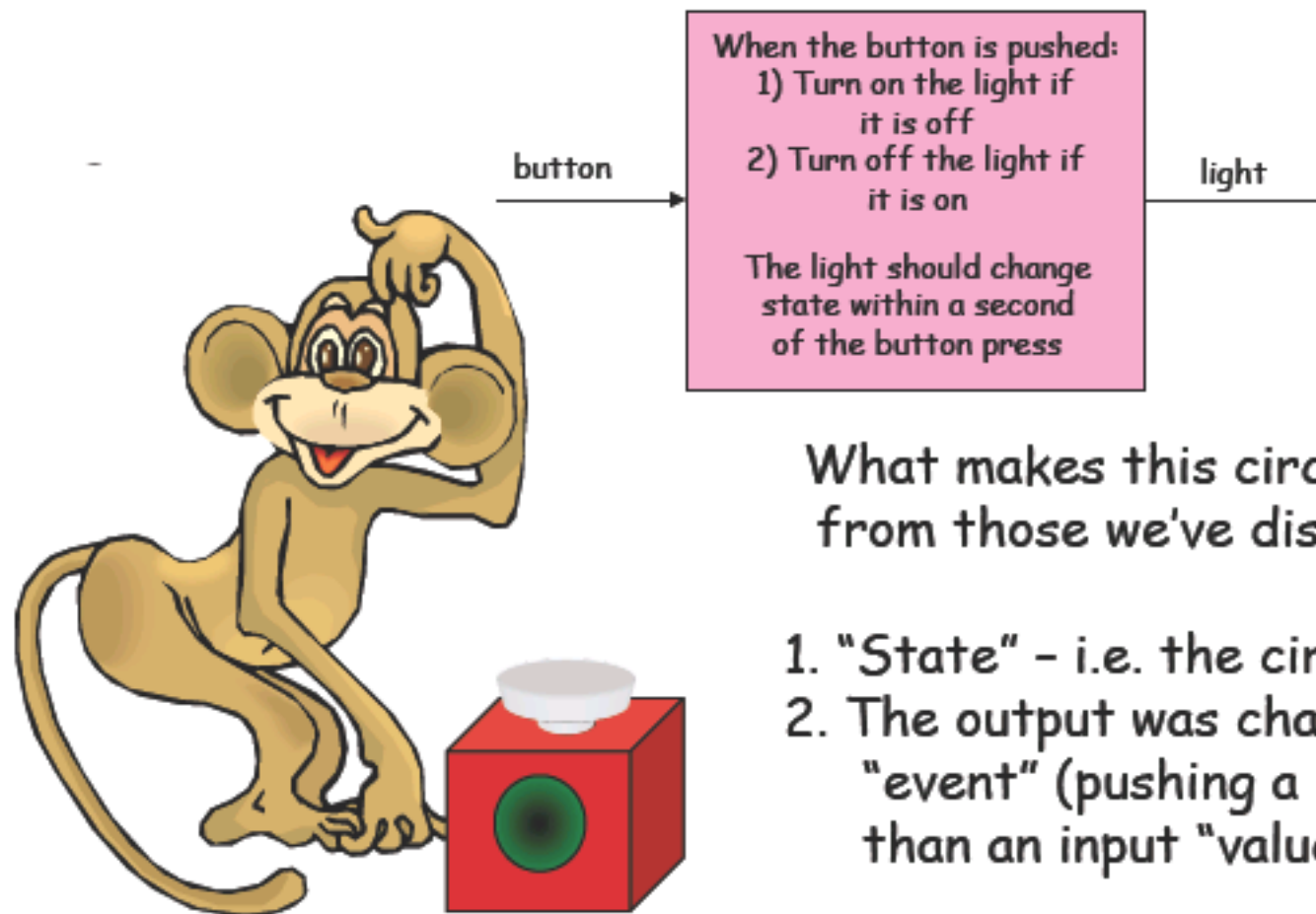


Next: Sequential Logic

Sequential Logic

Something We Can't Build (Yet)

What if you were given the following design specification:

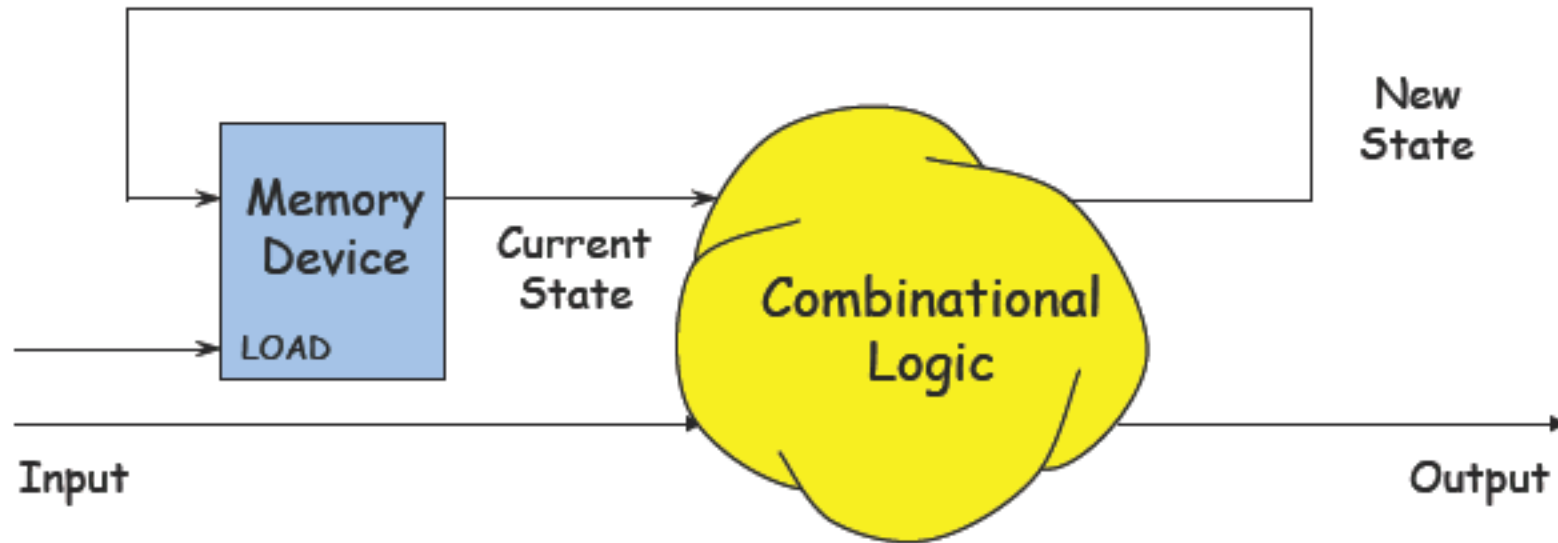


What makes this circuit so different from those we've discussed before?

1. "State" - i.e. the circuit has memory
2. The output was changed by a input "event" (pushing a button) rather than an input "value"

Digital State

One model of what we'd like to build



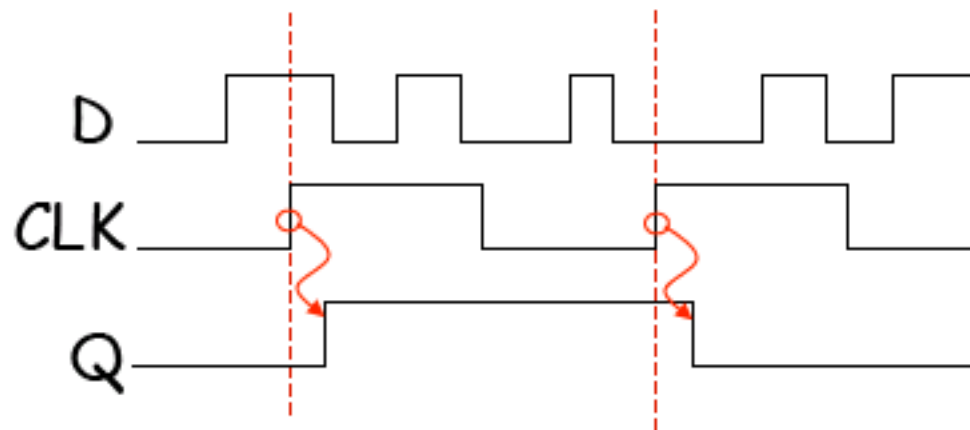
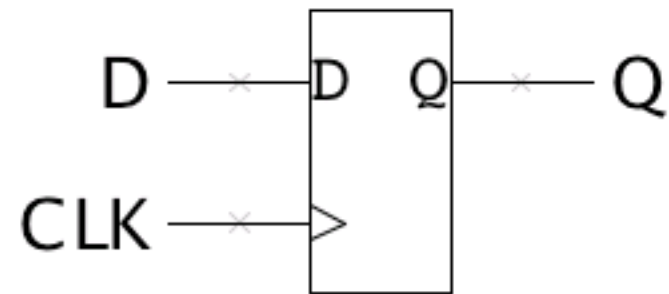
Plan: Build a Sequential Circuit with stored digital STATE -

- Memory stores CURRENT state, produced at output
- Combinational Logic computes
 - NEXT state (from input, current state)
 - OUTPUT bit (from input, current state)
- State changes on LOAD control input

When Output depends on input and current state, circuit is called a Mealy machine. If Output depends only on the current state, circuit is called a Moore machine.

Our next building block: the D register

The edge-triggered D register: *on the rising edge of CLK*, the value of D is saved in the register and then shortly afterwards appears on Q.

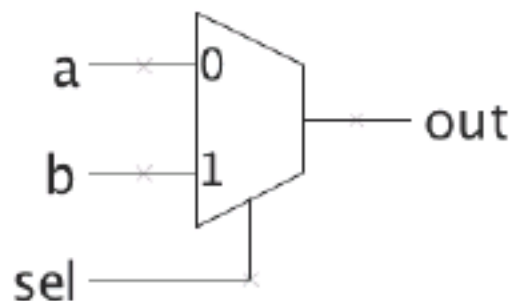


The Sequential **always** Block

Edge-triggered circuits are described using a sequential **always** block

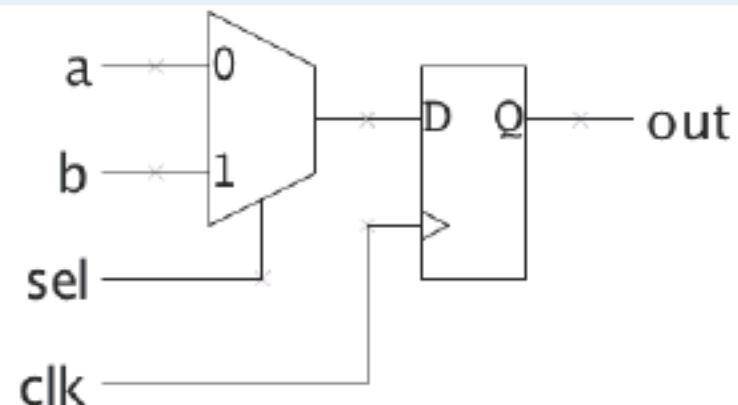
Combinational

```
module comb(input a, b, sel,
            output reg out);
  always @(*) begin
    if (sel) out = a;
    else out = b;
  end
endmodule
```



Sequential

```
module seq(input a, b, sel, clk,
            output reg out);
  always @(posedge clk) begin
    if (sel) out <= a;
    else out <= b;
  end
endmodule
```



Importance of the Sensitivity List

- The use of `posedge` and `negedge` makes an `always` block sequential (edge-triggered)
- Unlike a combinational `always` block, the sensitivity list **does** determine behavior for synthesis!

