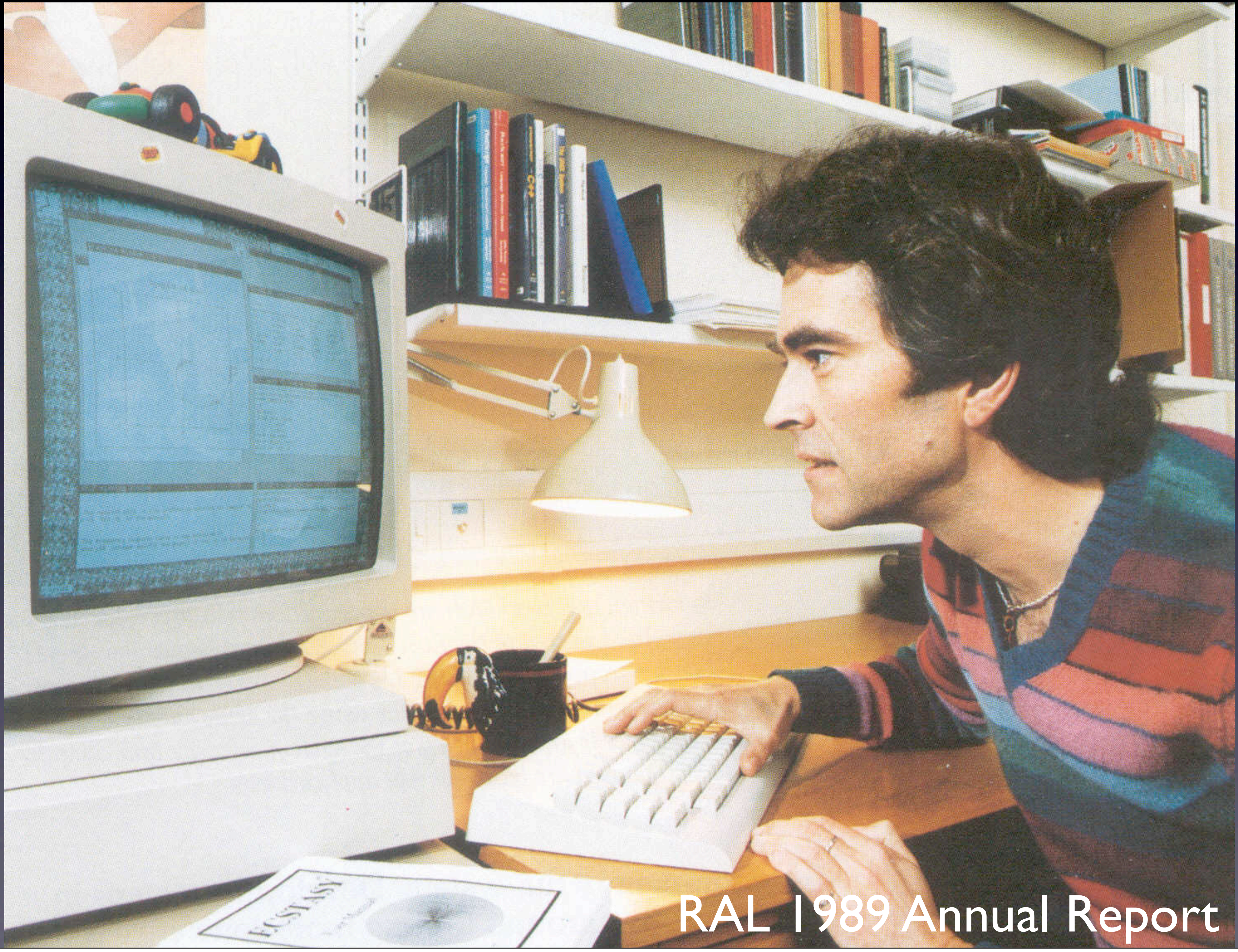
Avoiding pain in 378

Autumn ’09
CSE378
Jacob Nelson

Agenda

- Pep talk
- Lab overview
- Verilog
  - Mapping Verilog to hardware
  - Verilog tips and traps

