TÉCNICO LISBOA

Projecto e Controlo em Lógica Digital

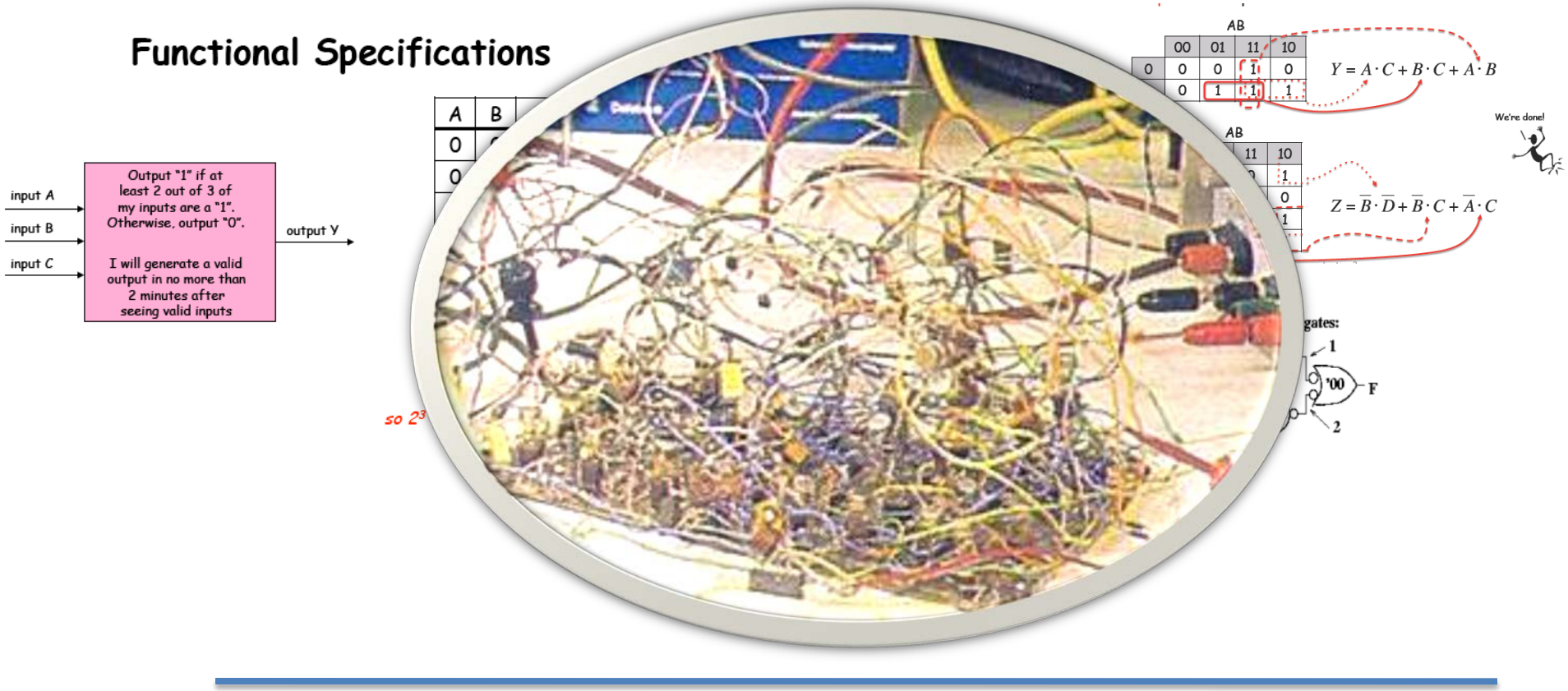
www.lip.pt/~pedjor/PCLD

- Combinational logic
- Modules
- Sequential Logic
- Tools

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

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

74LS…. vs FPGAs



+ HDL code

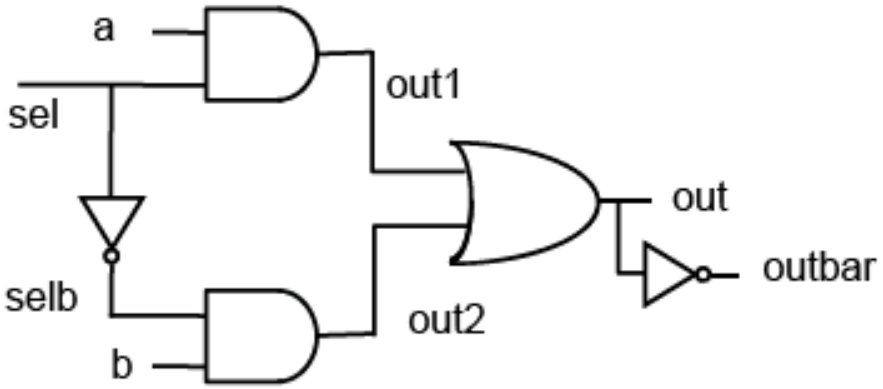
Gate Level Description

Verilog has built-in basic logic gates:
And, nand, or, nor, xor, xnor, not, buf

WIRE: represents connections between elements.

How to use:
wire out1; //declares there will be a connection named **OUT1**
and a1 (out1, input1, input2); //meaning:
//implement an **AND** gate
//named **A1**
//the output is **OUT1**
//the inputs are **INPUT1**, **INPUT2**

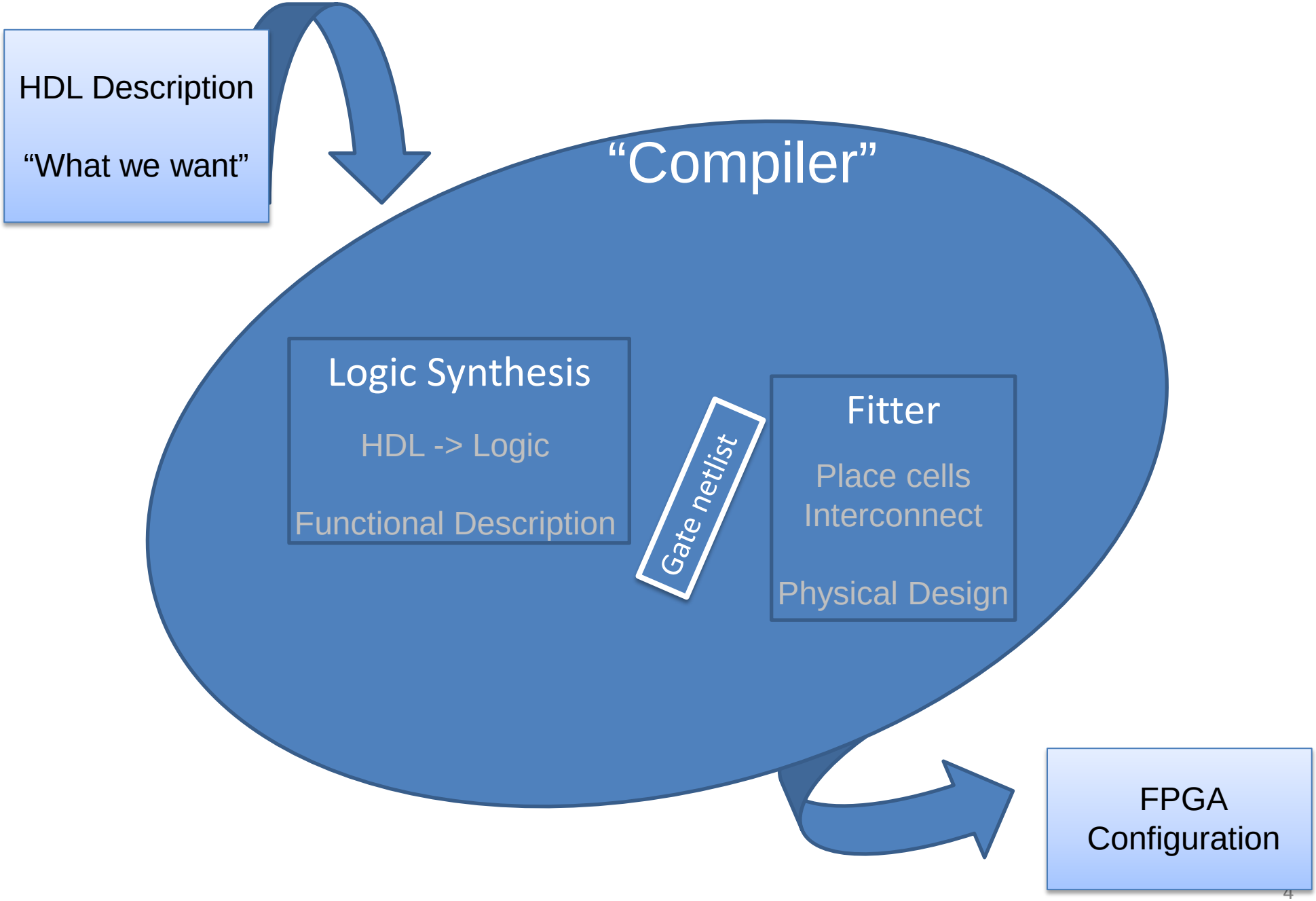
Building a multiplexer using gates




```
module mux(input a,b,sel,
           output out, outbar);
   wire selb, out1, out2;
   not n1(selbar, sel);
   and a1(out1, a, sel);
   and a2(out2, b, selb);
   or o1(out, out1, out2);
   not n2(outbar, out)
endmodule
```

The HDL

HDL = Hardware Description Language



Verilog data values

Value	Meaning
0	Logic zero, “Low”
1	Logic one, “High”
Z or ?	High Impedance (tri-state)
X	Unknow (simulation)

Numeric constants

Full format: <Width>’<Radix>value
Width: number
Radix: d=decimal, h=hex, o=ocatl, b=binary

Value	Meaning
123	Default: decimal radix
‘d123	‘d=decimal radix
‘h7B	‘h=hexadecimal radix
‘o173	‘o=ocatl radix
‘b111_101	‘b=binary radix
16’b11111	A binary with 16 bits
16’d5	A 16 bit decimal = ‘b0000_0000_0000_0101

Boolean operators

- Bitwise operators perform bit-oriented operations on vectors
  - ~(4'b0101) = {~0,~1,~0,~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

~a	NOT
a & b	AND
a | b	OR
a ^ b	XOR
a ~^ b a ^~ b	XNOR

Reduction

&a	AND
~&a	NAND
|a	OR
~|a	NOR
^a	XOR
~^a ^~a	XNOR

Note distinction between ~a and !a when operating on multi-bit values

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

Other operators

Conditional

a ? b : c	If a then b else c

Relational

a > b	greater than
a >= b	greater than or equal
a < b	Less than
a <= 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 << b	logical left shift
a >> b	logical right shift
a <<< b	arithmetic left shift
a >>> b	arithmetic right shift

How many bits?

Wire ab;	A 1 bit wire called ab
Wire ab,cd;	Two 1 bit wires called ab and cd
wire [31:0] ef;	A 32 bits wire bus called ef;
{ab,cd}	Concatenation of ab and cd;
ef[15]	Bit #15 (the sixteenth) of ef
ef[7:0]	First 8 bits of ef (the ones to the right

What does this means?

```
wire [31:0] kk;
Wire [7:0] a,b,c,d;
Assign kk={d,c,b,a}
```

Note:
wire [7:0]a;
wire [0:7]b;
The two forms can be used. Prone to error if mixed.
Standard convention: [MSB:LSB] w/ MSB>LSB & LSB=0

[WIDTH-1:0]

The REG keyword

reg = register
is intended as a variable type

LeftHandSide inside always blocks must be reg!

For historical reasons...
reg is not always a clocked register (althought it would make more sense)

reg can be used directly in the port declaration:

```
output reg [15:0] result
```

or in the declaration of a net:

```
reg a;
reg[31:0] b;
```

Do not use reg with assigns:

```
reg out;
assign out=0;
```

Continuous Assignments
(aka dataflow)

The assign construction
assign LHS = RHS ;
can use operators...

```
wire a,b,c,d,e,f,g;
assign a=b;
assign c=!d;
assign e=f+g;
```

Sequential behaviours
(aka procedural)

The “always @” construction
always @ (sensitivity list)
begin
LHS=RHS;
end
can use operators, if, case, ...

```
wire b,d,f,g;
reg a,c,e;
always @ (b,d,f,g)
begin
  a=b;
  c=!d;
  e=f+g;
end
```

can be:
`always @ (*)`

The multiplexer using...