*D-Register with **synchronous** clear*

```
module dff_sync_clear(  
    input d, clearb, clock,  
    output reg q  
);  
    always @(posedge clock)  
    begin  
        if (!clearb) q <= 1'b0;  
        else q <= d;  
    end  
endmodule
```

`always` block entered only at each positive clock edge

*D-Register with **asynchronous** clear*

```
module dff_sync_clear(  
    input d, clearb, clock,  
    output reg q  
);  
    always @(negedge clearb or posedge clock)  
    begin  
        if (!clearb) q <= 1'b0;  
        else q <= d;  
    end  
endmodule
```

`always` block entered immediately when (active-low) `clearb` is asserted

Note: The following is incorrect syntax: `always @(clear or negedge clock)`
If one signal in the sensitivity list uses `posedge/negedge`, then all signals must.

- Assign any signal or variable from only one `always` block. Be wary of race conditions: `always` blocks with same trigger execute concurrently...

Blocking vs. Nonblocking Assignments

- Verilog supports two types of assignments within `always` blocks, with subtly different behaviors.
- *Blocking assignment* (`=`): evaluation and assignment are immediate

```
always @(*) begin
  x = a | b;      // 1. evaluate a|b, assign result to x
  y = a ^ b ^ c;  // 2. evaluate a^b^c, assign result to y
  z = b & ~c;     // 3. evaluate b&(~c), assign result to z
end
```

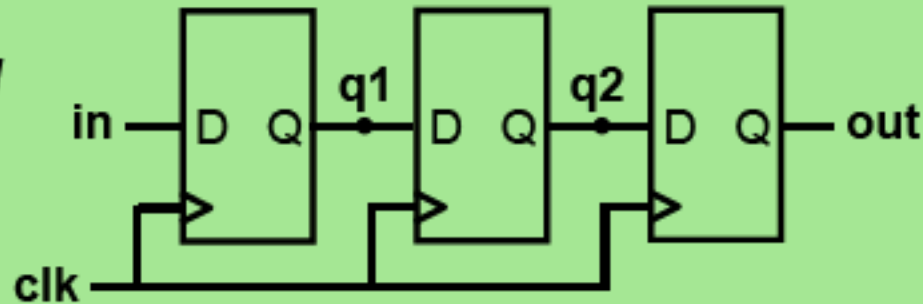
Nonblocking assignment (`<=`): all assignments deferred to end of simulation time step after all right-hand sides have been evaluated (*even those in other active `always` blocks*)

```
always @(*) begin
  x <= a | b;      // 1. evaluate a|b, but defer assignment to x
  y <= a ^ b ^ c;  // 2. evaluate a^b^c, but defer assignment to y
  z <= b & ~c;     // 3. evaluate b&(~c), but defer assignment to z
  // 4. end of time step: assign new values to x, y and z
end
```

Sometimes, as above, both produce the same result. Sometimes, not!

Assignment styles for sequential logic

Flip-Flop Based Digital Delay Line



- Will nonblocking and blocking assignments both produce the desired result?

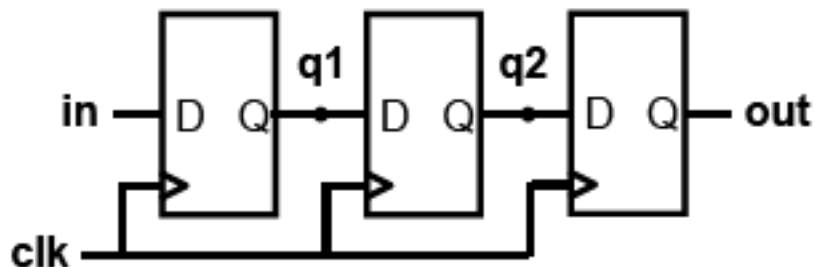
```
module nonblocking(in, clk, out);  
  input in, clk;  
  output out;  
  reg q1, q2, out;  
  always @ (posedge clk)  
  begin  
    q1 <= in;  
    q2 <= q1;  
    out <= q2;  
  end  
endmodule
```

```
module blocking(in, clk, out);  
  input in, clk;  
  output out;  
  reg q1, q2, out;  
  always @ (posedge clk)  
  begin  
    q1 = in;  
    q2 = q1;  
    out = q2;  
  end  
endmodule
```

Use nonblocking for sequential logic

```
always @ (posedge clk)
begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
end
```

“At each rising clock edge, $q1$, $q2$, and out **simultaneously receive the old values** of in , $q1$, and $q2$.”

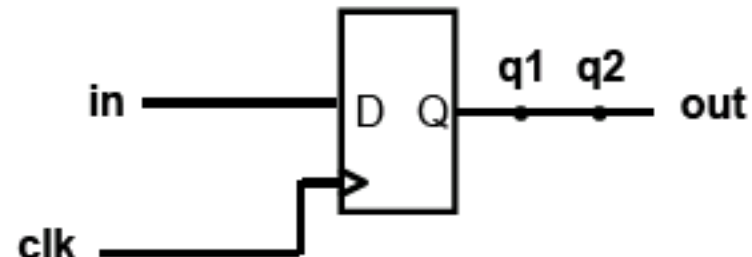


```
always @ (posedge clk)
begin
    q1 = in;
    q2 = q1;
    out = q2;
end
```

“At each rising clock edge, $q1 = in$.

After that, $q2 = q1 = in$.

After that, $out = q2 = q1 = in$.
Therefore $out = in$.”

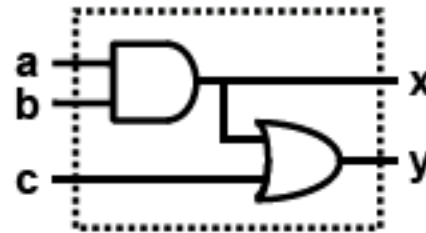


- Blocking assignments do not reflect the intrinsic behavior of multi-stage sequential logic
- **Guideline: use nonblocking assignments for sequential always blocks**

Use blocking for combinational logic

Blocking Behavior

	a	b	c	x	y
(Given) Initial Condition	1	1	0	1	1
a changes; always block triggered	0	1	0	1	1
x = a & b;	0	1	0	0	1
y = x c;	0	1	0	0	0



```

module blocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

  always @ (a or b or c)
  begin
    x = a & b;
    y = x | c;
  end
endmodule
  
```

Nonblocking Behavior

	a	b	c	x	y	Deferred
(Given) Initial Condition	1	1	0	1	1	
a changes; always block triggered	0	1	0	1	1	
x <= a & b;	0	1	0	1	1	x<=0
y <= x c;	0	1	0	1	1	x<=0, y<=1
Assignment completion	0	1	0	0	1	

```

module nonblocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

  always @ (a or b or c)
  begin
    x <= a & b;
    y <= x | c;
  end
endmodule
  
```

- Nonblocking and blocking assignments will synthesize correctly. Will both styles simulate correctly?
- Nonblocking assignments do not reflect the intrinsic behavior of multi-stage combinational logic
- While nonblocking assignments can be hacked to simulate correctly (expand the sensitivity list), it's not elegant
- **Guideline: use blocking assignments for combinational always blocks**

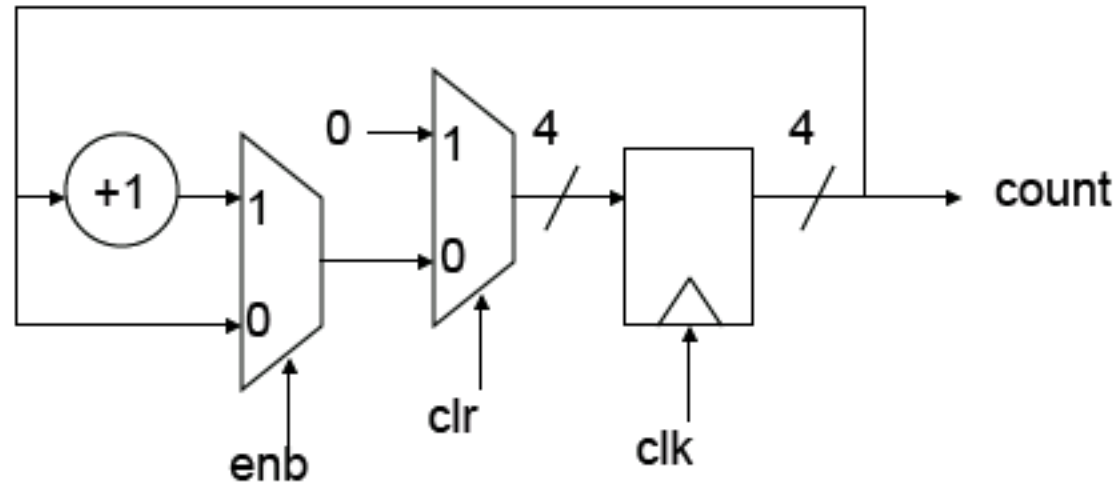
= vs <= inside always

```
always @(posedge clk) begin
    a = b;    // blocking assignment
    b = a;    // execute sequentially
end
```

```
always @(posedge clk) begin
    a <= b;   // non-blocking assignment
    b <= a;   // eval all RHSs first
end
```

Rule: always change state using <= (e.g., inside always @(posedge clk)...)