My desktop circa 1998



RAL 1989 Annual Report

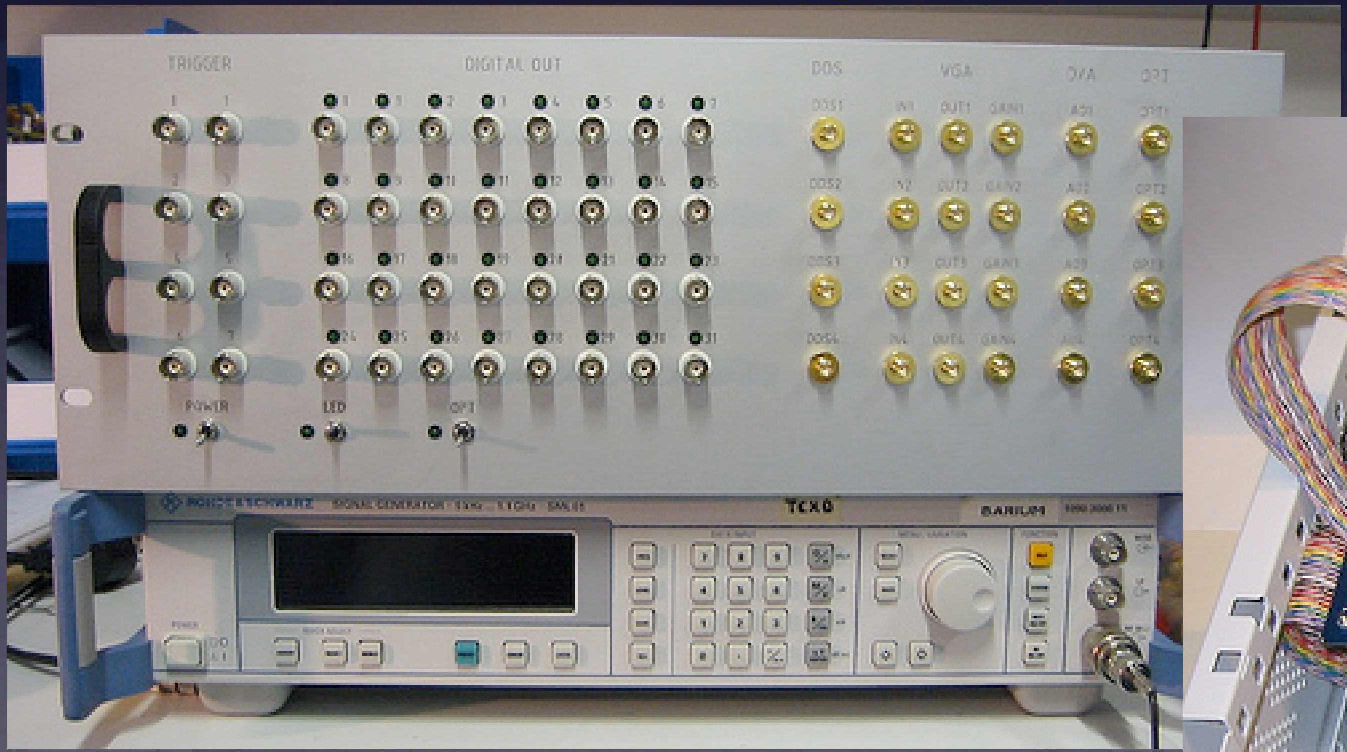