ASSIGN

```
// 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
```

ALWAYS @ IF...THEN...ELSE

```
// 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
```

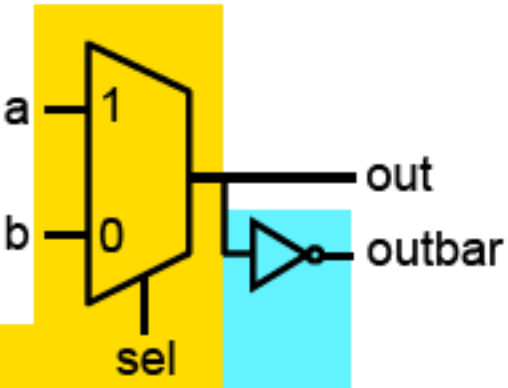
ALWAYS @ CASE

```
// 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
```

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
```



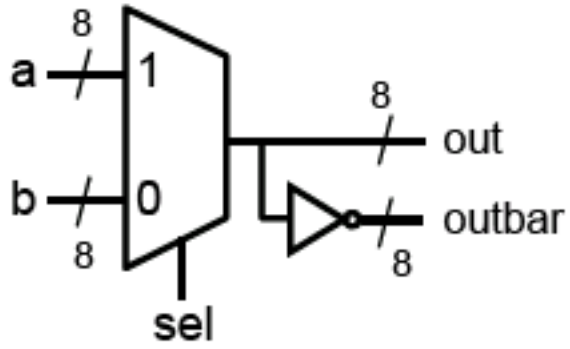
procedural description

continuous description

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:**

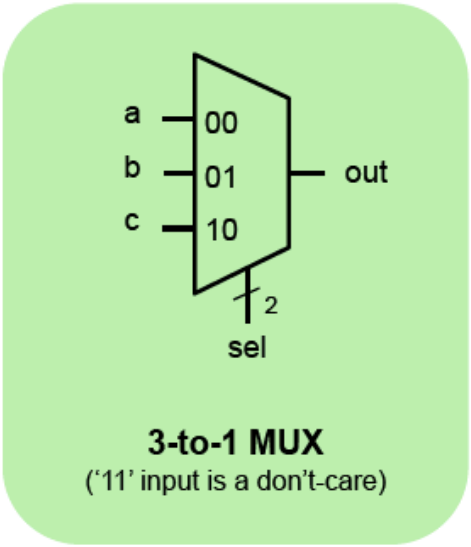
```
module add32(a, b, sum);
  input[31:0] a,b;
  output[31:0] sum;
  assign sum = a + b;
endmodule
```

- **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
```

Danger

Goal:



3-to-1 MUX
('11' input is a don't-care)

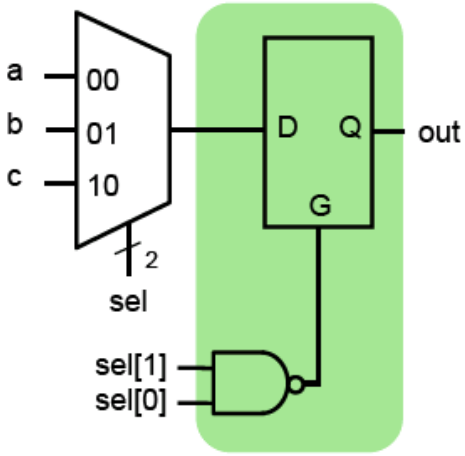
Proposed Verilog Code:

```
module maybe_mux_3to1(a, b, c
                      sel, ou
  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;
    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
```

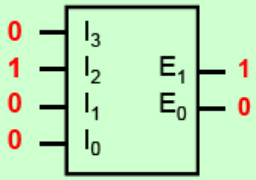
```
always @(a or b or c or sel)
  begin
  case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
      default: out = 1'bx;
    endcase
  end
endmodule
```

- …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:

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

Proposed Verilog Code:

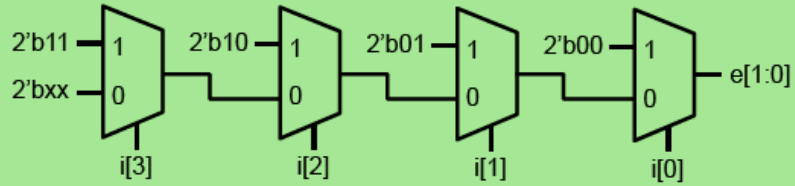
```
module binary_encoder(i, e);
  input [3:0] i;
  output [1:0] e;
  reg [1:0] e;

  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
```

What is the resulting circuit?

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

```
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:



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

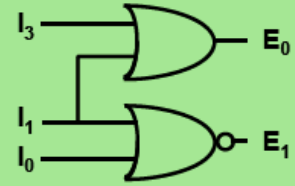
Solution:

- **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
```

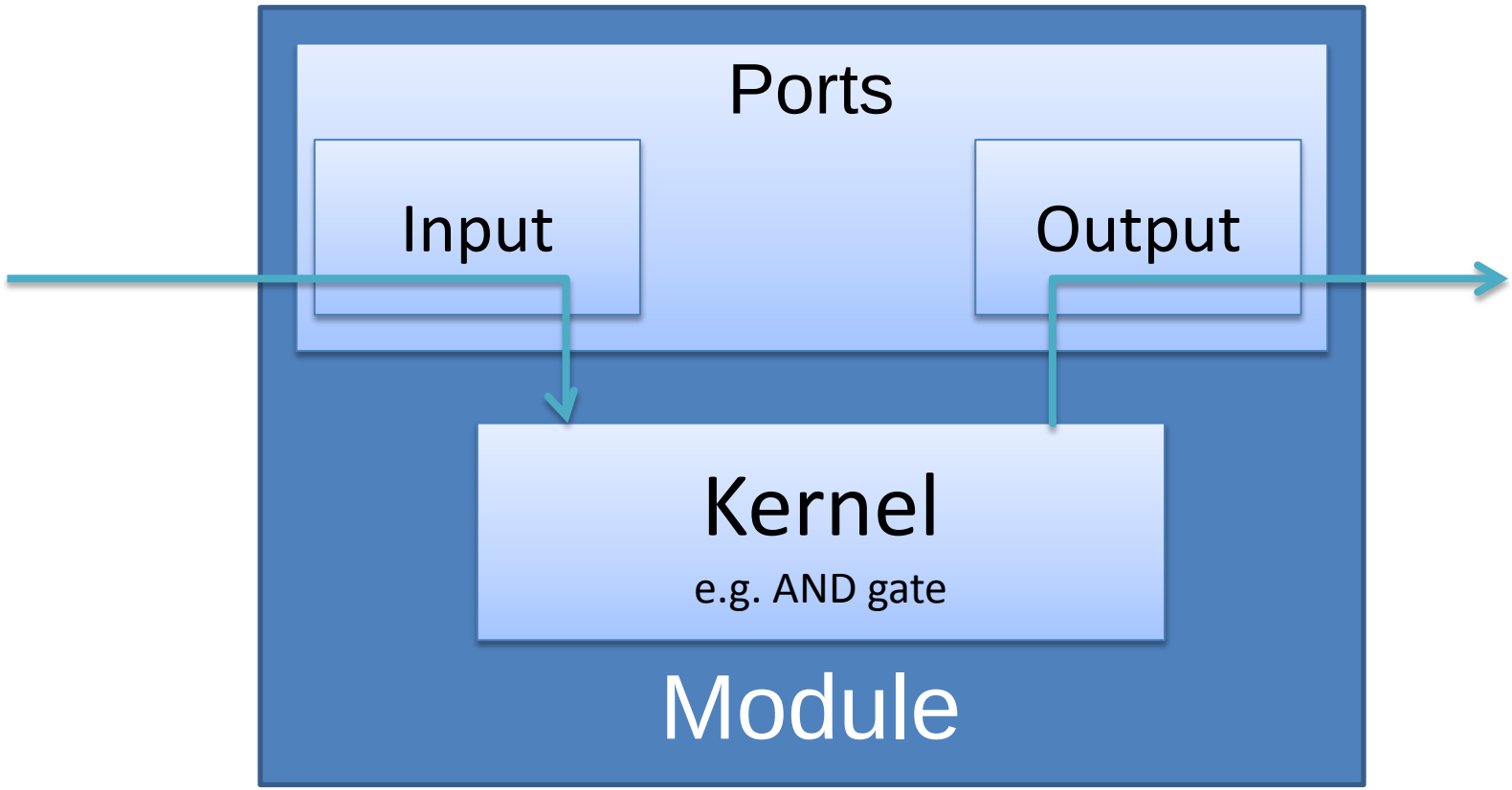
Minimized Result:



Modules, again, and again and again...

Structuring... Modules

Modules are the building blocks in Verilog!
There is at least one module: The top one that has the name of the project!



Ports in the top level module link directly to the pins

Modules with different juice...

```
module mux(input a,b,sel, output q, qbar);
```

```
wire selbar, q1, q2;
not n1(selbar, sel);
and a1(q1, a, selbar);
and a2(z2, b, sel);
or o1(q, q1, q2);
not n2(qbar, q)
```

```
assign q = sel? a : b;
assign qbar = ~ q;
```

```
always @ (a,b,sel)
begin
  if (sel) q=a;
  else q=b;
end
assign qbar = ~ q;
```

```
endmodule
```

We just need to know the shell, the interface, and its function!

Call the port by its name!

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

Option 1: Know the order and write all ports...

mux my_mux (add_out, sub_out, mul_out, f[1:0], res);

Option 2: Know their names...

syntax: module_type module_name (.port_name(net_name), .port2_name(net2_name));

```
mux my_mux (.i0(add_out),
            .i1(sub_out),
            .i2(mul_out),
            .sel(f[1:0]),
            .out(res));
```

port name

net

OR

```
mux my_mux (.sel(f[1:0]),
            .out(res),
            .i0(add_out),
            .i1(sub_out),
            .i2(mul_out),
            );
```

OR even

```
mux my_mux (.sel(2'b00),
            .out(res),
            .i0(add_out),
            );
```

Parametrized modules

could be useful if we could build “generic” modules
for instance a mux for data buses that can be 1 bit to 32bit
if we have a parameter that seems easy

THE #(PARAMETER ....)

```
module mux #(parameter W=1)
             (input [W-1:0 i0,i1]
             output [W-1] out
             input sel);
       assign out= sel ? i1 : i0;
endmodule
```

parameter named “W”
default=1

Using a parametrized module

```
wire a,b,c,z;
mux #(.W(1)) m1(.sel(a),.i0(b),.i1(c),.out(z));


wire aa;
wire [15:0] bb,cc,zz;
mux #(.W(16)) m1(.sel(aa),.i0(bb),.i1(cc),.out(zz));
```

and of course... be creative...

You can instantiante lots of modules that feeds one another...
Each instance is like a “new chip” of that type
Each instance must have different names

How many instances can drive a net??

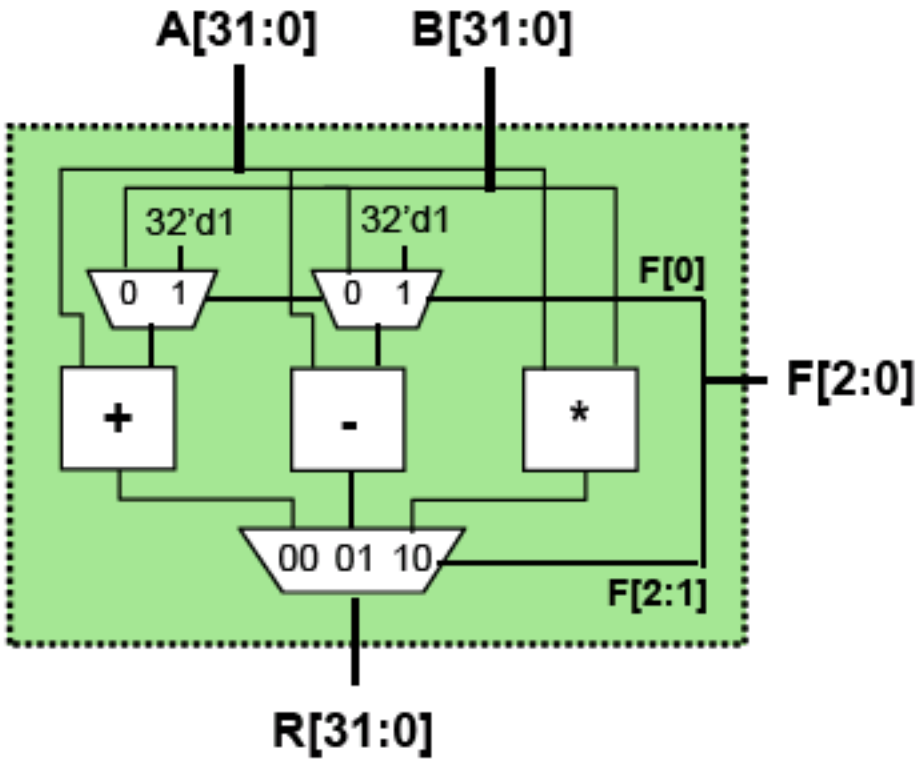
Sometimes drawing in a paper is useful...
just draw boxes, don’t need to be an artist...

```
// 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
```

Build an ALU

Connecting modules to build complex machines

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 Subtracter

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

assign diff = i0 - i1;