Example: A simple counter



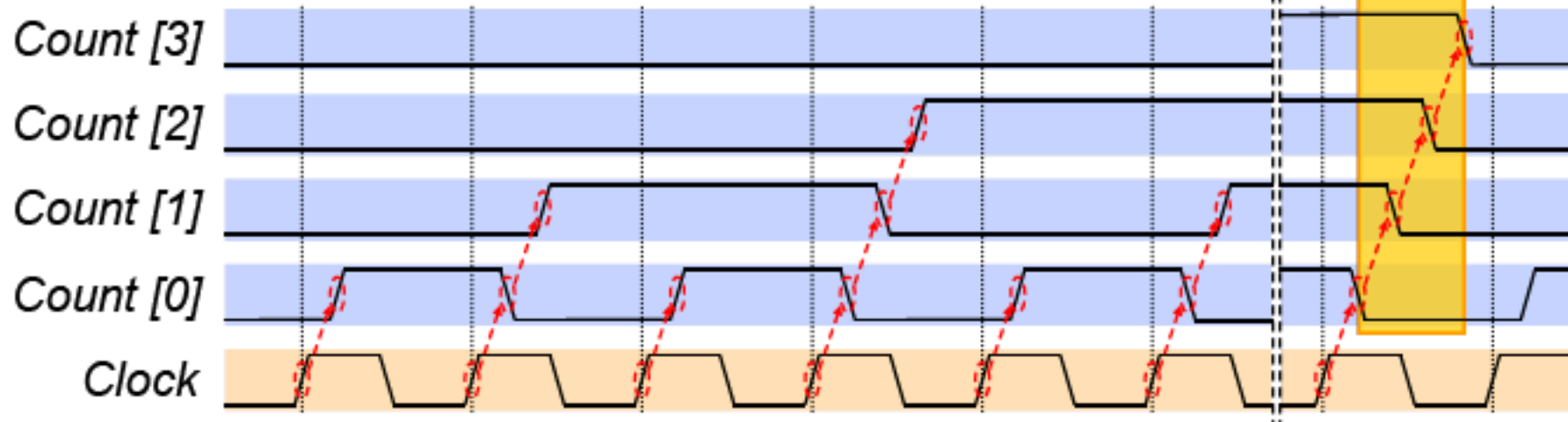
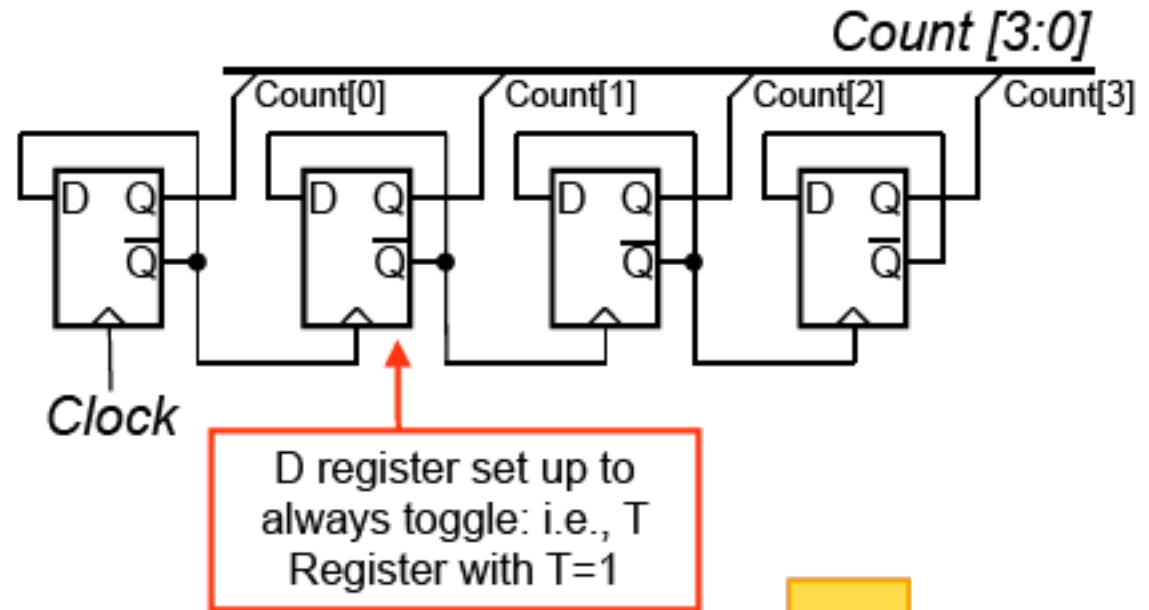
```
// 4-bit counter with enable and synchronous clear
module counter(input clk,enb,clr,
               output reg [3:0] count);
    always @(posedge clk) begin
        count <= clr ? 4'b0 : (enb ? count+1 : count);
    end
endmodule
```

The Asynchronous counter

A simple counter architecture

- uses only registers
(e.g., 74HC393 uses T-register and negative edge-clocking)
- Toggle rate fastest for the LSB

...but ripple architecture leads to large skew between outputs

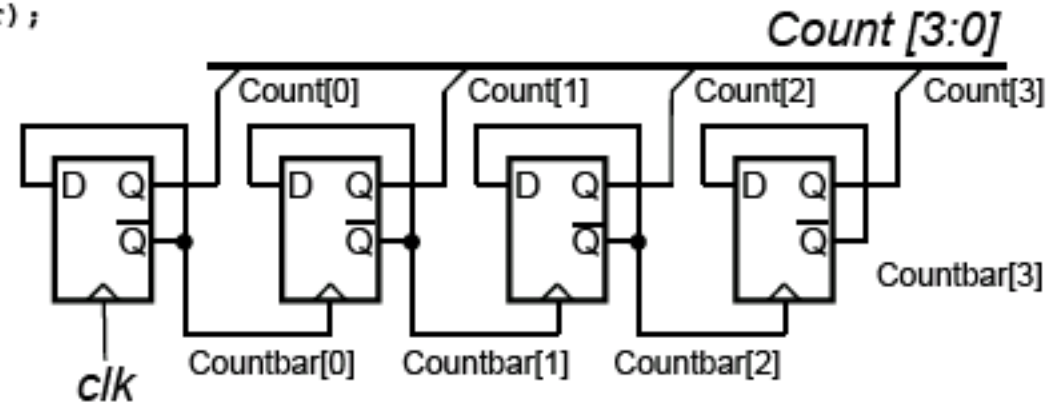


The ripple counter in verilog

Single D Register with Asynchronous Clear:

```
module dreg_async_reset (clk, clear, d, q, qbar);
input d, clk, clear;
output q, qbar;
reg q;

always @ (posedge clk or negedge clear)
begin
if (!clear)
q <= 1'b0;
else q <= d;
end
assign qbar = ~q;
endmodule
```



Structural Description of Four-bit Ripple Counter:

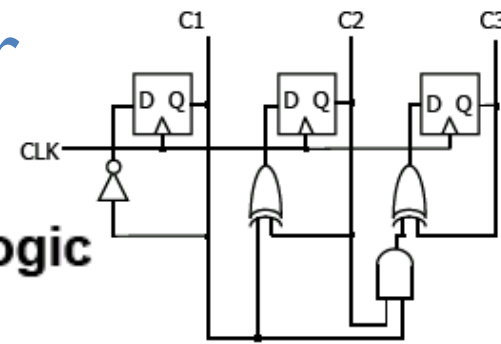
```
module ripple_counter (clk, count, clear);
input clk, clear;
output [3:0] count;
wire [3:0] count, countbar;

dreg_async_reset bit0(.clk(clk), .clear(clear), .d(countbar[0]),
                    .q(count[0]), .qbar(countbar[0]));
dreg_async_reset bit1(.clk(countbar[0]), .clear(clear), .d(countbar[1]),
                    .q(count[1]), .qbar(countbar[1]));
dreg_async_reset bit2(.clk(countbar[1]), .clear(clear), .d(countbar[2]),
                    .q(count[2]), .qbar(countbar[2]));
dreg_async_reset bit3(.clk(countbar[2]), .clear(clear), .d(countbar[3]),
                    .q(count[3]), .qbar(countbar[3]));

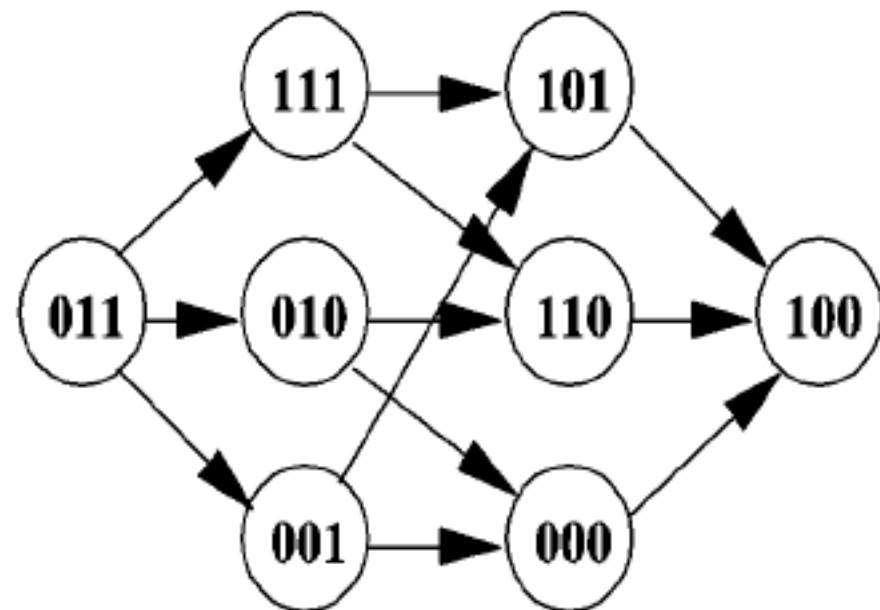
endmodule
```

Logic for a synchronous counter

- Count (C) will be retained by a D Register
- Next value of counter (N) computed by combinational logic
- Any time multiple bits change, the counter output needs time to settle.
- Even though all flip-flops share the same clock, individual bits will change at different times.
 - Clock skew, propagation time variations
- Can cause glitches in combinational logic driven by the counter
- The RCO can also have a glitch.