Your processor

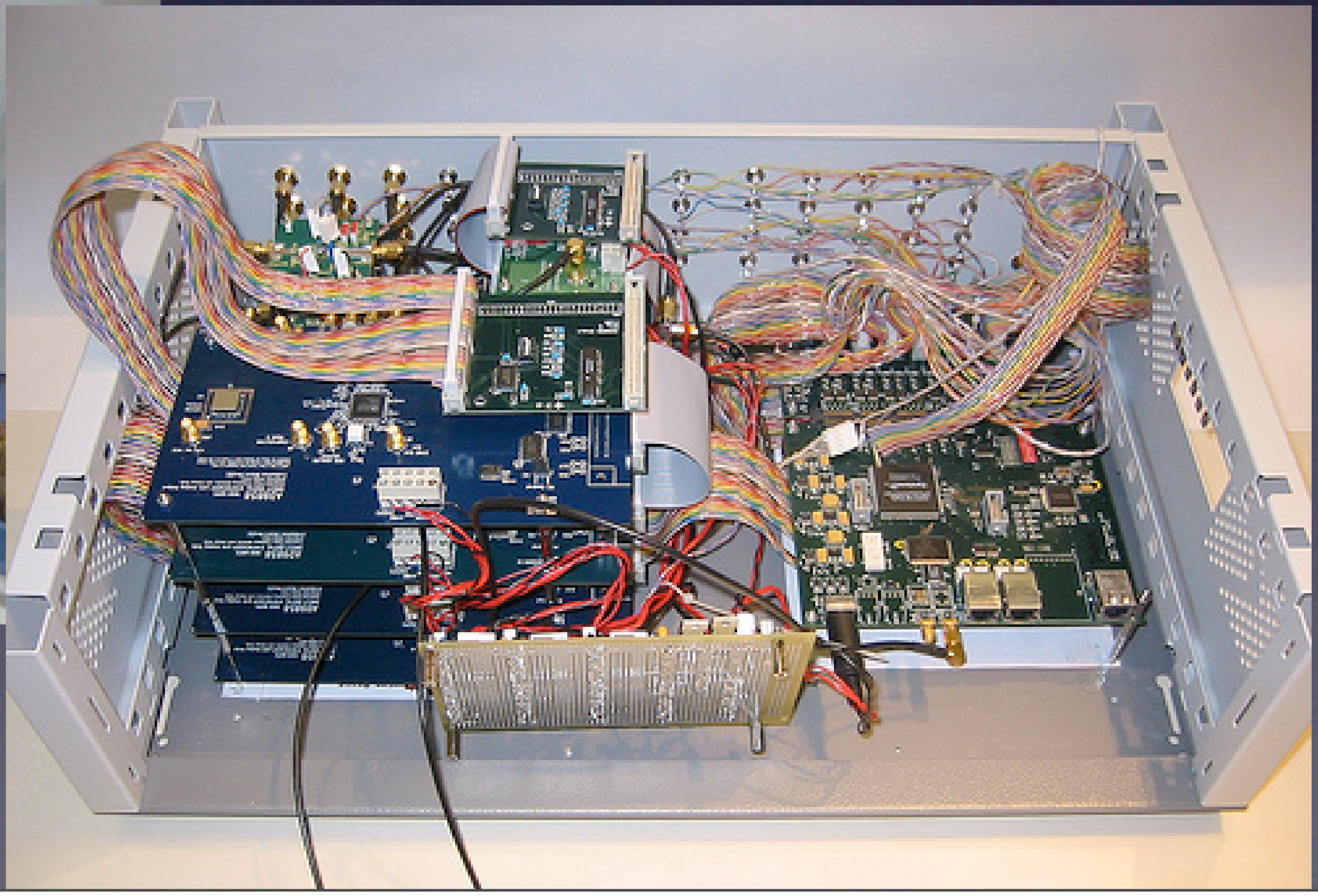
>>

Sun 3/50

What can we do?