endmodule
```

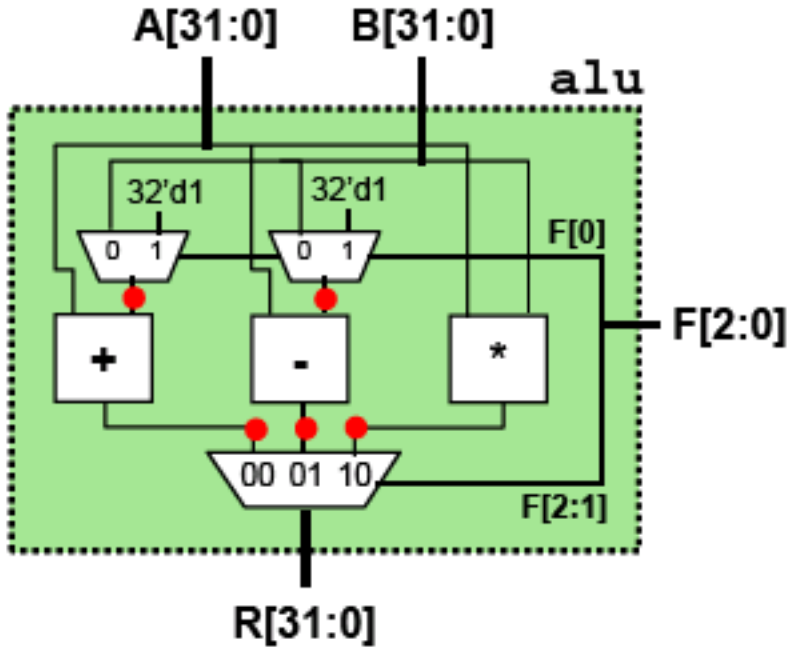
16-bit Multiplier

```
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_subtracter(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
```

intermediate output nodes

module names

(unique) instance names

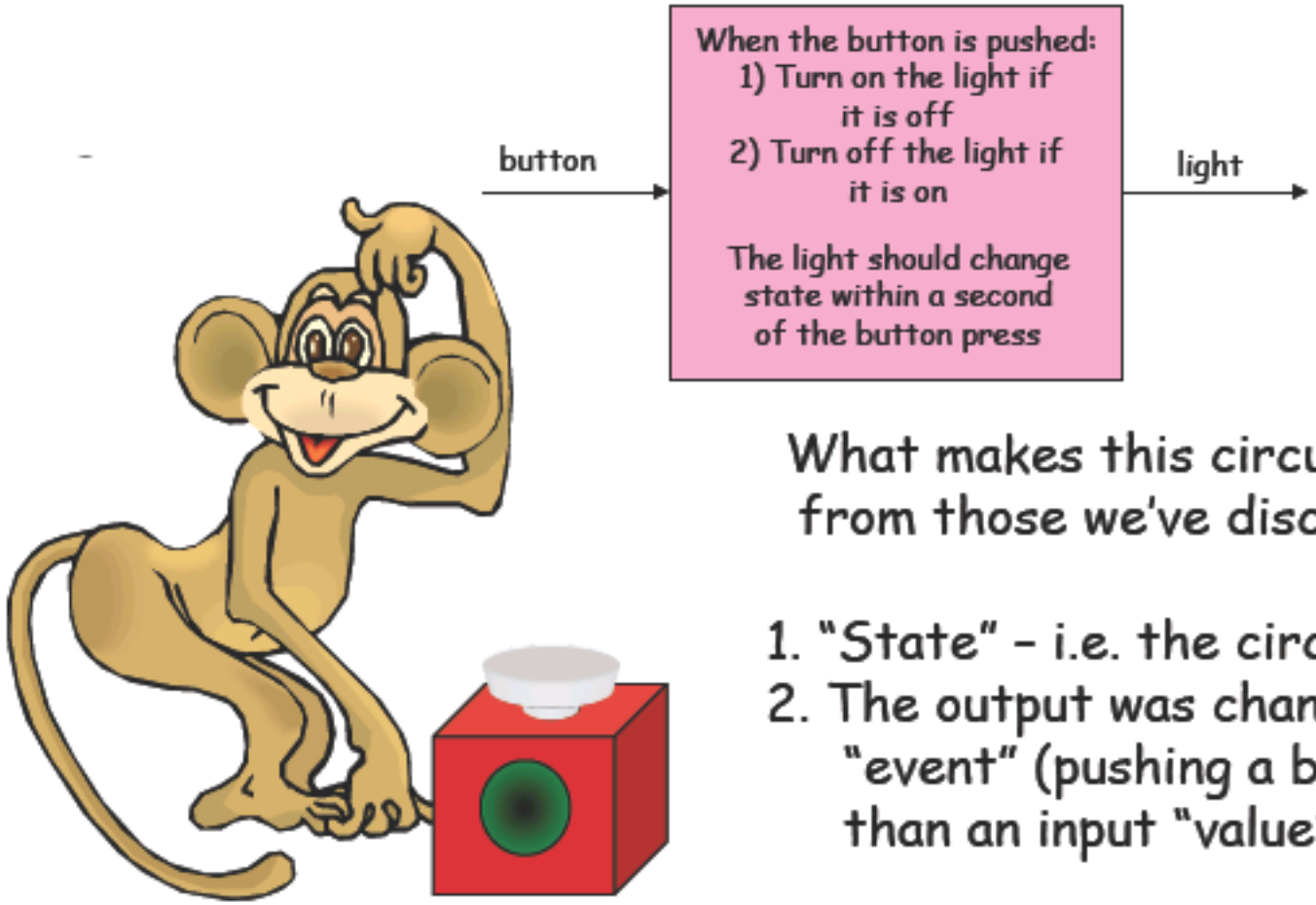
corresponding wires/regs in module alu

Until here: Combinational logic

Sequential Logic

Something We Can't Build (Yet)

What if you were given the following design specification:

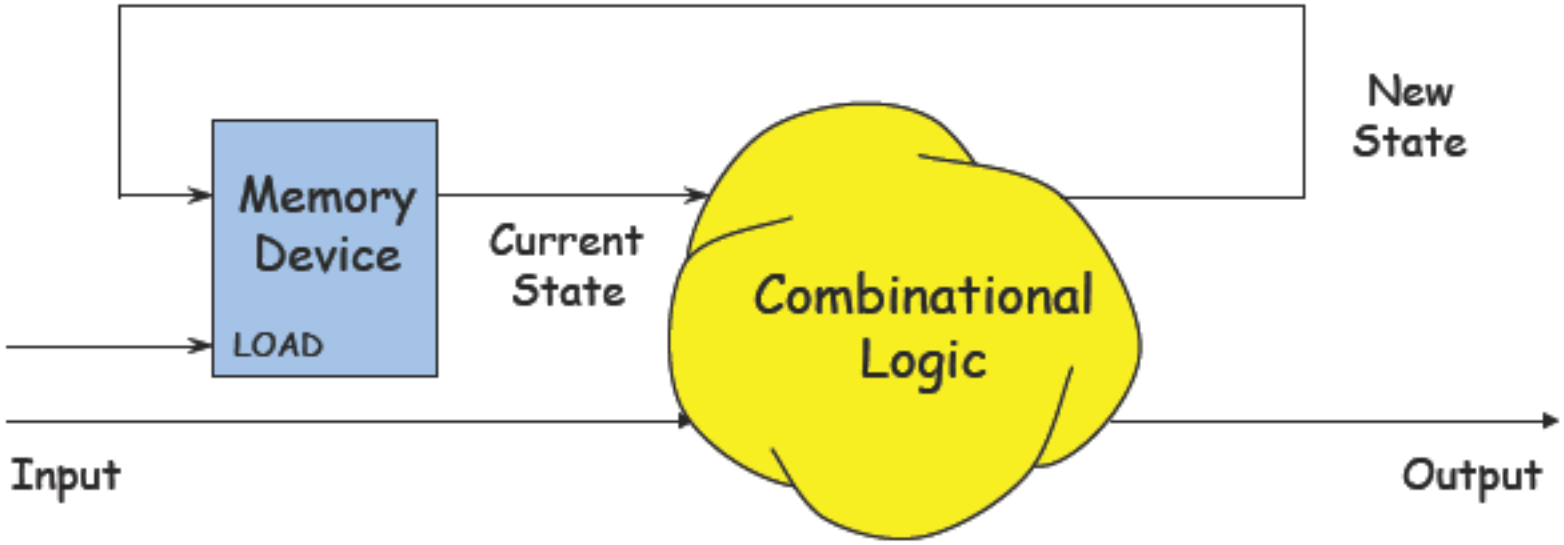


What makes this circuit so different from those we've discussed before?

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



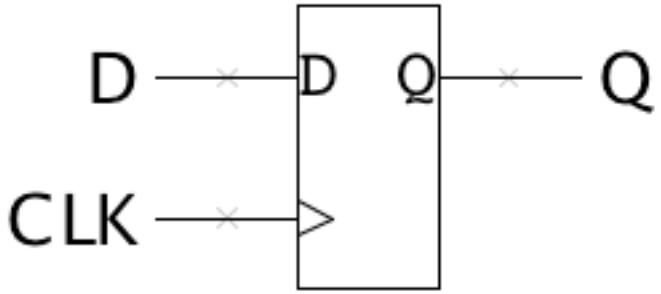
Plan: Build a Sequential Circuit with stored digital STATE –

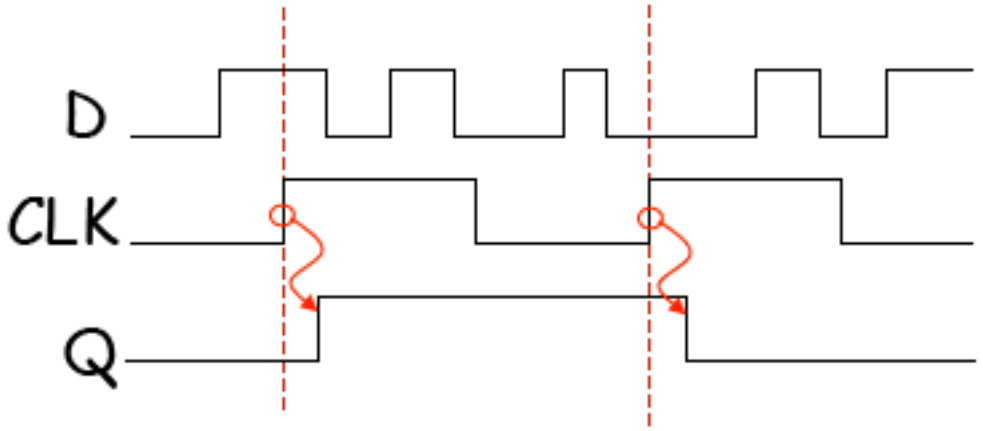
- Memory stores CURRENT state, produced at output
- Combinational Logic computes
  - NEXT state (from input, current state)
  - OUTPUT bit (from input, current state)
- State changes on LOAD control input

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.



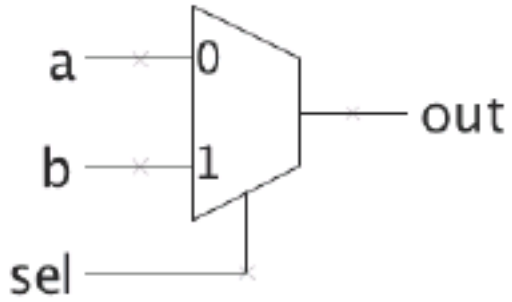


The Sequential always Block

Edge-triggered circuits are described using a sequential always block

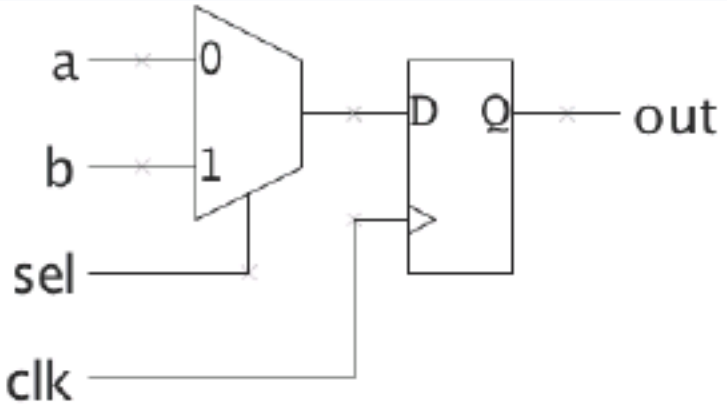
Combinational

```
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
```

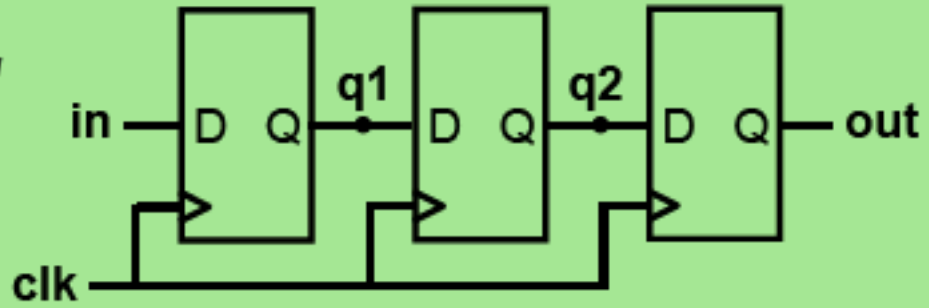
Nonblocking assignment (<=): all assignments deferred to end of simulation time step after <u>all</u> right-hand sides have been evaluated (*even those in other active* `always` *blocks*)

```
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?**

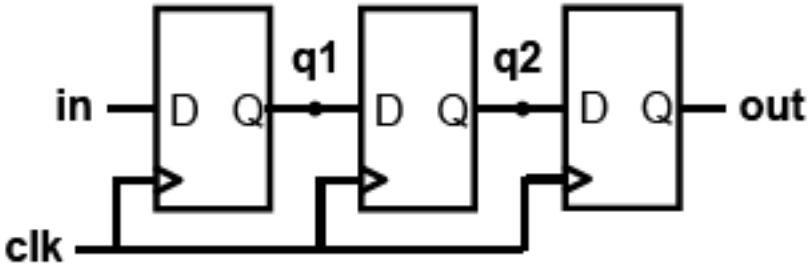
```
module nonblocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;

  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

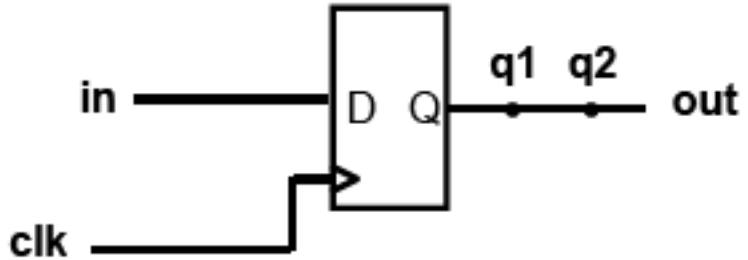
```
always @ (posedge clk)
begin
  q1 <= in;
  q2 <= q1;
  out <= q2;
end
```

"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
```

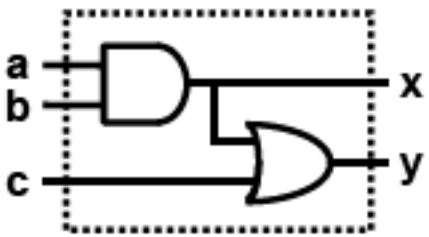
"At each rising clock edge, *q1* = *in*.
After that, *q2* = *q1* = *in*.
After that, *out* = *q2* = *q1* = *in*.
Therefore *out* = *in*."