**Care is required of the
Ripple Carry Output:
It can have glitches:
Any of these transition
paths are possible!**



TOOLS

Tools - Simulation

Verilog

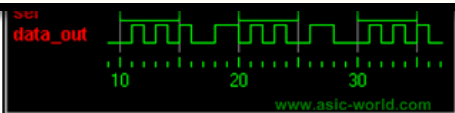
Example: Mux



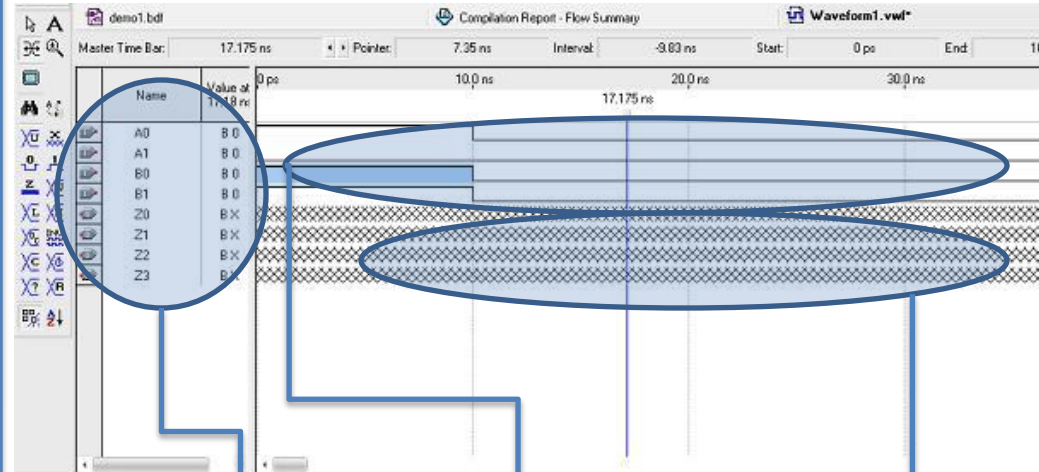
System Tasks & Functions

Quartus II support for system tasks and functions is described below. See *Description Language Based on the Verilog Hardware Description Language*

Section	Verilog HDL Construct	Quartus II Support <u>(1)</u>	<u>Note</u>
14.1	Display System Tasks	Not supported.	
14.2	File Input-Output System Tasks	Not supported.	
14.3	Timescale System Tasks	Not supported.	
14.4	Simulation Control System Tasks	Not supported.	
14.5	Timing Check System Tasks	Not supported.	
14.6	PLA Modeling System Tasks	Not supported.	
14.7	Stochastic Analysis System Tasks	Not supported.	
14.8	Simulation Time System Functions	Not supported.	
14.9	Conversion Functions for Reals	Not supported.	
14.10	Probabilistic Distribution Functions	Not supported.	



Quartus II



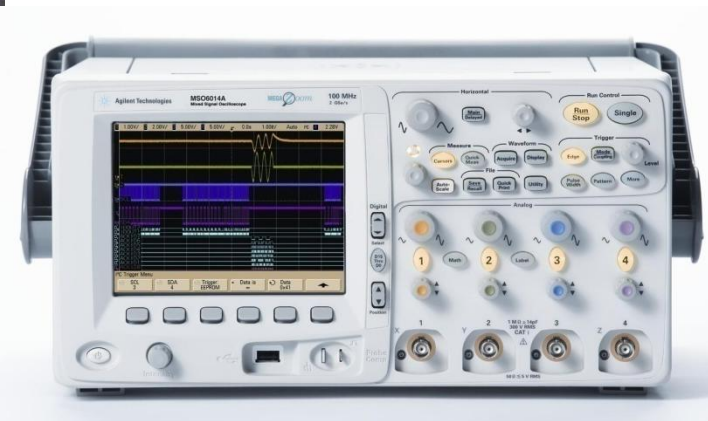
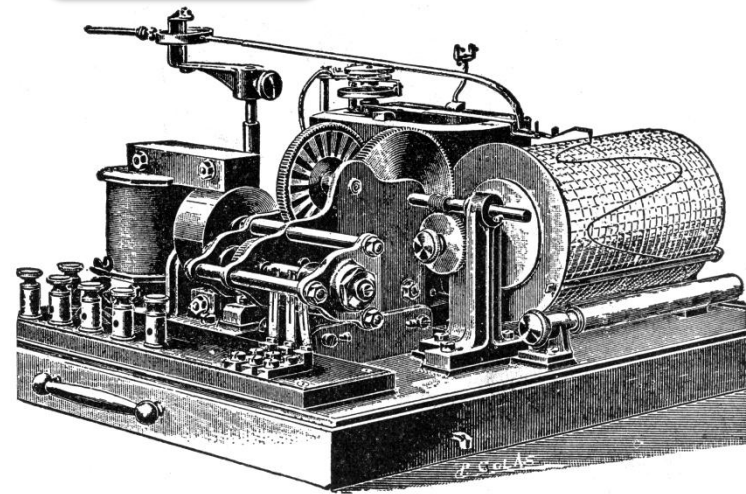
Define the signals :
Inputs – stimulus
Outputs - results

Define the stimulus to the system

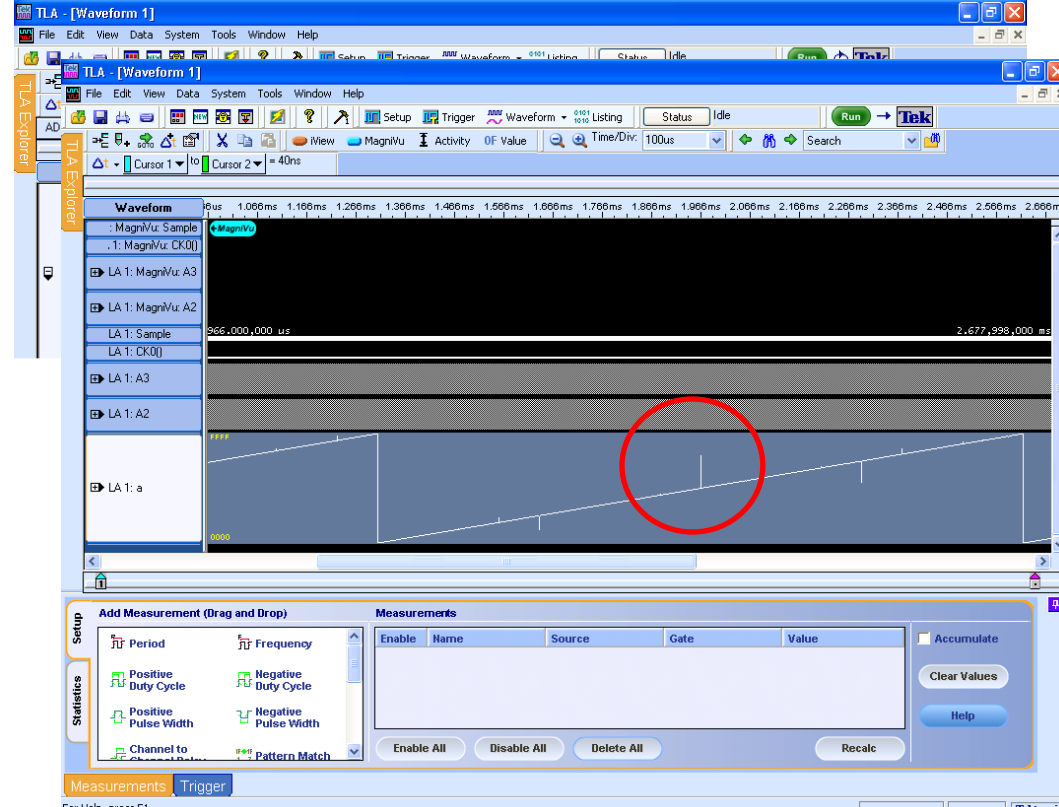
Run the simulation and you get the result

Tools – Measurement instruments

Waveforms



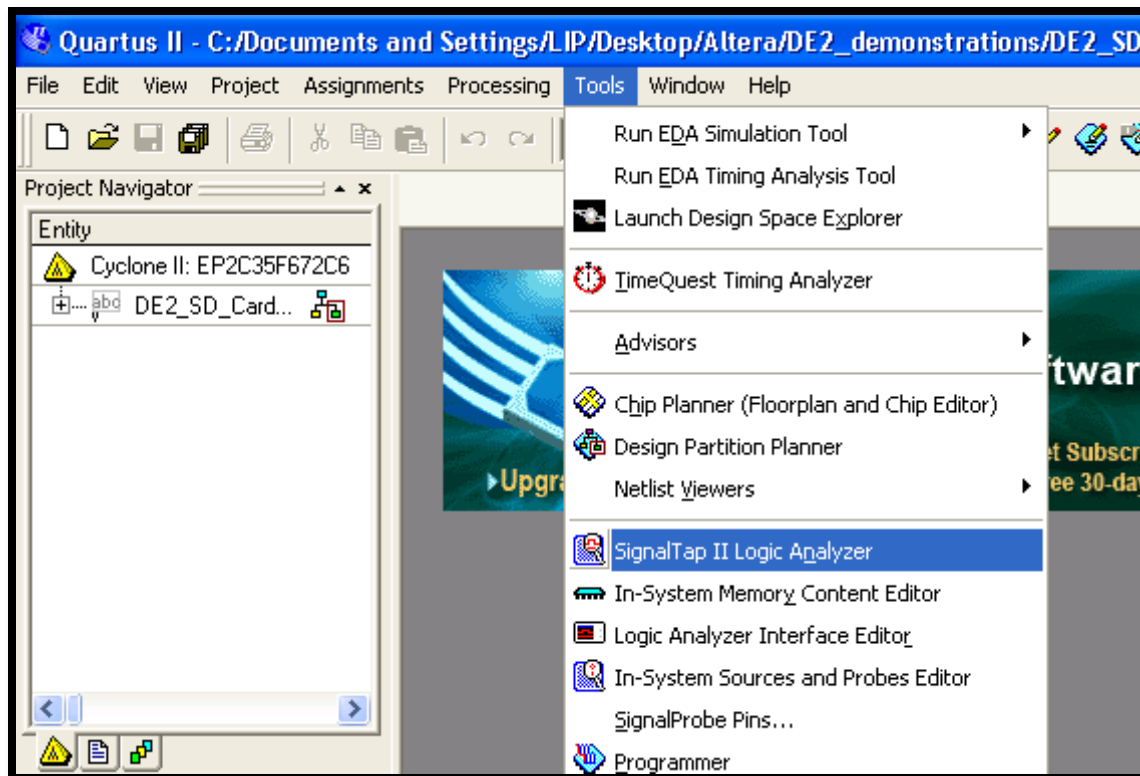
Logic Levels



Tools – Internal Logic Analyser

Signal-TAP embedded Logic Analyser

Quartus II Handbook Version 9.0 Volume 3: Verification
14. Design Debugging Using the
SignalTap II Embedded Logic Analyzer



Tools – Internal Logic Analyser

The screenshot shows the Quartus II Internal Logic Analyser interface. The main window is titled "Quartus II - C:/Documents and Settings/LIP/Desktop/Altera/DE2_SD_Card_Audio/DE2_SD_Card_Audio - DE2_SD_Card_Audio - [stp1.stp]". The interface includes a menu bar (File, Edit, View, Project, Processing, Tools, Window), a toolbar, and several panels. The "Instance Manager" panel at the top left shows a table with columns: Instance, Status, LEs, Memory, M512_MLAB, M4K_M9K, and M-RAM. The "JTAG Chain Configuration" panel at the top right shows "No device is selected". The "Signal Configuration" panel on the right is divided into "Clock", "Data", and "Trigger" sections. The "Data" section includes "Sample depth" (128), "RAM type" (Auto), "Segmented" (2 64 sample segments), "Storage qualifier" (Type: Continuous), and "Input port". The "Trigger" section includes "Trigger flow control" (Sequential), "Trigger position" (Pre trigger position), and "Trigger conditions" (1). The "Hierarchy Display" panel at the bottom left shows a tree view with "auto_singaltap_0". The "Data Log" panel at the bottom right shows a list of signals, including "auto_singaltap_0". Red arrows point from the text labels on the right to the corresponding panels in the interface.