http://www.xkl.com





http://pulse-sequencer.sourceforge.net

Free as in { freedom | beer }

simulation: Icarus Verilog; GTKWave

schematics/board design: gEDA

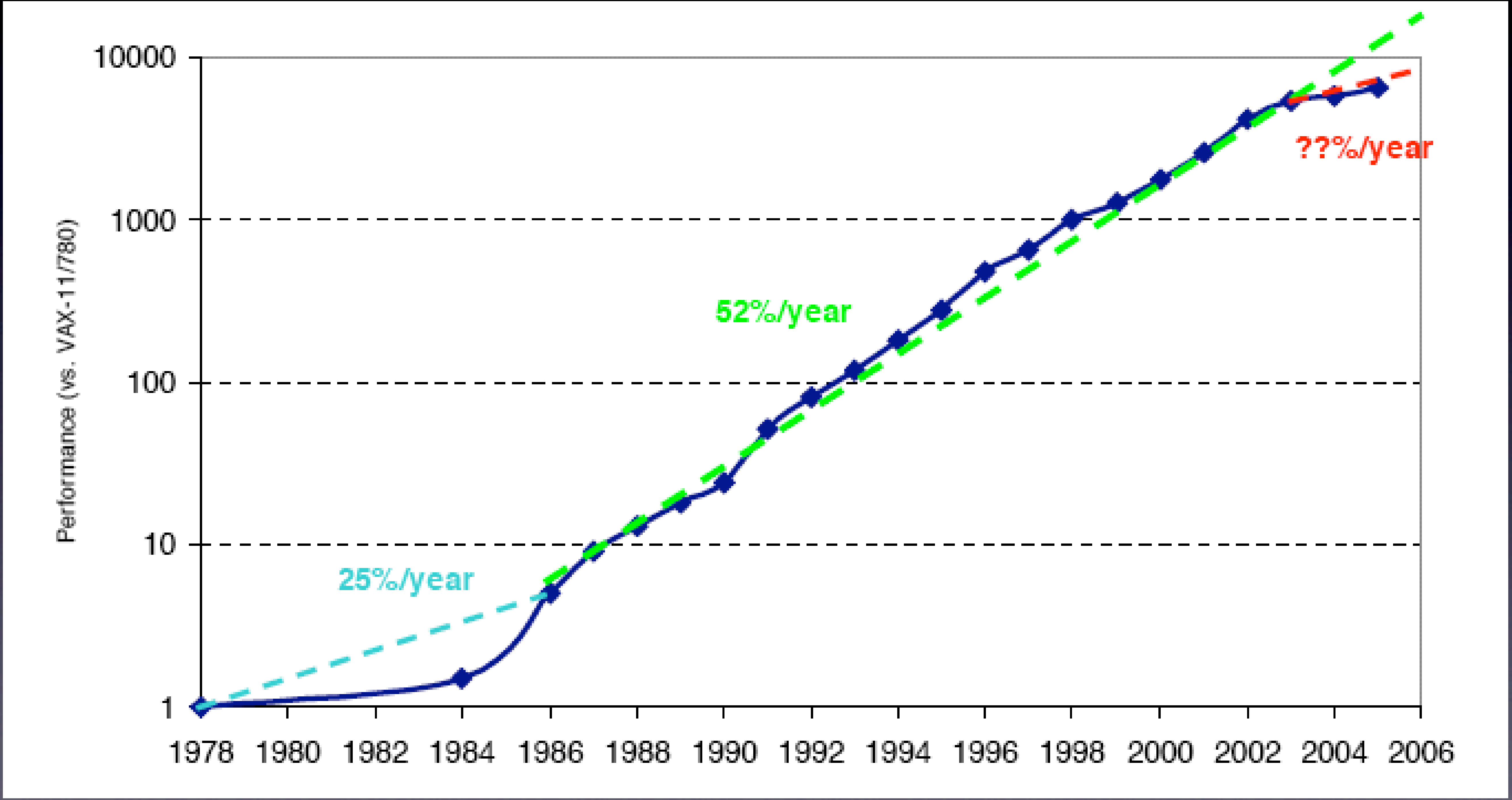
chips/implementation: Xilinx, Altera, etc.

boards: cheap PCB houses; toaster oven

programming/debugging: OpenOCD

Not doing it for you?

PARALLEL CRISIS!!!



The Landscape of Parallel Computing Research: A View From Berkeley

Funky parallelism

- Hardware is inherently parallel
- FPGA =
  Fine-grained massively parallel computer
- Verilog =
  Funky parallel programming language