- **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

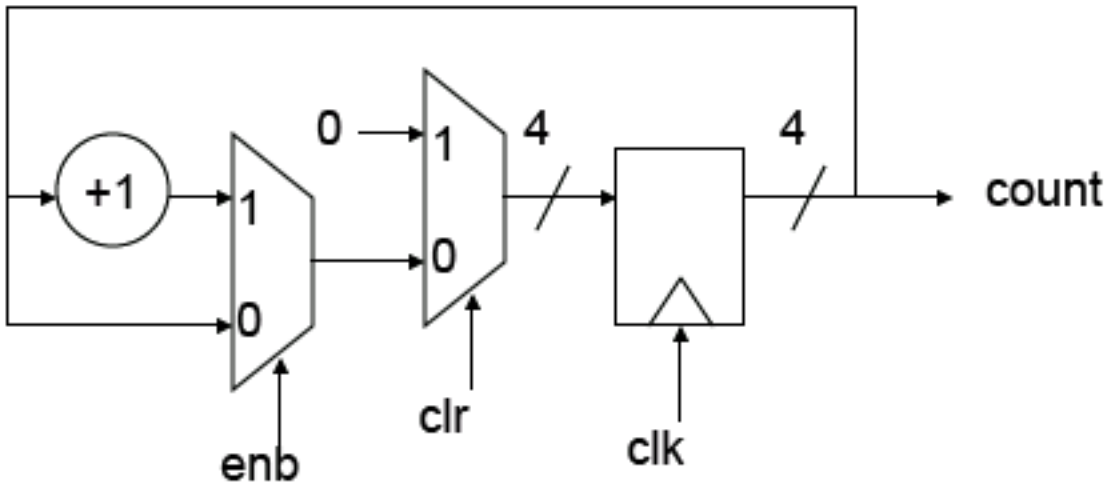
= vs <= inside always

```
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
```

Counter 32 bits in the leds...

```
reg [31:0] count ;

assign LEDR = count[17:0];
always @ (posedge CLOCK_50)
begin
        count <= count +1;
end
```

PWM – Pulse Width Modulation

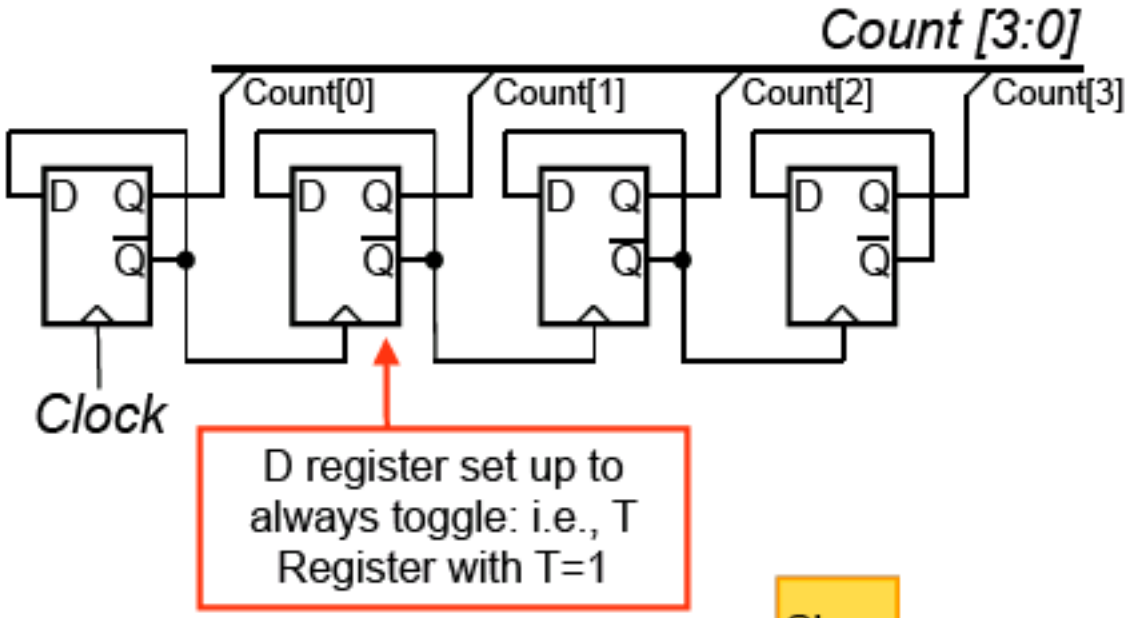
```
wire [7:0] H_PWM;
wire [3:0] bright;
assign bright=SW[3:0];
reg [3:0] PWM_count;
always @ (posedge CLOCK_50)
        PWM_count<=PWM_count+1;
assign H_PWM=(bright>PWM_count)? 7'hFF : 7'h00;
```

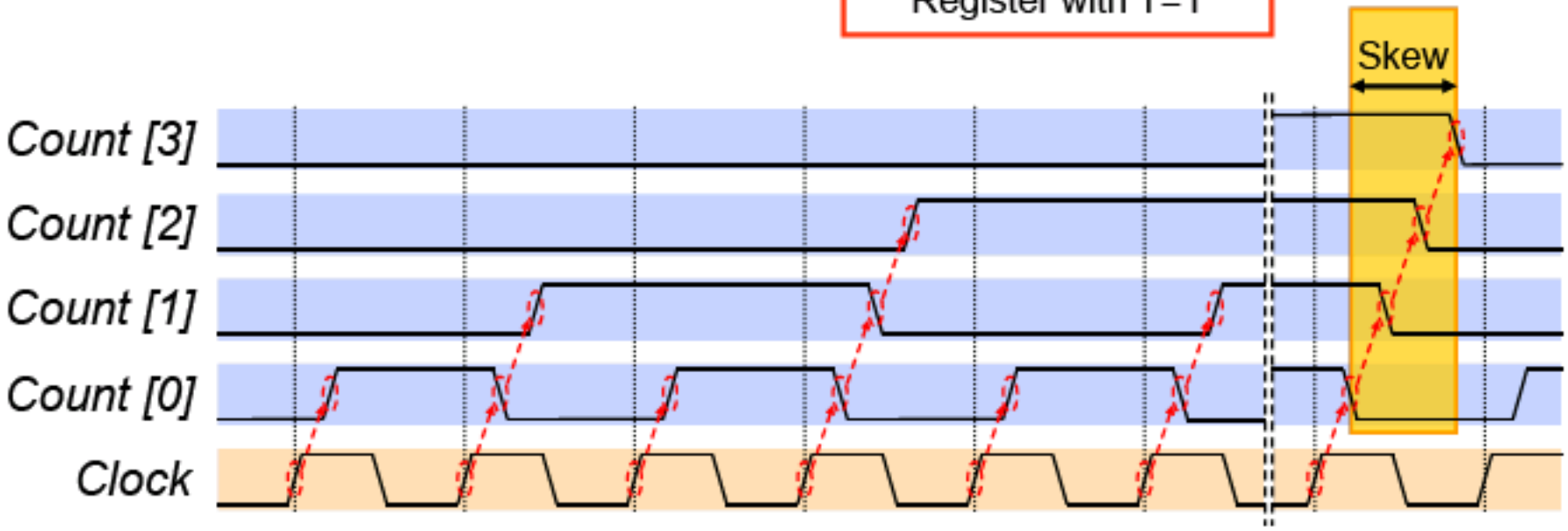
The Asynchronous counter

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

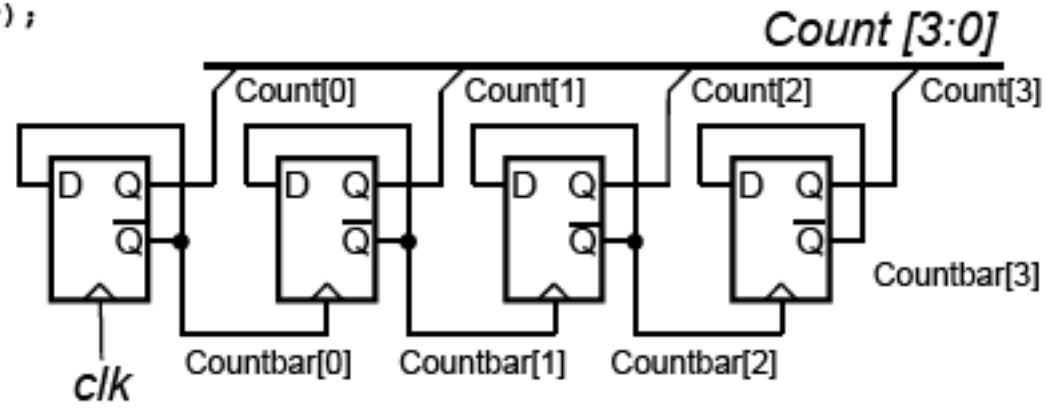




The ripple counter in verilog

Single D Register with Asynchronous Clear:

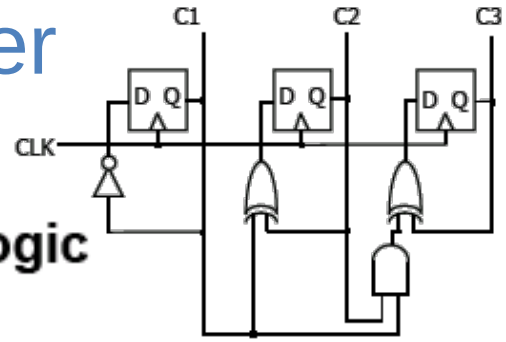
```
module dreg_async_reset (clk, clear, d, q, qbar);
input d, clk, clear;
output q, qbar;
reg q;

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
```

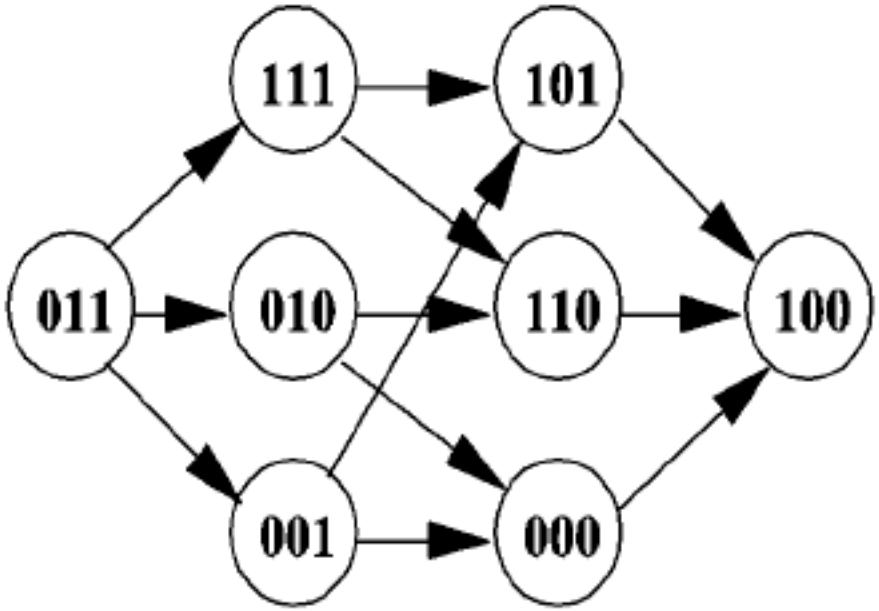
Logic for a synchronous counter



- Count (C) will 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

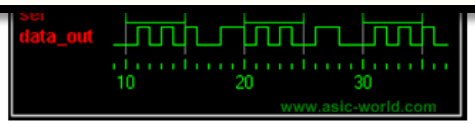
Verilog

Example - Mux

System Tasks & Functions

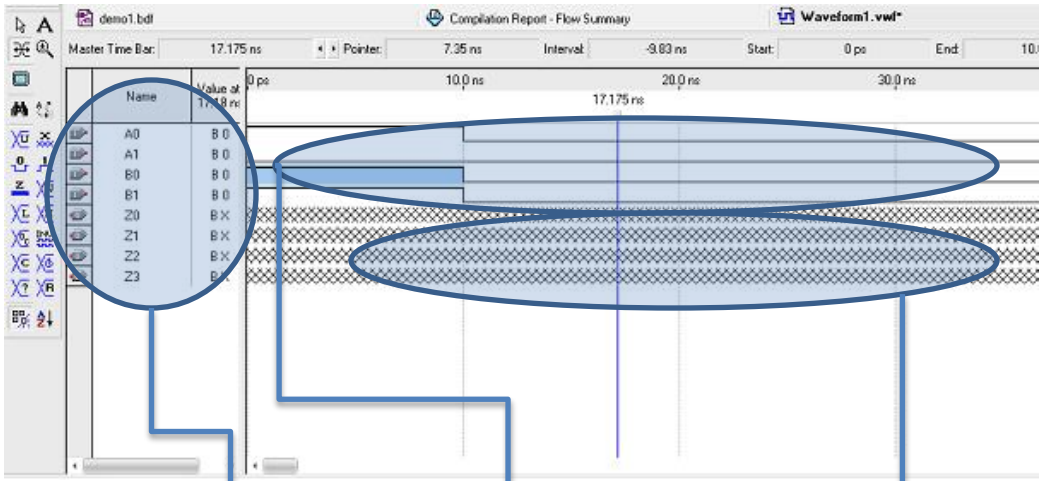
Quartus II support for system tasks and functions is described below. Se
Description Language Based on the Verilog Hardware Description Language

Section	Verilog HDL Construct	Quartus II Support *Note (1)*
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.



http://www.asic-world.com/verilog/timing_ctrl1.html#Example_-_clk_gen

Quartus II



Define the signals :
Inputs – stimulus
Outputs - results

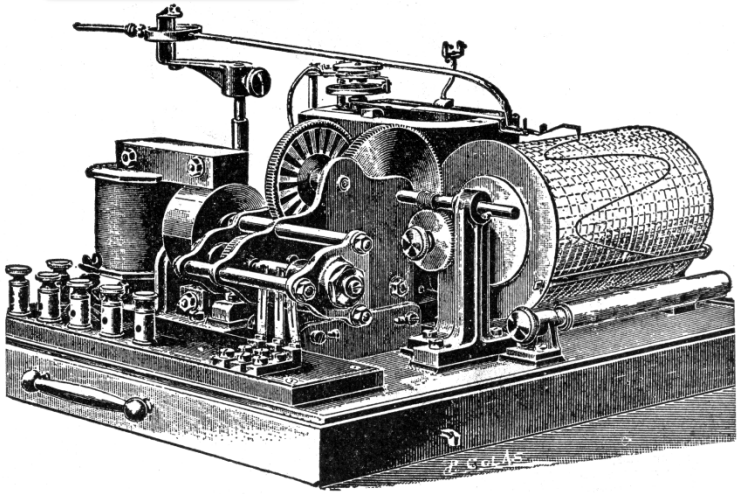
Define the stimulus to the system

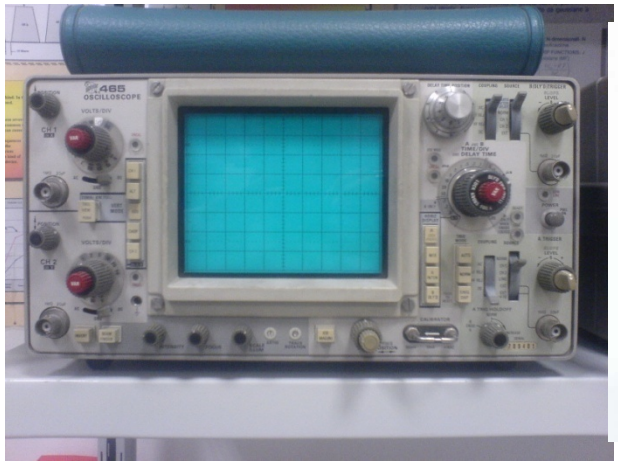
Run the simulation and you get the result