Invalid JTAG configuration

Instance Manager:

Instance	Status	LEs	Memory	M512_MLAB	M4K_M9K	M-RAM
auto_singaltap_0	Not running	0 cells	0 bits	NA	NA	NA

JTAG Chain Configuration: No device is selected

Hardware: Disabled Setup...

Device: None Detected Scan Chain

SDF Manager:

auto_singaltap_0

Type	Alias	Name	Data Enable	Trigger Enable	Trigger Conditions
			0	0	1 Basic

Double-click to add nodes

Signal Configuration:

Clock:

Data

Sample depth: 128 RAM type: Auto

Segmented: 2 64 sample segments

Storage qualifier

Type: Continuous

Input port:

Record data discontinuities

Disable storage qualifier

Trigger

Trigger flow control: Sequential

Trigger position: Pre trigger position

Trigger conditions: 1

Hierarchy Display:

Data Log:

auto_singaltap_0

auto_singaltap_0

For Help, press F1

CONTROL

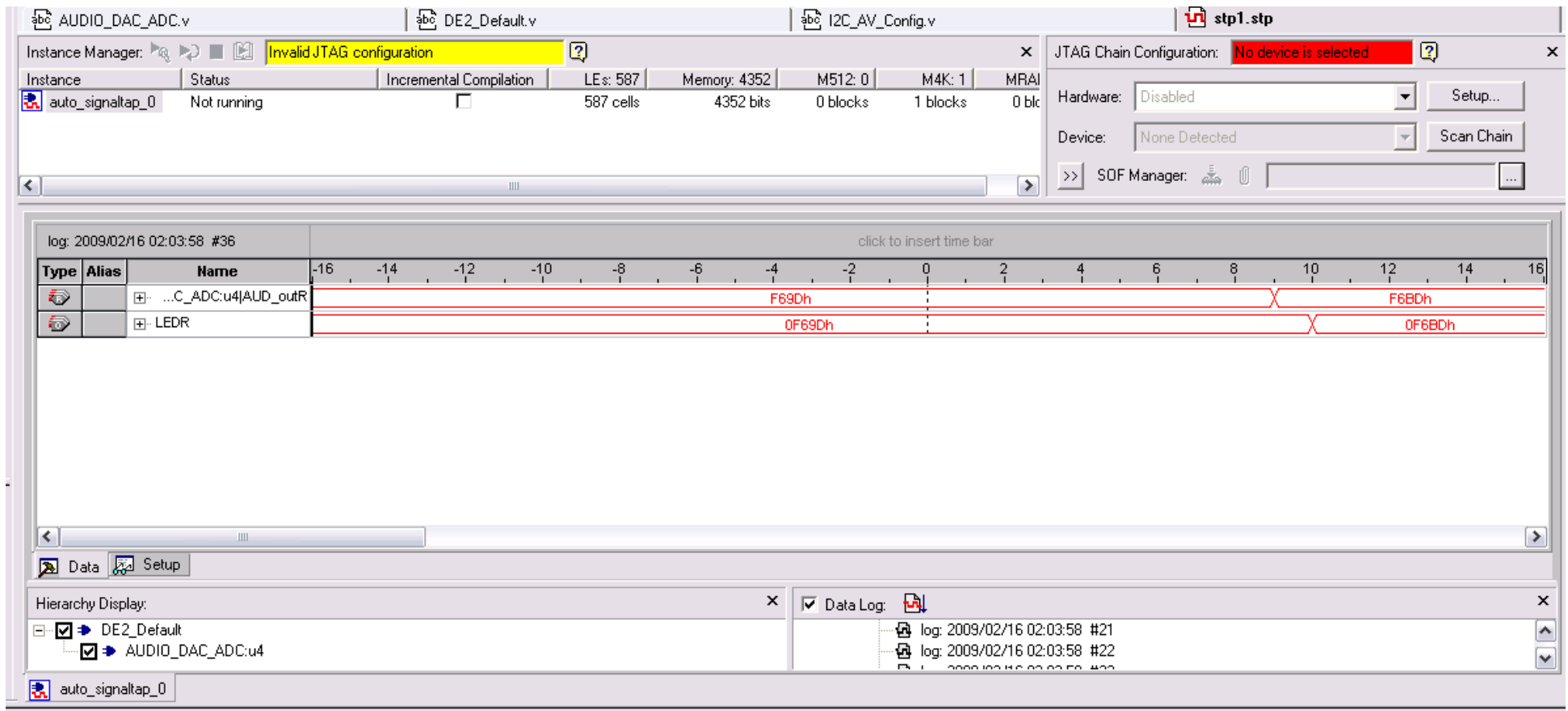
Clock
definition

Trigger
definition

The signal
you want to
"see"

Tools – Internal Logic Analyser

The logic analyser will collect data from the registers and output it through the JTAG programming interface



Tools – Signal probe

Internal signals can be extracted to output pins and connected to an external logic analyser. Signals can be exchanged easily...

SignalTap step by step

- Open DE2 default and compile it!
- Program DE2
- Tools→Signal TapII Logic Analyser

Quartus II - C:/Documents and Settings/pedjor/Desktop/DE2_Default/DE2_Default - DE2_Default - [stp1.stp]

File Edit View Project Processing Tools Window

Invalid JTAG configuration

Instance Manager: Invalid JTAG configuration

Instance	Status	LEs: 0	Memory: 0	M512,MLAB: 0/0	M4K,M9K: 75/105	M-RAM,M144K: 0/0
auto_sigtap_0	Not running	0 cells	0 bits	NA	NA	NA

JTAG Chain Configuration: No device is selected

Hardware: Please Select Setup...

Device: None Detected Scan Chain

SOF Manager: [icon] [icon] [icon]

Signal Configuration:

Clock: [icon]

Data

Sample depth: 128 RAM type: Auto

Segmented: 2 64 sample segments

Storage qualifier

Type: Continuous

Input port: [icon]

Record data discontinuities

Disable storage qualifier

Trigger

Trigger flow control: Sequential

auto_sigtap_0 Allow all changes

Node		Data Enable	Trigger Enable	Trigger Conditions
Type	Alias			
		0	0	1 Basic

Double-click to add nodes

Hierarchy Display:

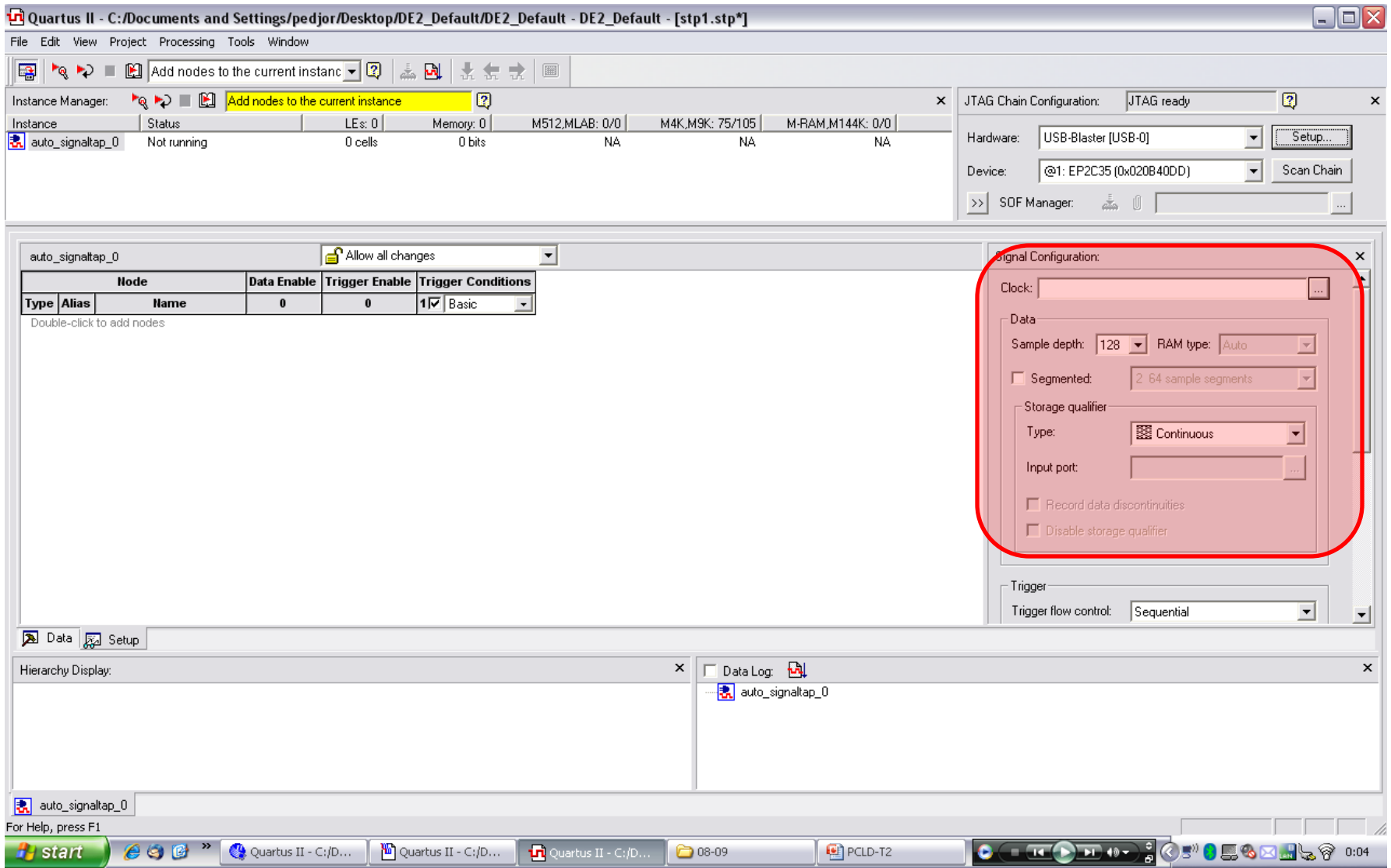
Data Log: auto_sigtap_0

auto_sigtap_0

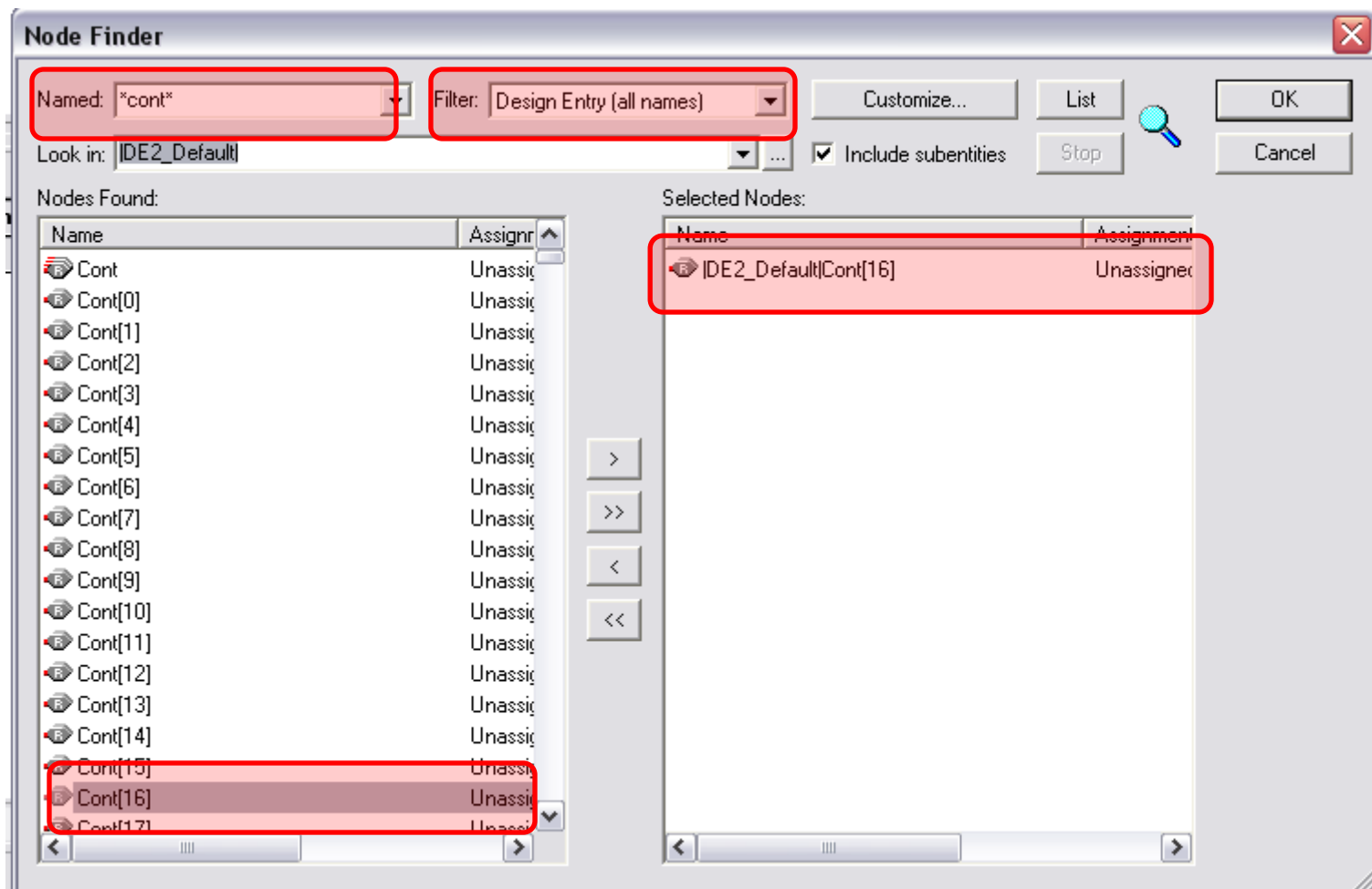
For Help, press F1

start Quartus II - C:/Docu... Quartus II - C:/Docu... 08-09 PCLD-T2 0:00

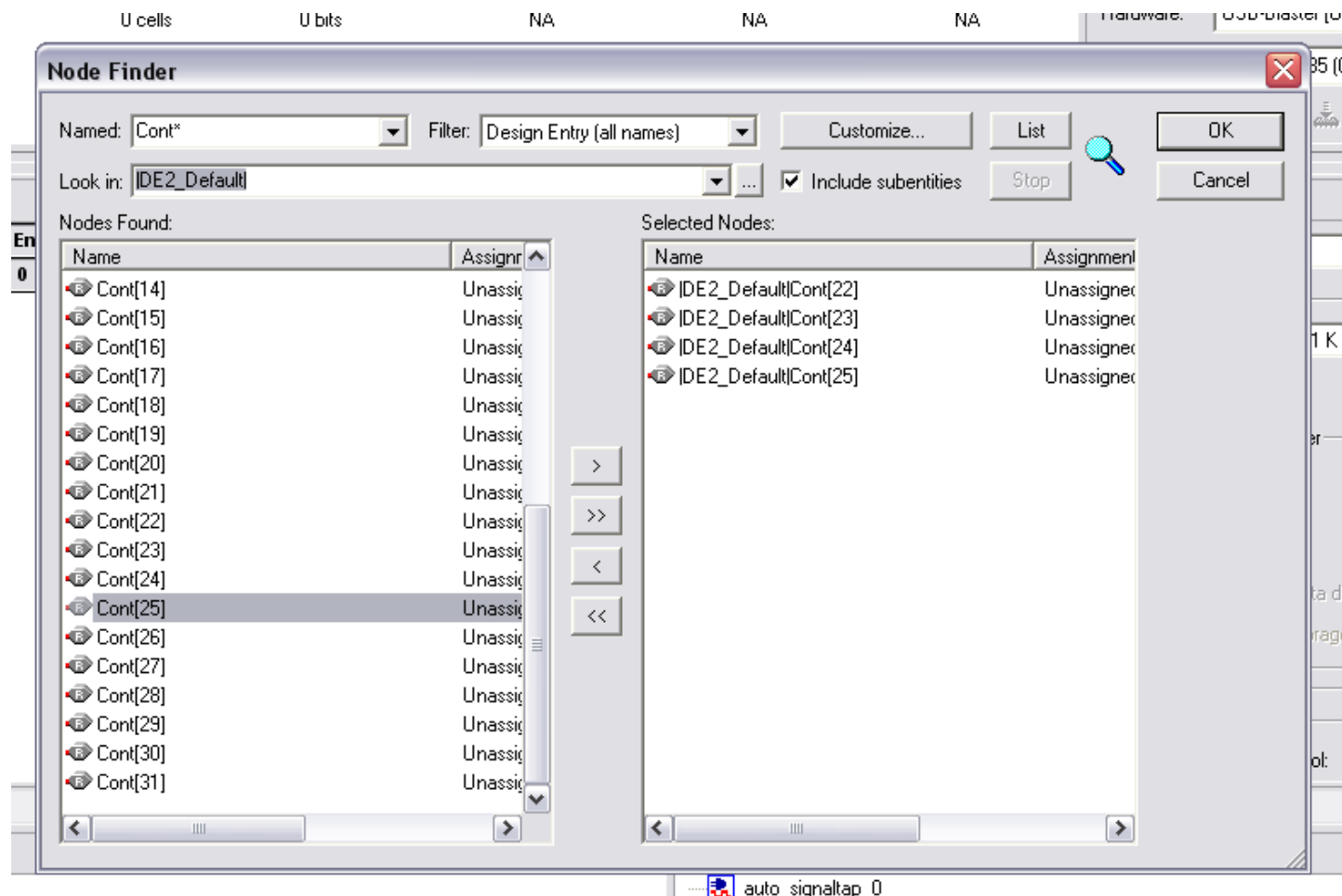
- Setup the hardware (choose the USB blaster)



- Define the clock

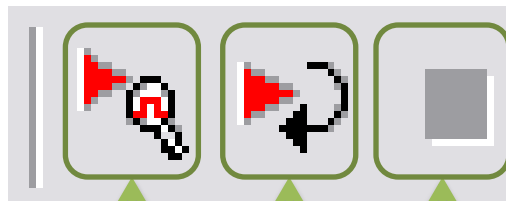
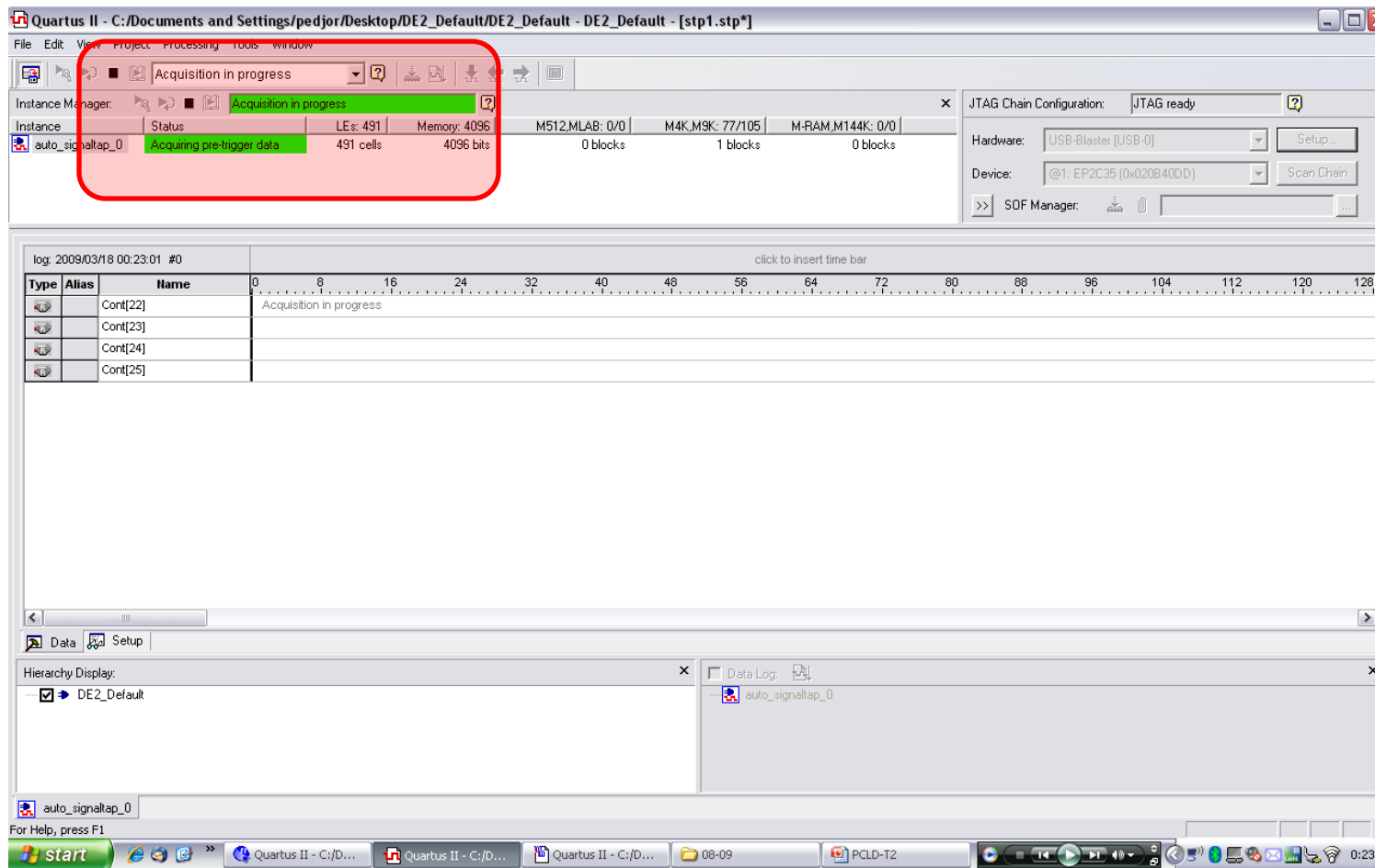