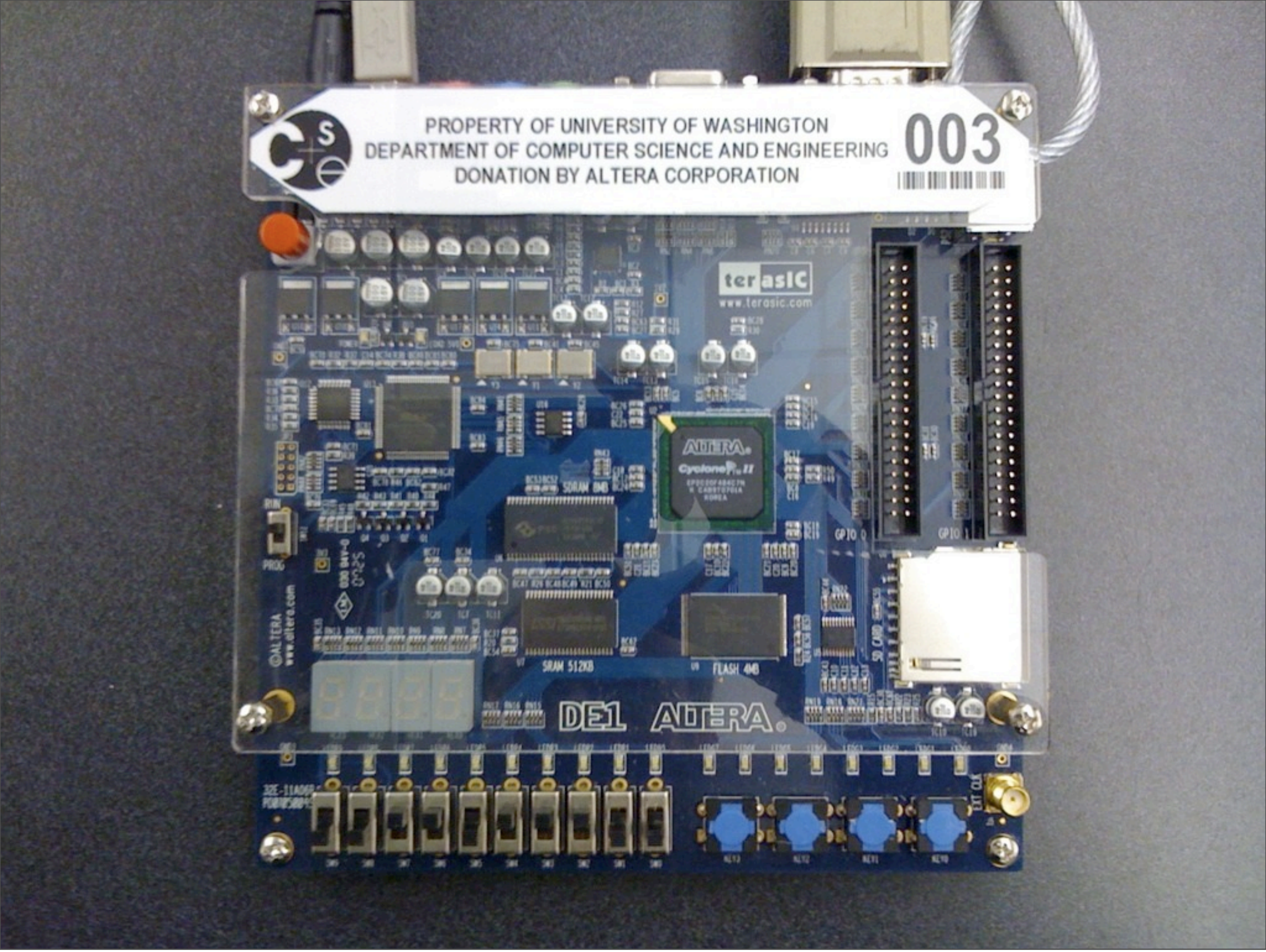
Lab Process and Goals

Four main lab tasks

- *One*: build single-cycle datapath; create jump/branch logic
- *Two*: build control logic for single-cycle CPU
- *Three*: add pipeline registers
- *Four*: finish pipline with forwarding and hazard detection

First task

- Mostly connecting things together in BDE; write Verilog for jumps and branches
- We provide testbenches, but they're incomplete (add tests?)
- Final step: write small program to flash lights


PROPERTY OF UNIVERSITY OF WASHINGTON
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
DONATION BY ALTERA CORPORATION
003
terasIC
www.terasic.com
ALTERA
Cyclone II
SDRAM 8MB
SRAM 512KB
FLASH 4MB
GPIO 0
GPIO 1
SD CARD
DE1
ALTERA

Our hardware

- Altera Cyclone II EP2C20
  - 18,752 4-input lookup tables
  - 18,752 one-bit registers
  - 240 kilobytes of memory on the chip
- (everything else on the board)