http://fpga4u.epfl.ch/wiki/Image:Simulation.jpg

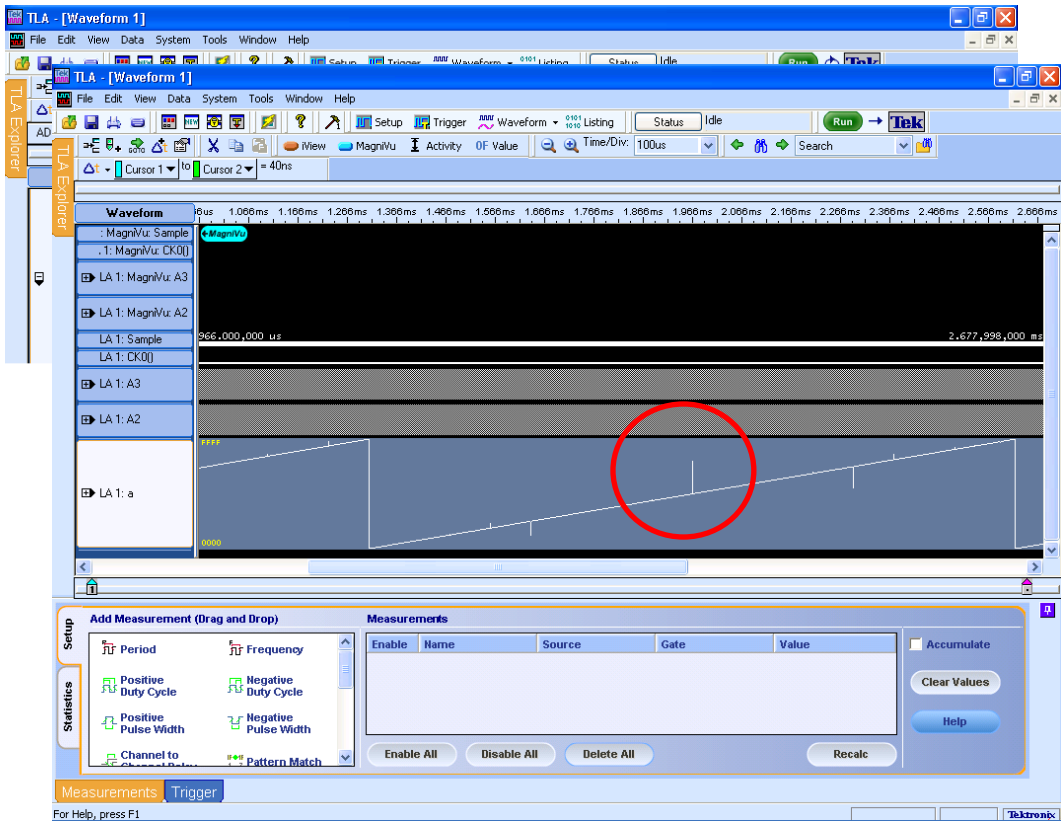
Tools – Measurement instruments

Waveforms





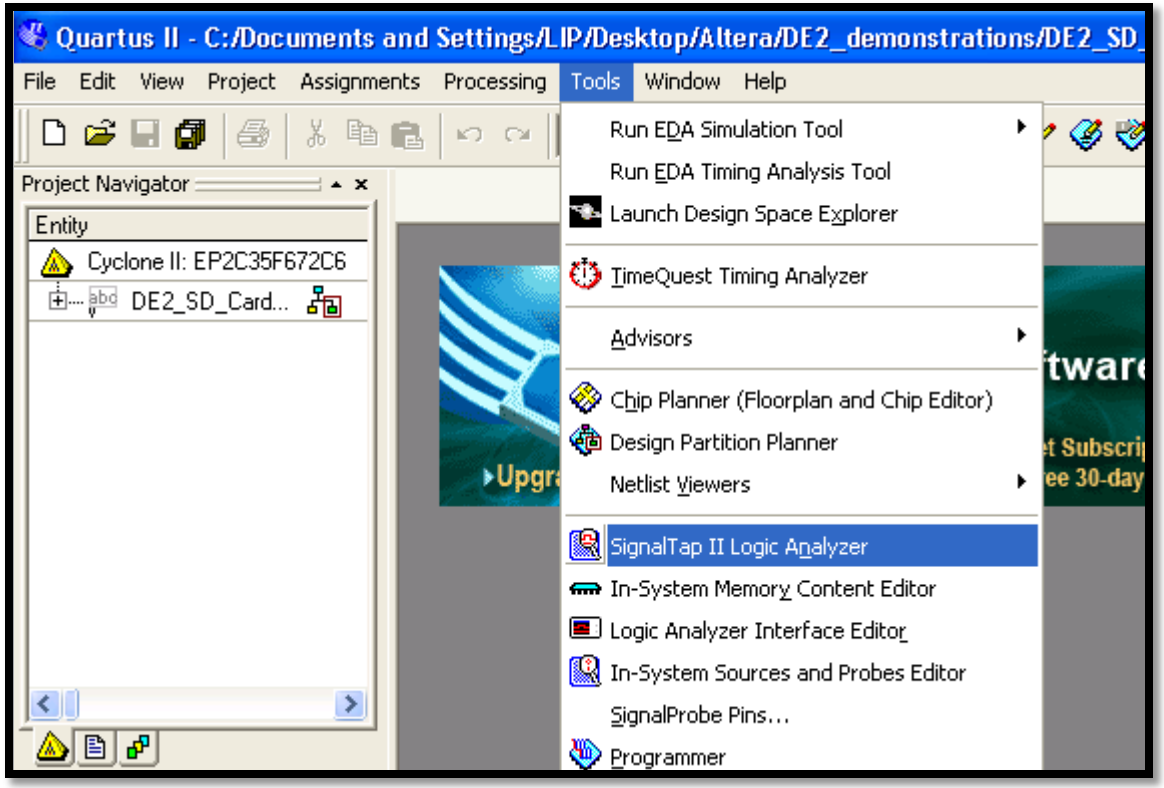
Logic Levels



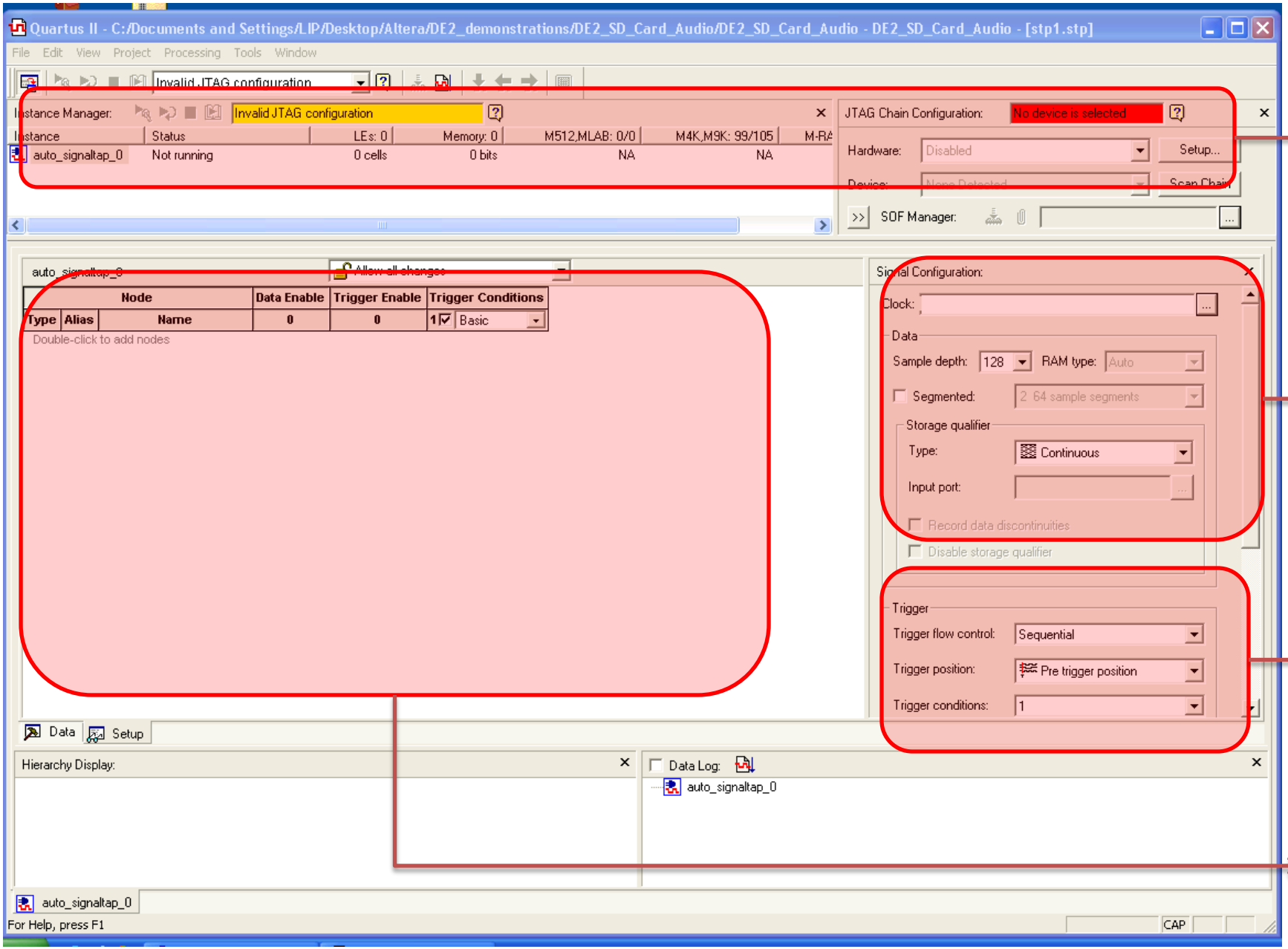
Tools – Internal Logic Analyser

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



CONTROL

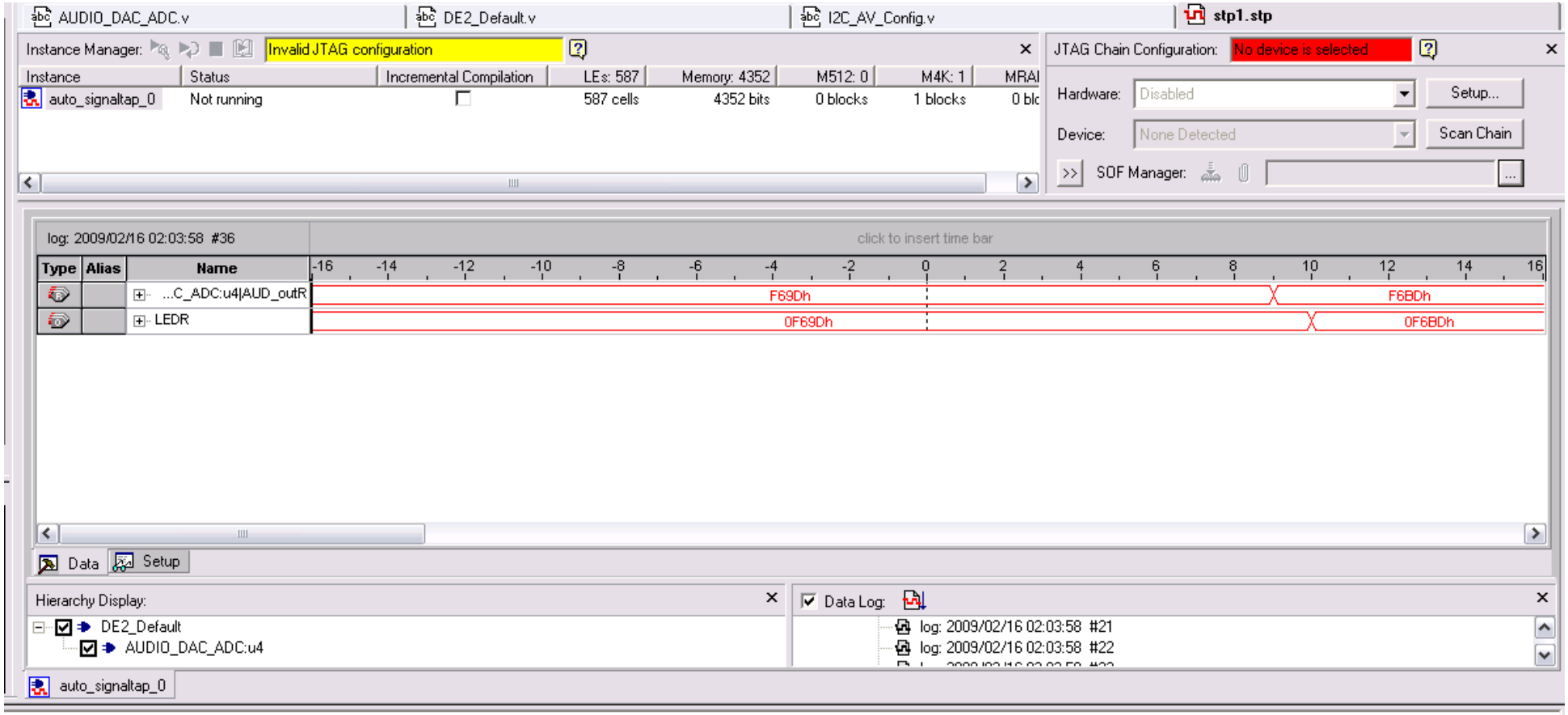
Clock definition

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