Choose the signals to observe. Choose Cont 25,24,23,22



- Compile the project! You may need to save some files and answer some questions
- Program the board

Run Analysis → Green (acquisition in progress, acquiring data)

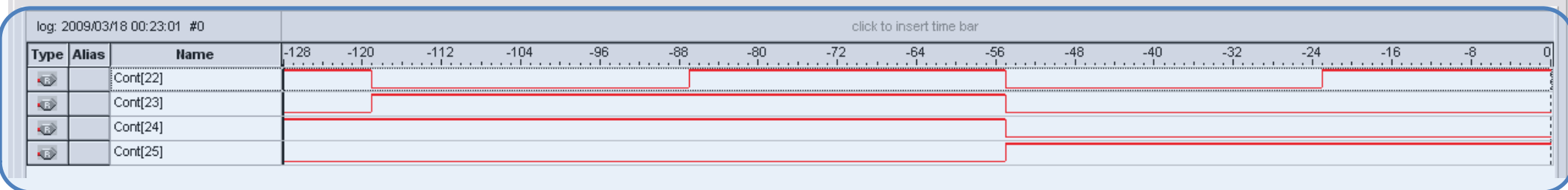
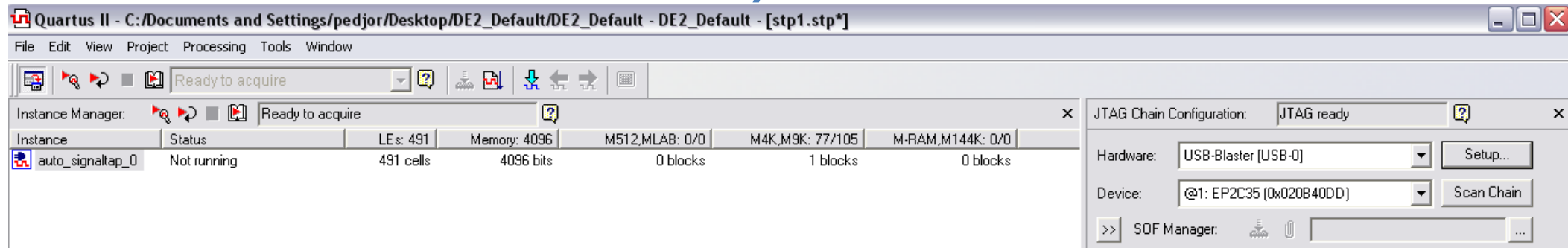


Run Analysis (1 time)

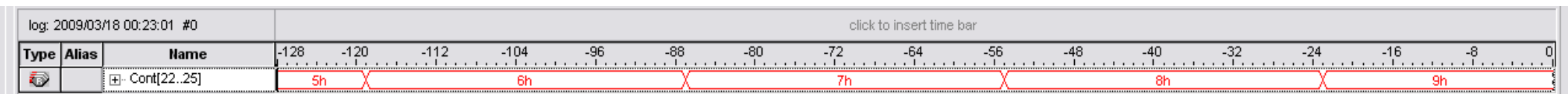
Run Analysis (continuous)

Stop

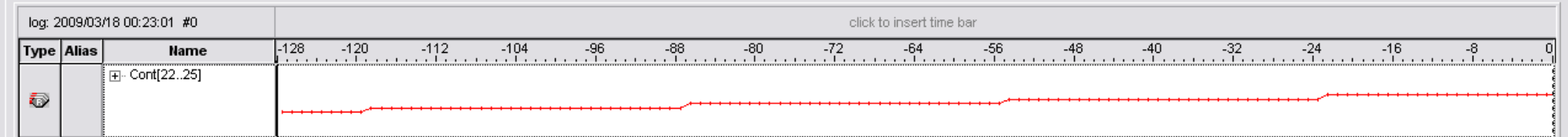
Finally: data!!



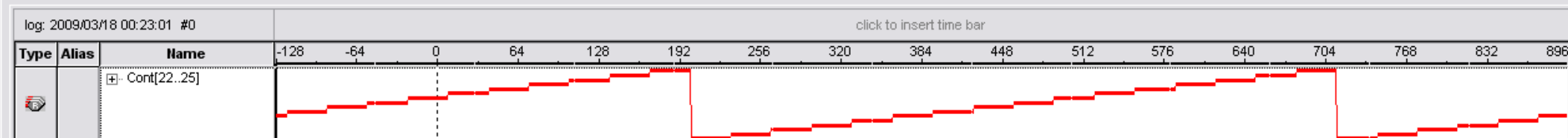
- Select the four signals;
- Edit→group



- Select the group
- Edit→Bus Display Format→Unsigned Line Chart



- Unzoom



FUTURE

Sobre os trabalhos de laboratório

Trabalhos/exercícios de introdução ao verilog

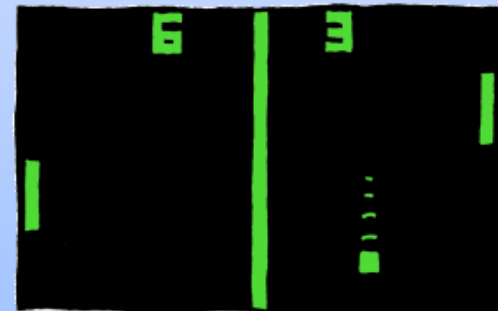
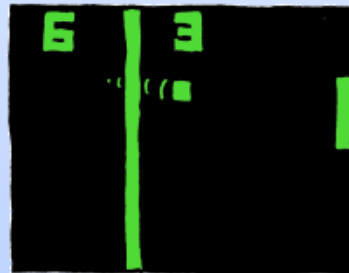
Something like
PONG!



SO WHAT DO WE DO IF
VIDEO GAME AI OPPONENTS
BECOME SMART ENOUGH TO
QUESTION THE "MATRIX" INTO
WHICH WE'VE PUT THEM?



WAIT A MINUTE! NONE OF THIS IS REAL!
I CAN SEE THROUGH THE WORLD!
I CAN SEE THE CODE!
I AM THE ONE!



Projecto final (caderno encargos, desenho, implementação de um projecto)

Ex:

Next: laboratory

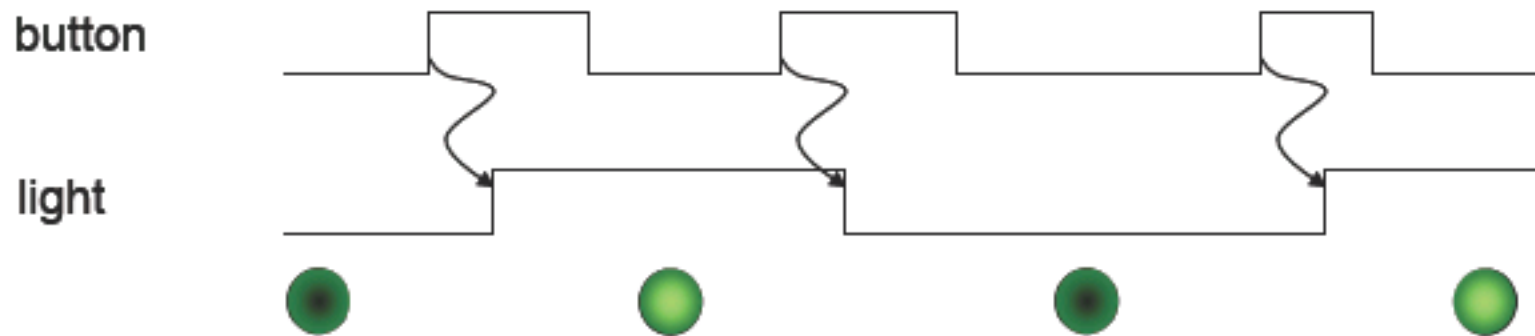
- 1) “Ola Mundo”
- 2) Descodificador 7 segmentos
- 3) Contadores

4) Fazia-me imenso jeito um cronómetro...

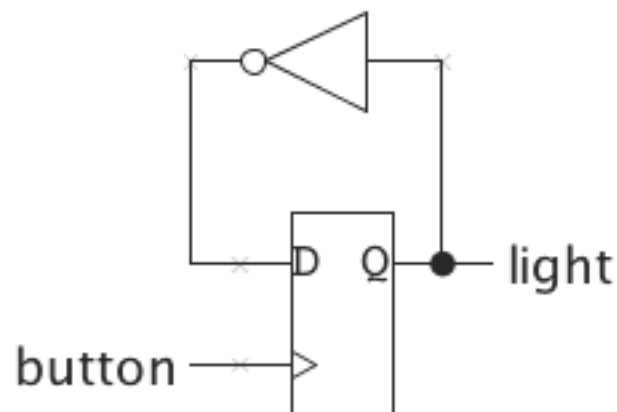
- Minutes : seconds : thousandths of seconds
- Start
- Stop
- reset



Implementation for on/off button



```
module onoff(input button, output reg light);  
    always @(posedge button) light <= ~light;  
endmodule
```



Synchronous on/off button

When designing a system that accepts many inputs it would be hard to have input changes serve as the system clock (which input would we use?). So we'll use a single clock of some fixed frequency and have the inputs control what state changes happen on rising clock edges.

```
module onoff_sync(input clk, button,
                  output reg light);
    always @ (posedge clk) begin
        if (button) light <= ~light;
    end
endmodule
```



Resetting to a known state

Usually one can't rely on registers powering-on to a particular initial state*. So most designs have a RESET signal that when asserted initializes all the state to known, mutually consistent initial values.

```
module onoff_sync(input clk, reset, button,
                  output reg light);
    always @ (posedge clk) begin
        if (reset) light <= 0;
        else if (button) light <= ~light;
    end
endmodule
```

* Actually, our FPGAs will reset all registers to 0 when the device is programmed. But it's nice to be able to press a reset button to return to a known state rather than starting from scratch by reprogramming the device.