A few tools

- Aldec Active-HDL simulates Verilog and BDE
  - Assembler turns code into bits for memory
- Altera Quartus does three things:
  - translates Verilog to hardware primitives
  - arranges hardware primitives on the chip
  - downloads design to chip

Hardware primitives

Logic

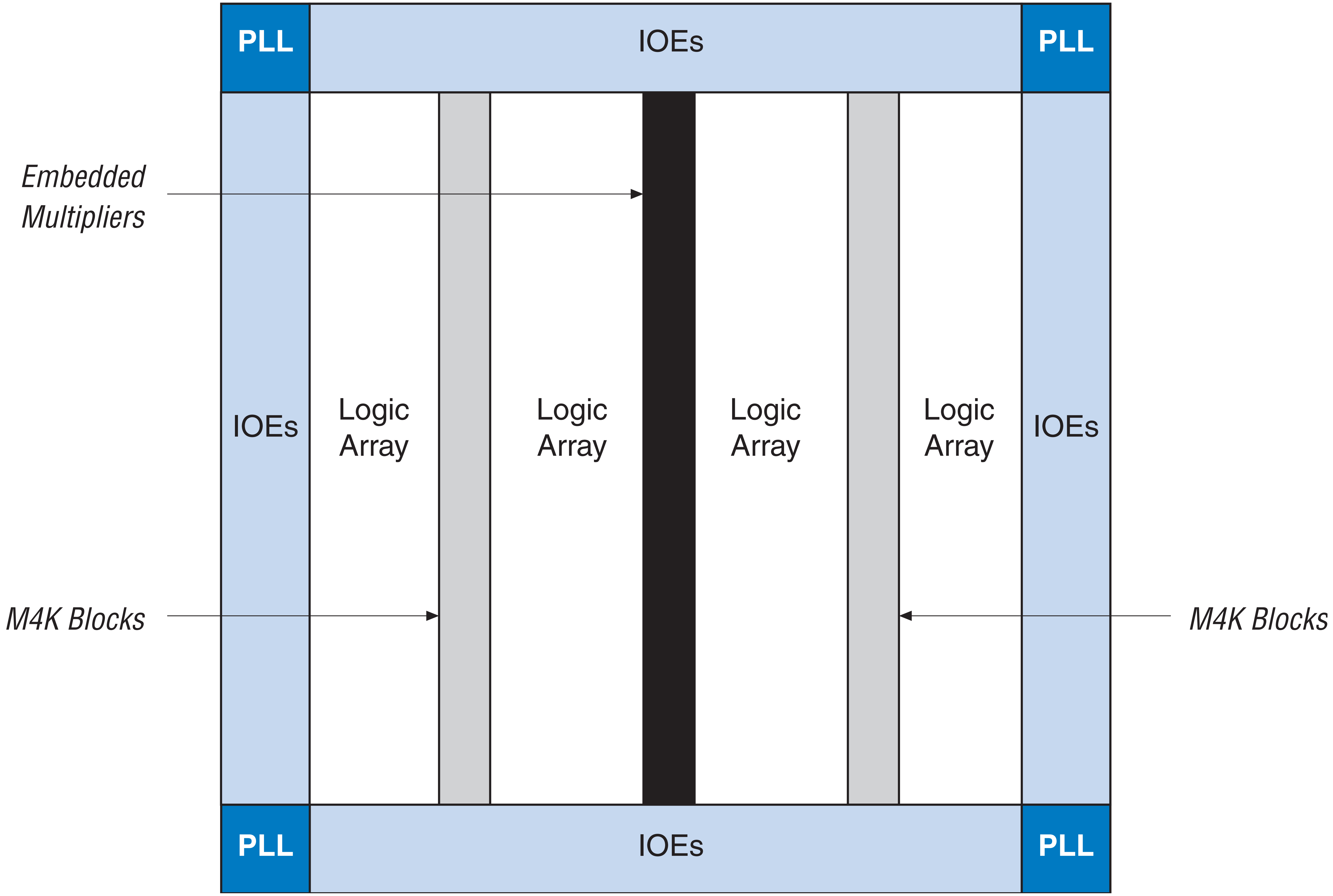
```
module foo (a,b,f,g);
    input wire a, b;
    output wire f;
    output reg g;

    assign f = a && b;
    always @(*)
        g = a && b;
endmodule
```