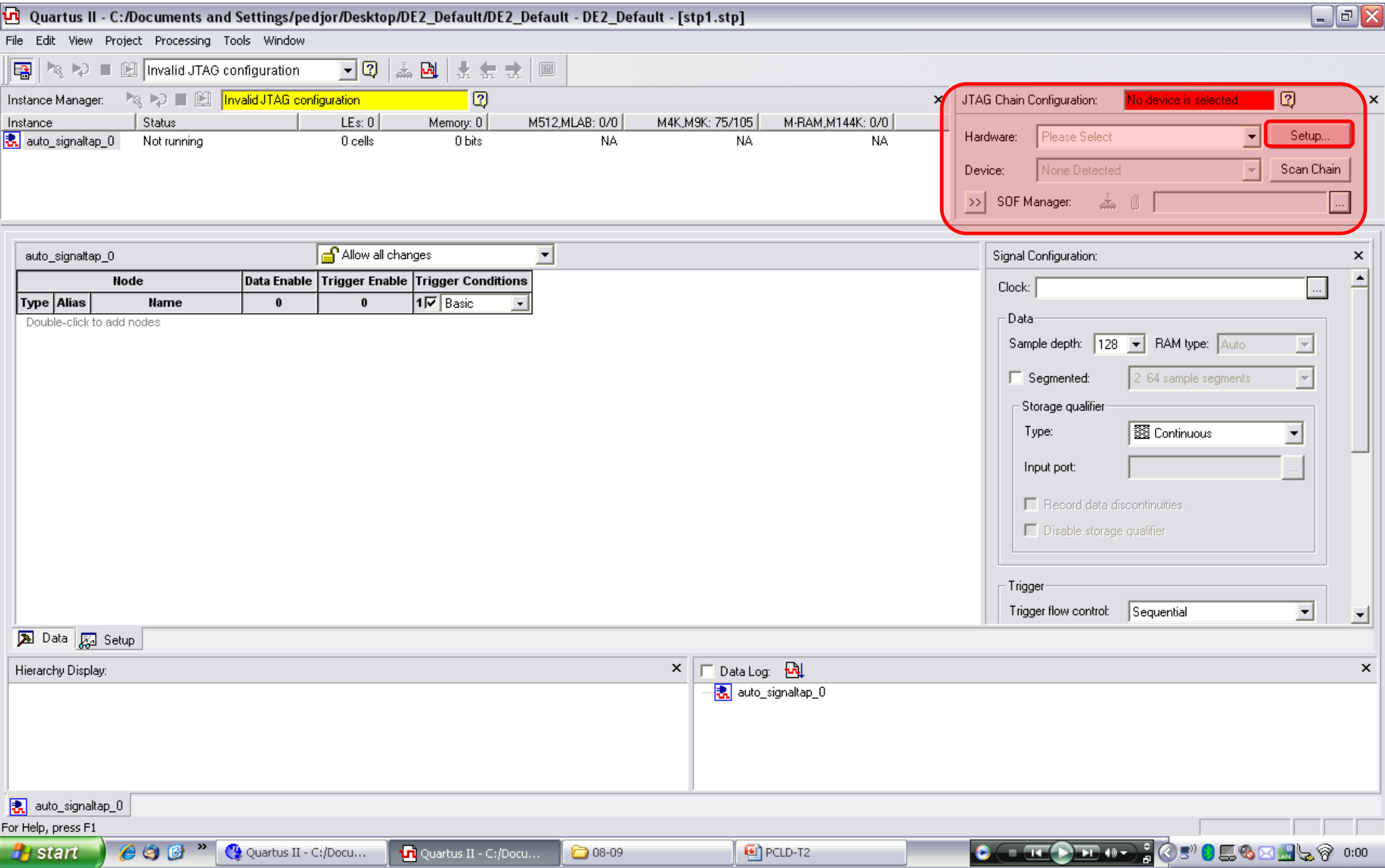


Tools – Signal probe

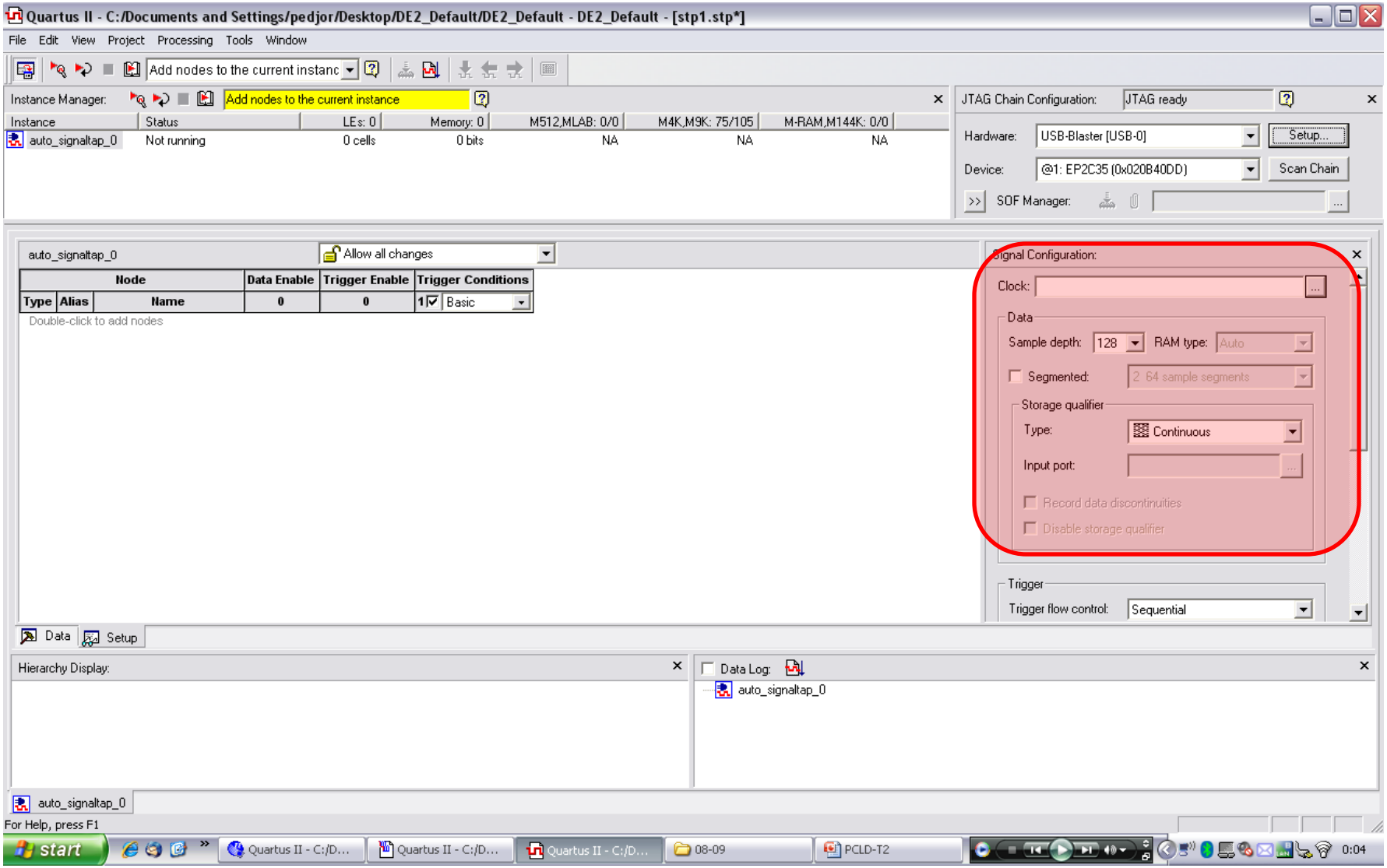
Internal signals can be extracted to output pins and connected to na external logic analyser. Signals can be exchanged easily…

SignalTap step by step

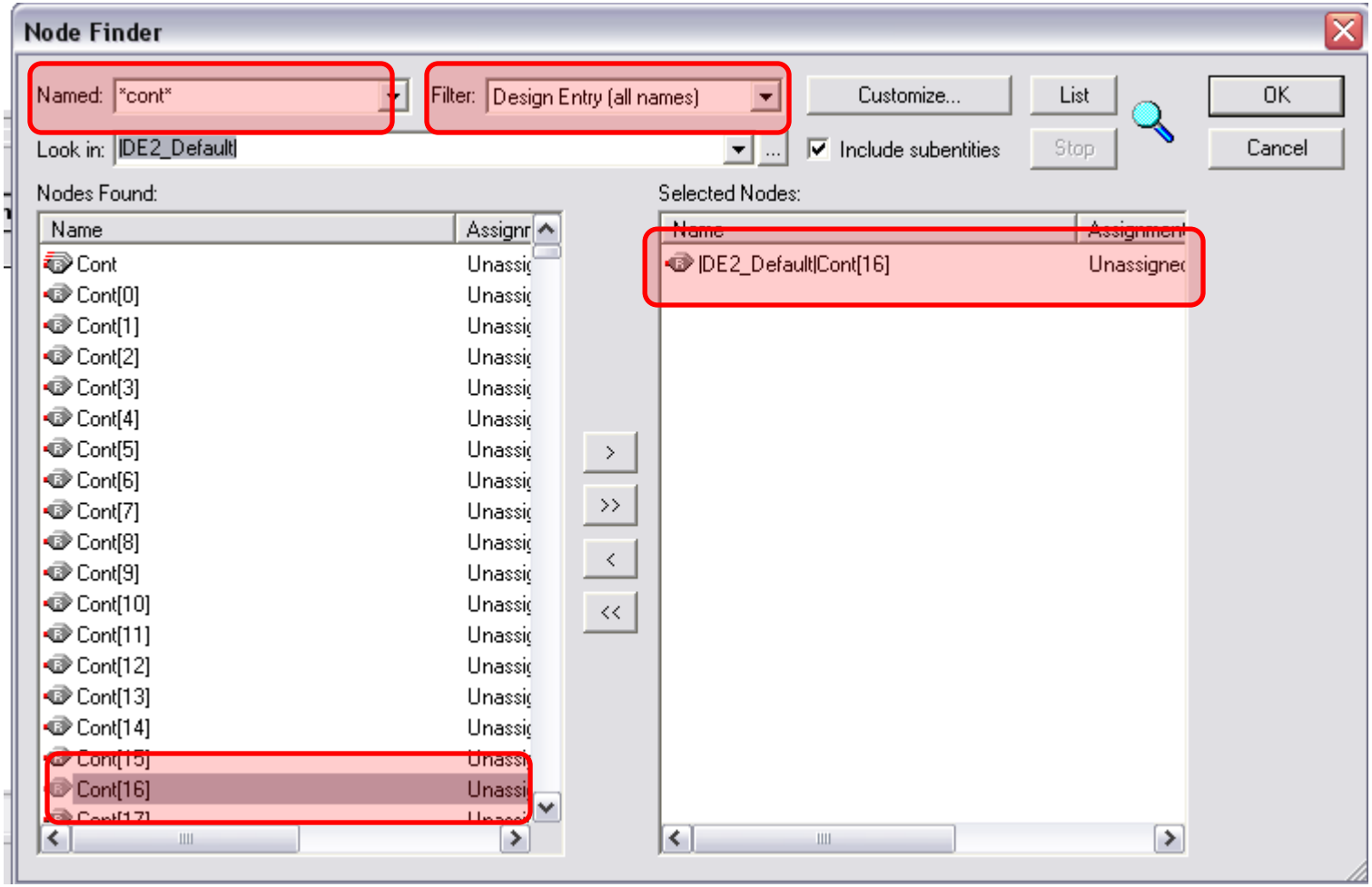
- Open DE2 default and compile it!
- Program DE2
- Tools→Signal TapII Logic Analyser



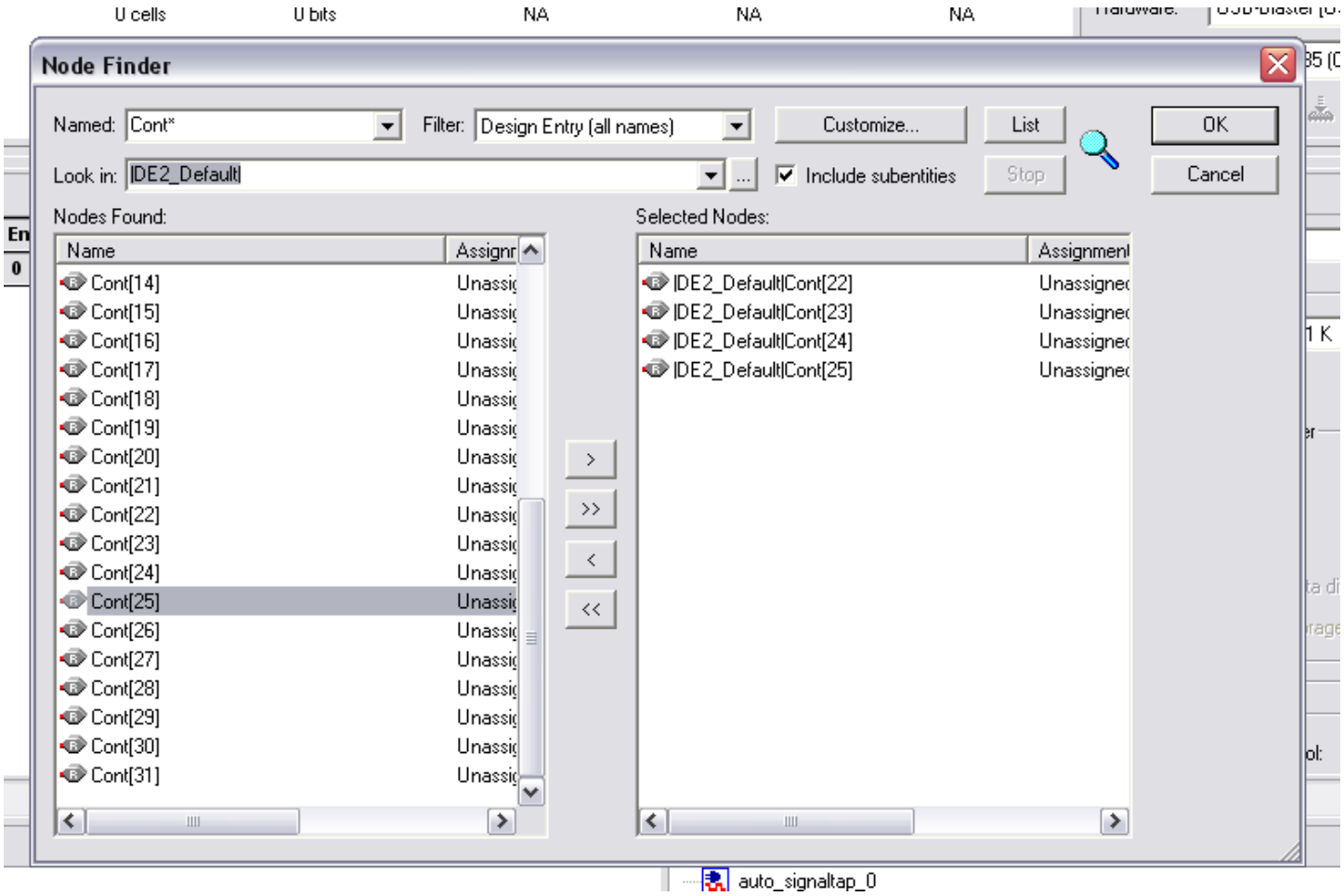
•Setup the hardware (choose the USB blaster)