Clocks are fast, we're slow!

The circuit on the last slide toggles the light on every rising clock edge for which button is 1. But clocks are fast (27MHz!) and our fingers are slow, so how do we press the button for just one clock edge? Answer: we can't, but we can add some state that remembers what button was last clock cycle and then detect the clock cycles when button changes from 0 to 1.

```
module onoff_sync(input clk, reset, button,
                  output reg light);
    reg old_button; // state of button last clk
    always @ (posedge clk) begin
        if (reset)
            begin light <= 0; old_button <= 0; end
        else if (old_button==0 && button==1)
            // button changed from 0 to 1
            light <= ~light;
            old_button <= button;
        end
    endmodule
```



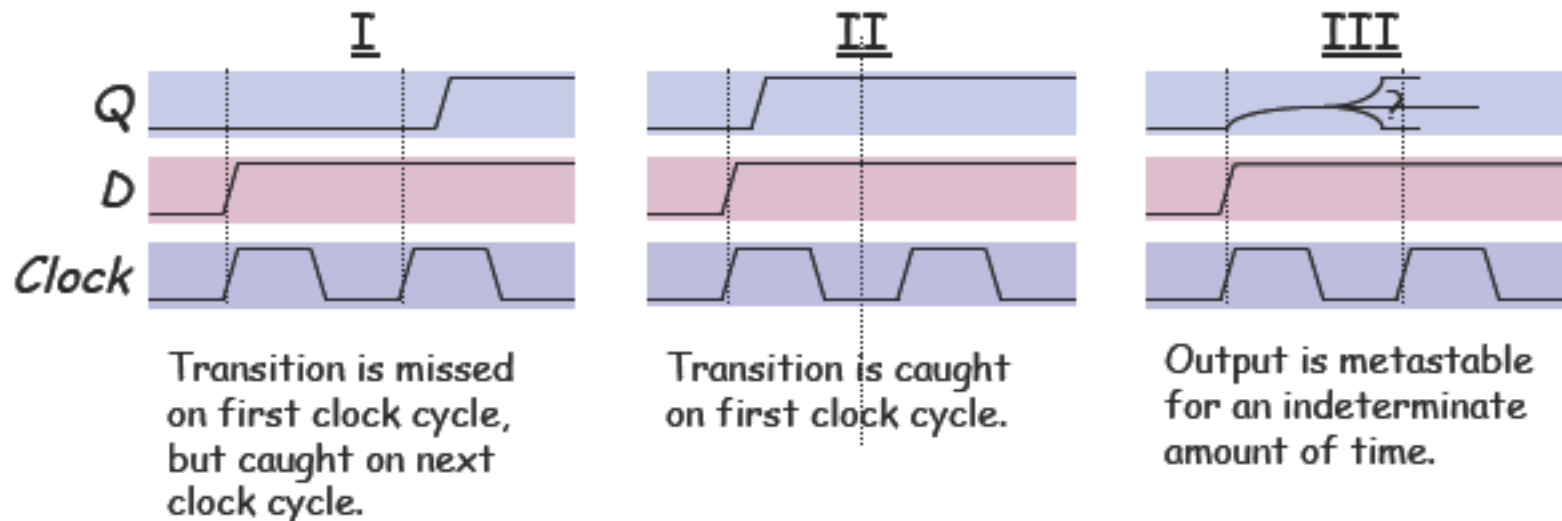

Asynchronous Inputs in Sequential Systems

What about external signals?



*Can't guarantee
setup and hold
times will be met!*

When an asynchronous signal causes a setup/hold violation...



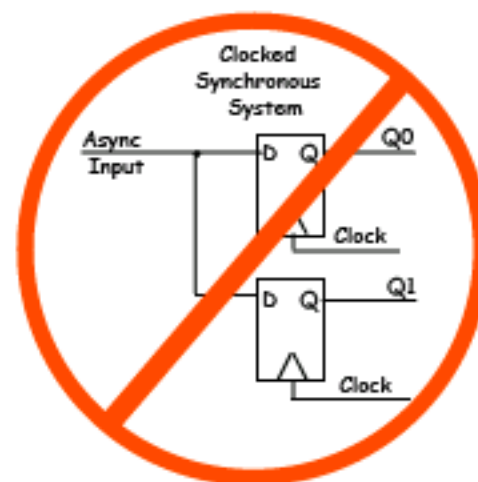
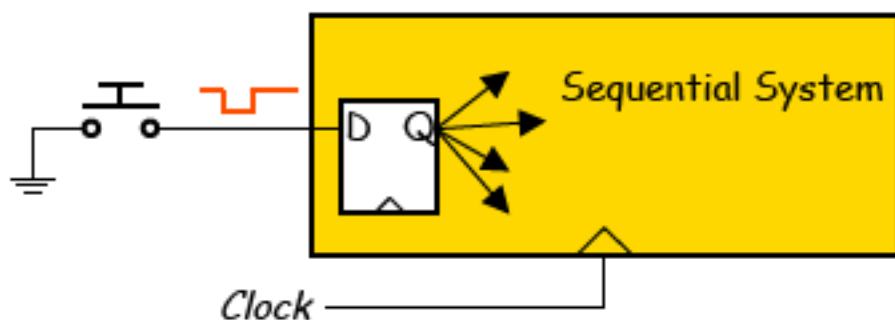
Q: Which cases are problematic?



Asynchronous Inputs in Sequential Systems

All of them can be, if more than one happens simultaneously within the same circuit.

Guideline: ensure that external signals directly feed exactly one flip-flop

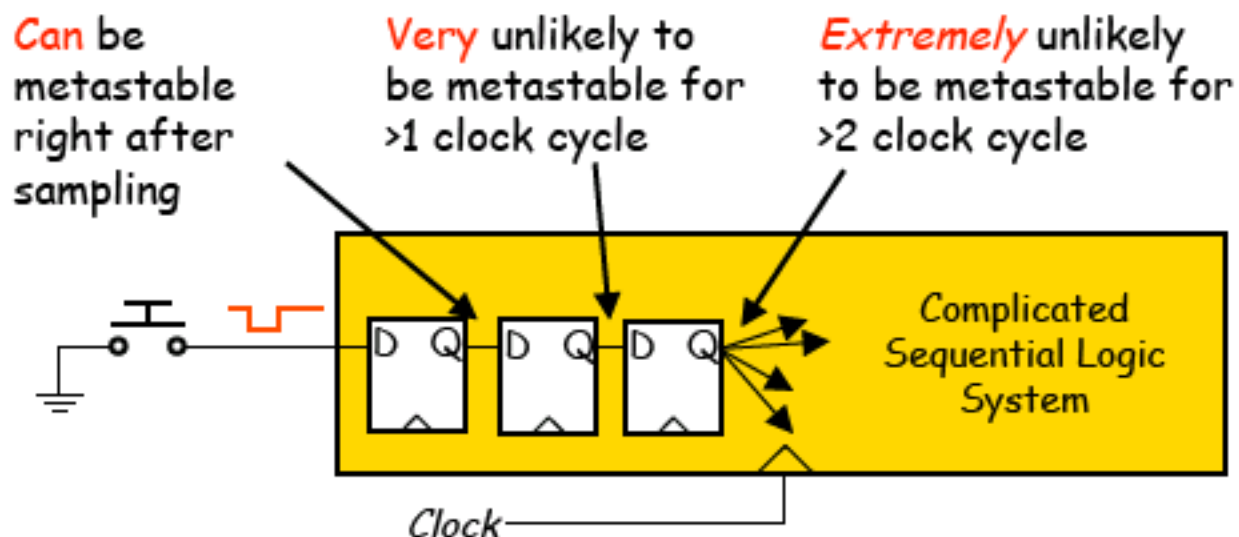


This prevents the possibility of I and II occurring in different places in the circuit, but what about metastability?



Handling Metastability

- Preventing metastability turns out to be an impossible problem
- High gain of digital devices makes it likely that metastable conditions will resolve themselves quickly
- Solution to metastability: allow time for signals to stabilize



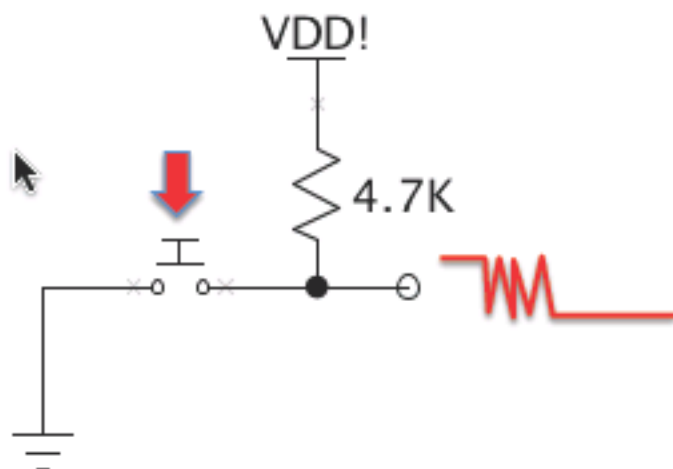
How many registers are necessary?

- Depends on many design parameters (clock speed, device speeds, ...)
- In 6.111, a pair of synchronization registers is sufficient



One last little problem...

Mechanical buttons exhibit contact "bounce" when they change position, leading to multiple output transitions before finally stabilizing in the new position:



We need a debouncing circuit!

```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
// DELAY = .01 sec with a 27Mhz clock
module debounce #(parameter DELAY=270000)
    (input reset, clock, noisy,
     output reg clean);

    reg [18:0] count;
    reg new;

    always @(posedge clock)
        if (reset) // return to known state
            begin
                count <= 0;
                new <= noisy;
                clean <= noisy;
            end
        else if (noisy != new) // input changed
            begin
                new <= noisy;
                count <= 0;
            end
        else if (count == DELAY) // stable!
            clean <= new;
        else // waiting...
            count <= count+1;

endmodule
```

On/off button: final answer

```
module onoff_sync(input clk, reset, button_in,
                  output reg light);
    // synchronizer
    reg button,btemp;
    always @(posedge clk)
        {button,btemp} <= {btemp,button_in};

    // debounce push button
    wire bpressed;
    debounce db1(.clock(clk),.reset(reset),
                .noisy(button),.clean(bpressed));

    reg old_bpressed; // state last clk cycle
    always @ (posedge clk) begin
        if (reset)
            begin light <= 0; old_bpressed <= 0; end
        else if (old_bpressed==0 && bpressed==1)
            // button changed from 0 to 1
            light <= ~light;
            old_bpressed <= bpressed;
        end
    endmodule
```