•Define the clock

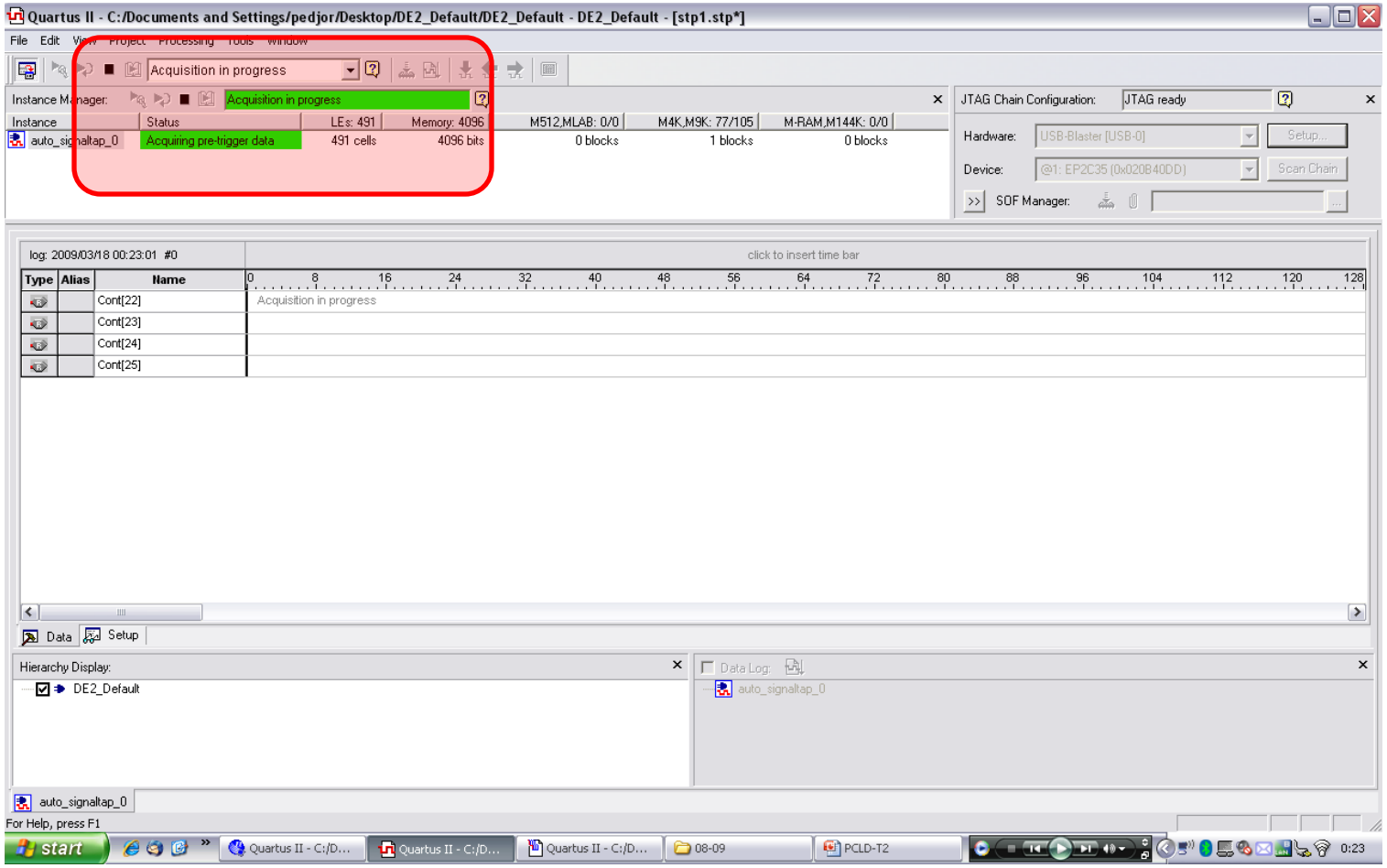

Node Finder
Named: *cont*
Filter: Design Entry (all names)
Customize...
List
OK
Look in: |DE2_Default|
Include subentities
Stop
Cancel
Nodes Found:
Name
Assignr
Cont
Cont[0]
Cont[1]
Cont[2]
Cont[3]
Cont[4]
Cont[5]
Cont[6]
Cont[7]
Cont[8]
Cont[9]
Cont[10]
Cont[11]
Cont[12]
Cont[13]
Cont[14]
Cont[15]
Cont[16]
>
>>
<
<<
Selected Nodes:
Name
Assignment
|DE2_Default|Cont[16]
Unassigned

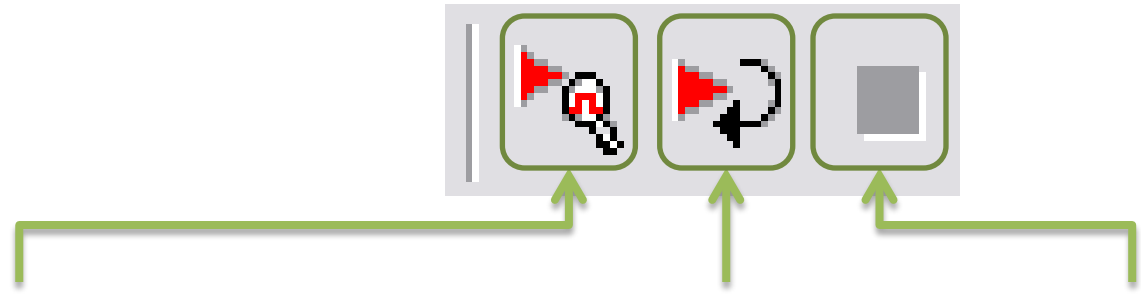
Choose the signals to observe. Choose Cont 25,24,23,22



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

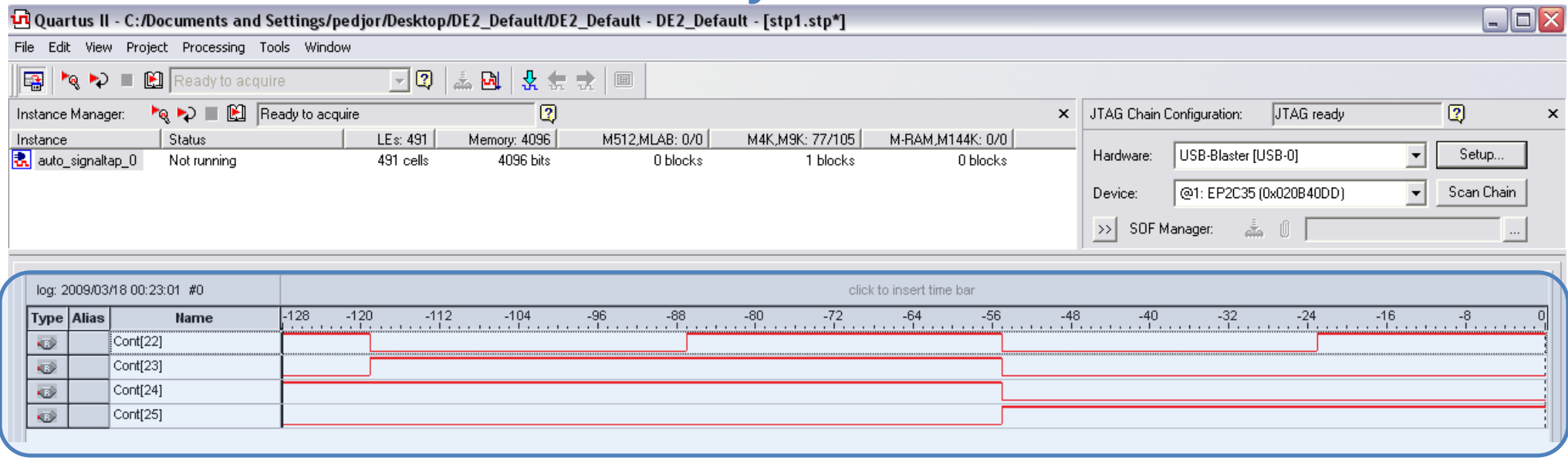
Run Analysis→Green (acquisition in progress, acquiring data



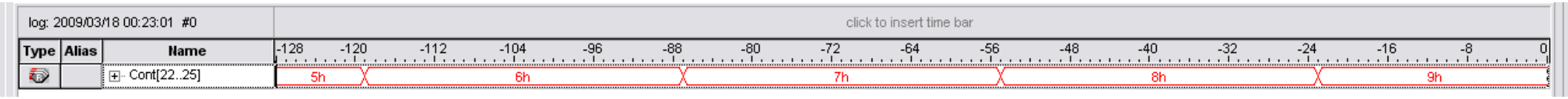


Run Analysis (1 time) Run Analysis (continuous) Stop

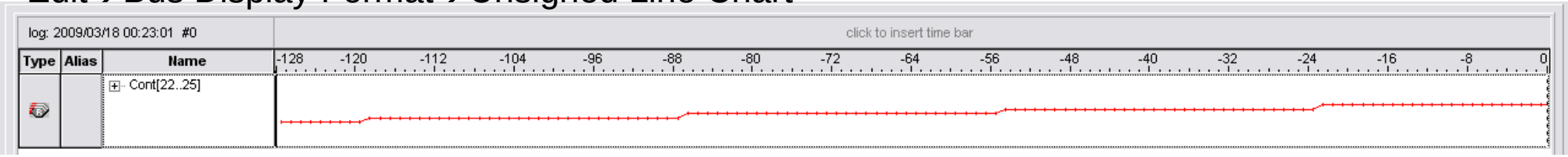
Finally: data!!



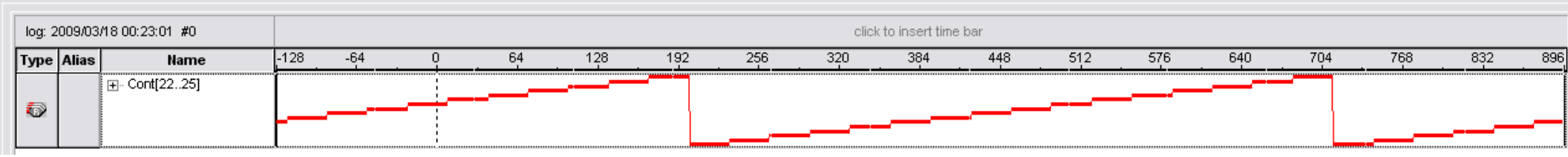
- Select the four signals;
- Edit→group



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

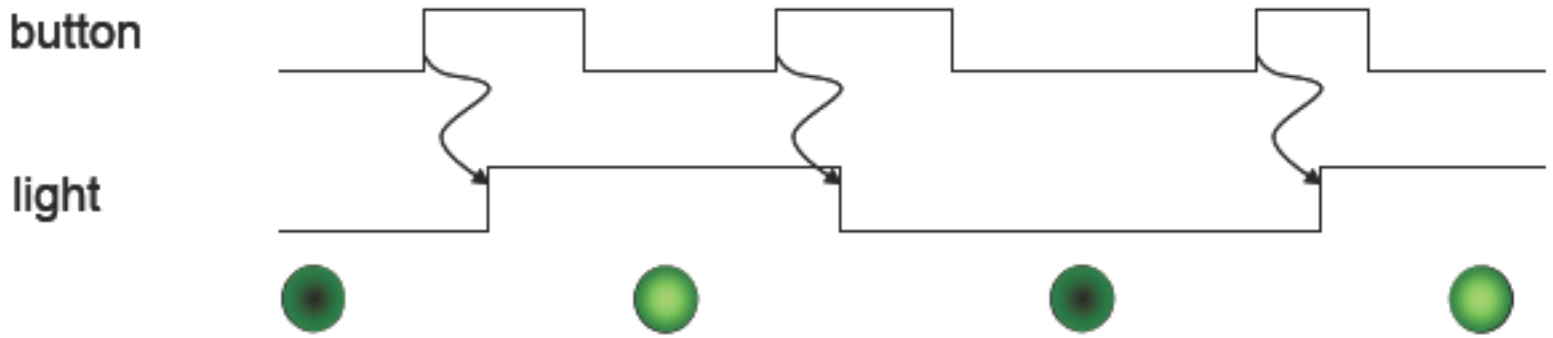


- Unzoom

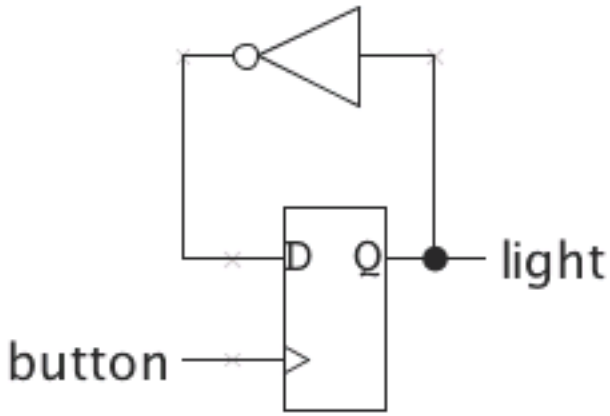



STOP

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.

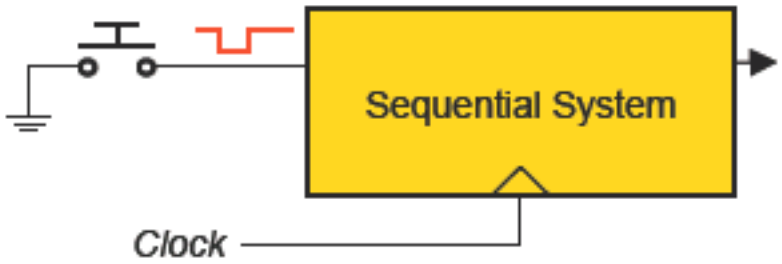
Clocks are fast, we're slow!

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 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
```

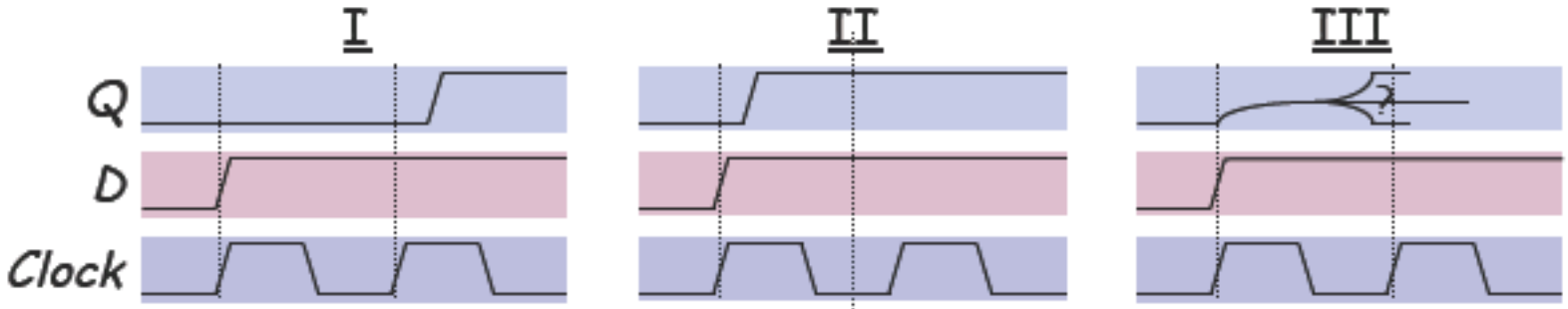
Asynchronous Inputs in Sequential Systems

What about external signals?



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

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



Transition is missed on first clock cycle, but caught on next clock cycle.

Transition is caught on first clock cycle.

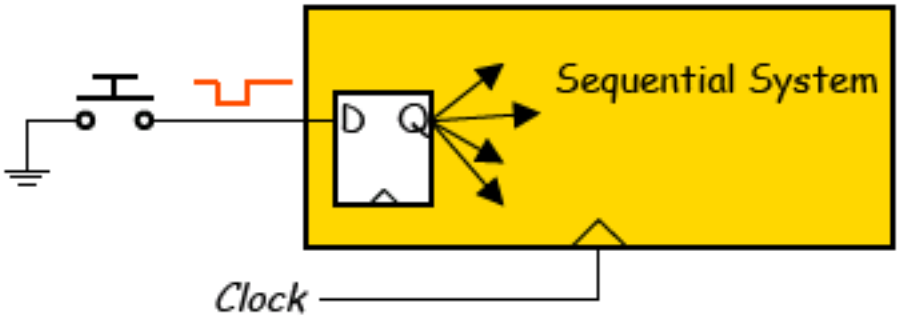
Output is metastable for an indeterminate amount of time.

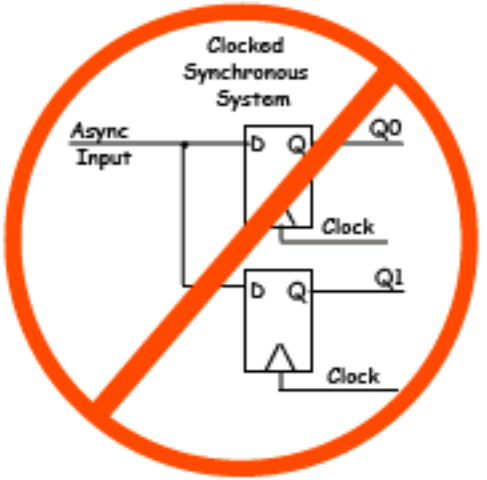
Q: Which cases are problematic?

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

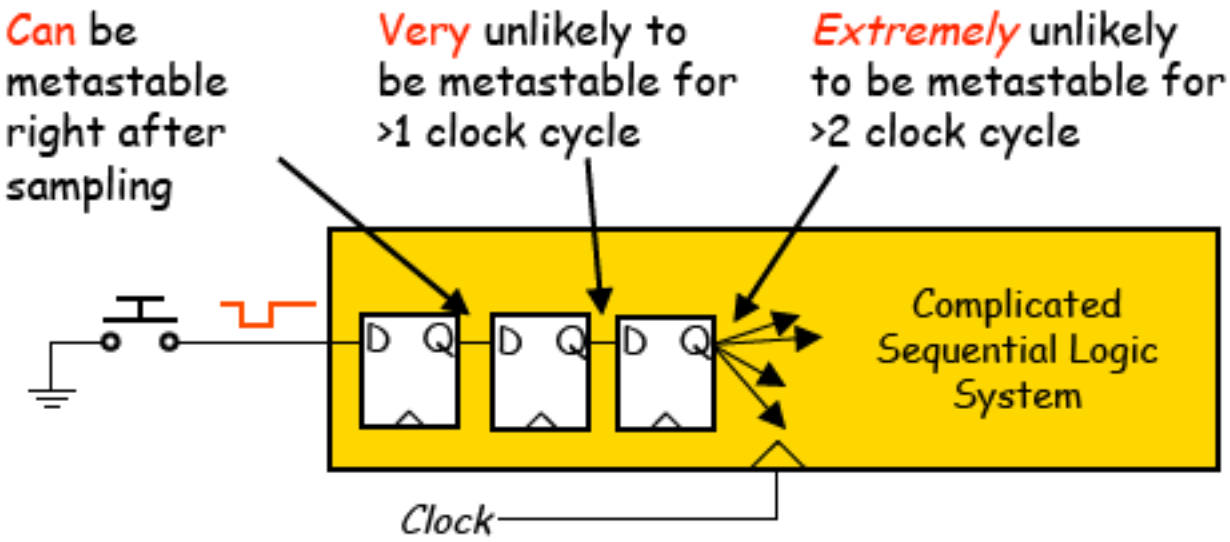




This prevents the possibility of I and II occurring in different places in the circuit, but what about metastability?

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

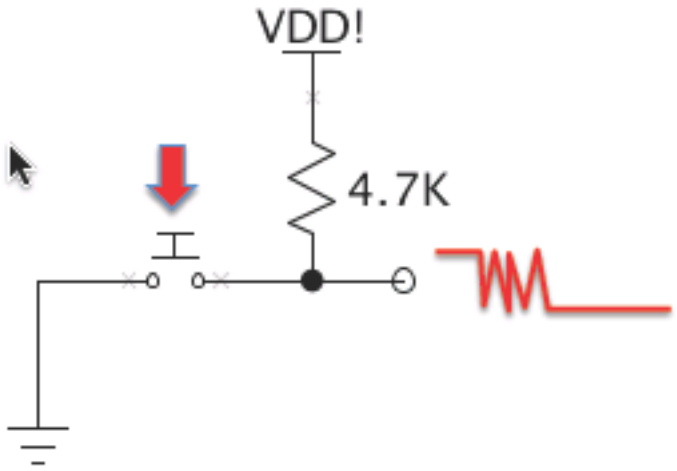


How many registers are necessary?

- 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!

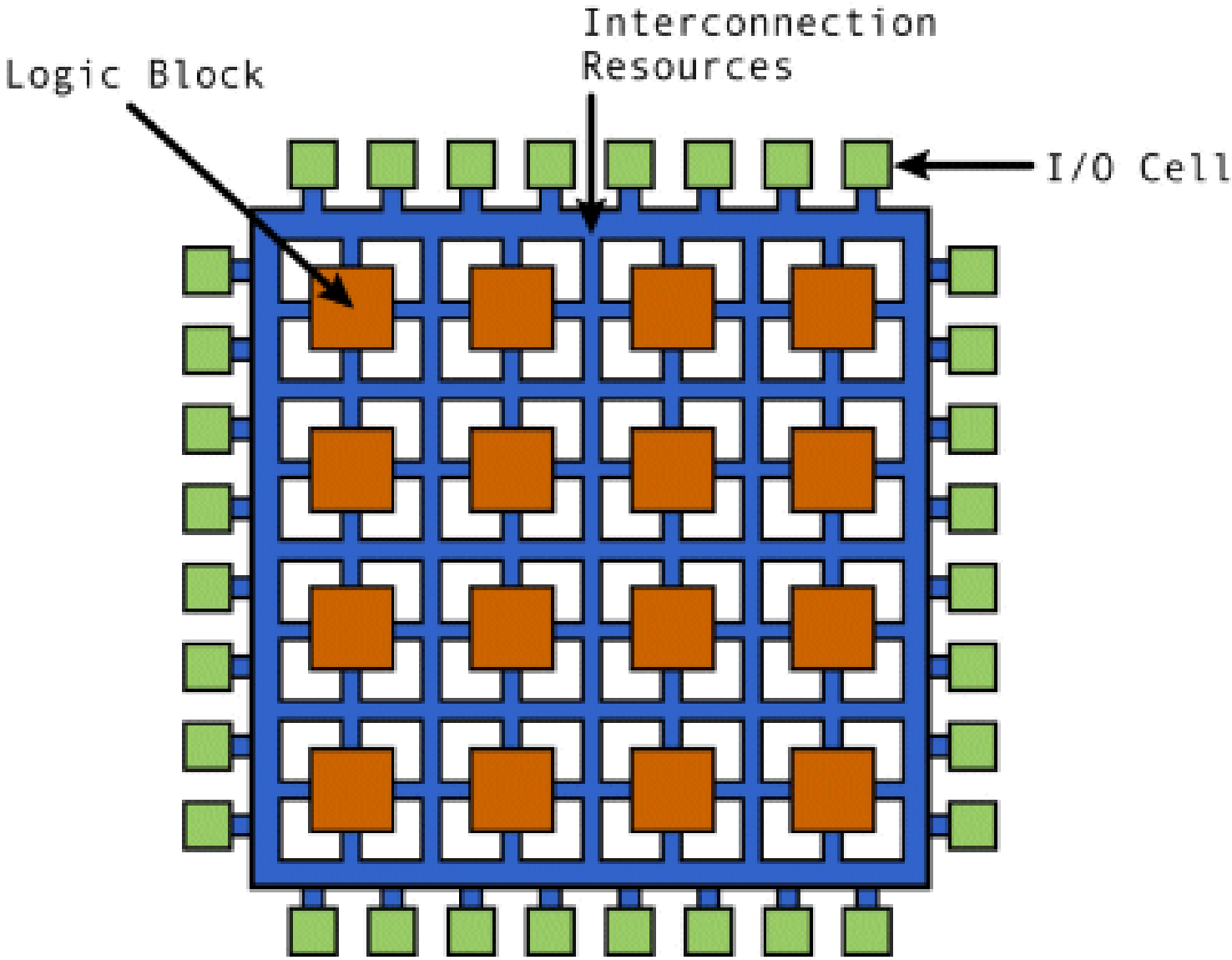
```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
// DELAY = .01 sec with a 27Mhz clock
module debounce #(parameter DELAY=270000)
                 (input reset, clock, noisy,
                  output reg clean);

   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
```

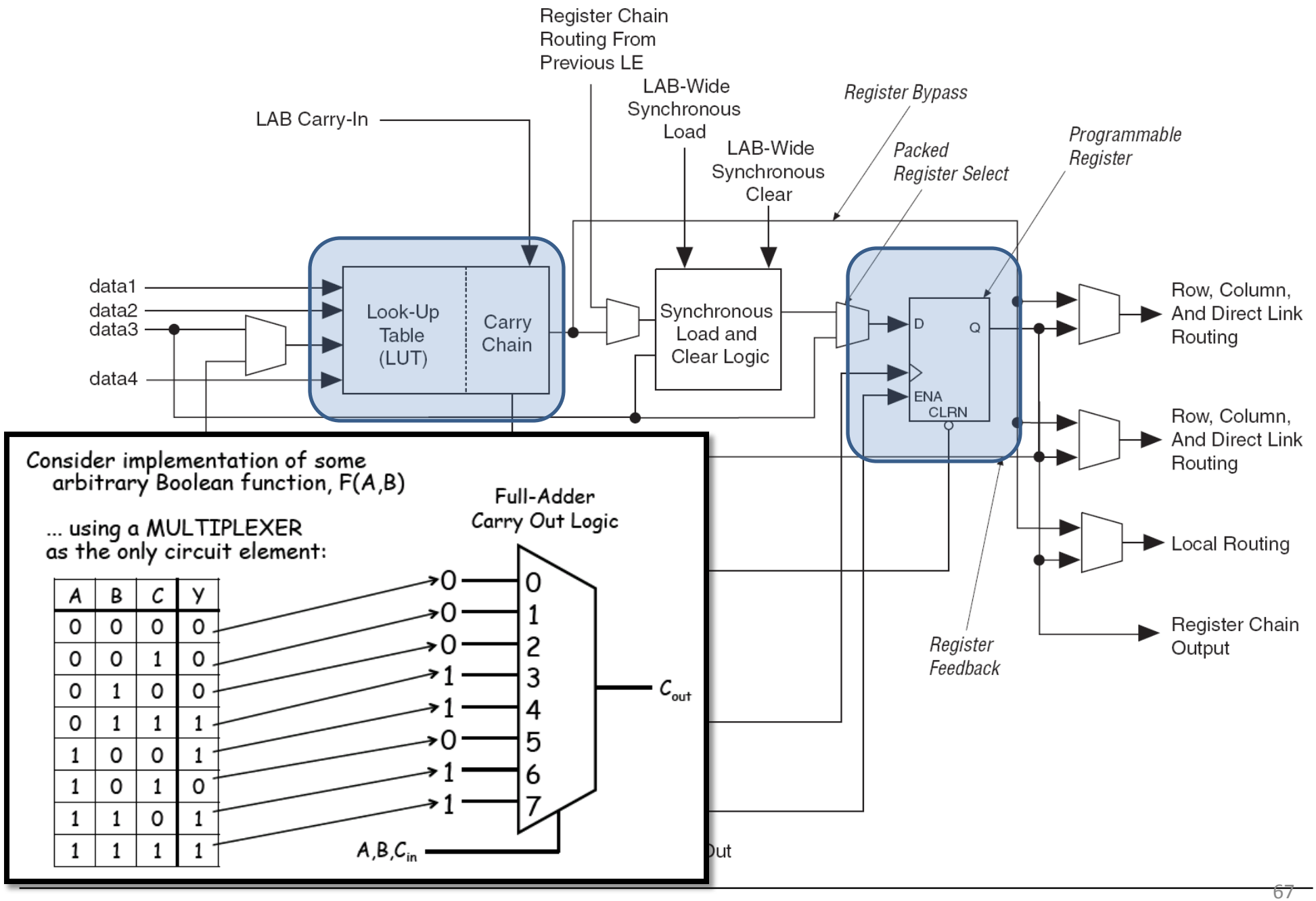
FPGA

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



Logic elements

Figure 2–2. Cyclone II LE



HDL

Hardware
Description
Language

A Tale of Two HDLs

VHDL	Verilog
ADA-like verbose syntax, lots of redundancy (which can be good!)	C-like concise syntax
Extensible types and simulation engine. Logic representations are not built in and have evolved with time (IEEE-1164).	Built-in types and logic representations. Oddly, this led to slightly incompatible simulators from different vendors.
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.	Design is composed of modules.
Behavioral, dataflow and structural modeling. Synthesizable subset...	Behavioral, dataflow and structural modeling. Synthesizable subset...
Harder to learn and use, not technology-specific, DoD mandate	Easy to learn and use, fast simulation, good for hardware design