Registers

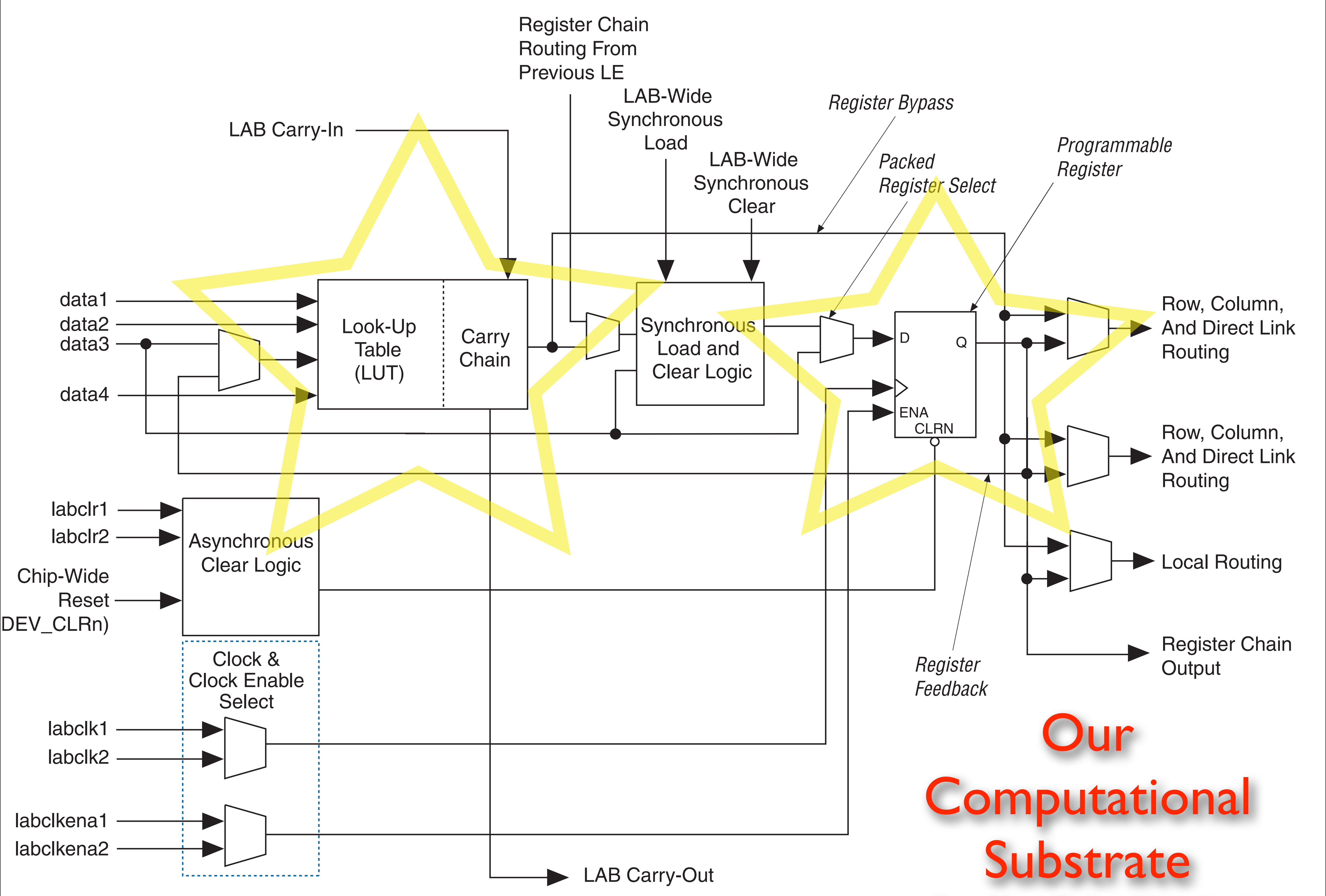
```
input wire nextFoo;
reg foo;

always @(posedge clk)
   foo <= nextFoo;
```

Our Computational Substrate

(from Altera Cyclone II datasheet)



igure 2–2. Cyclone II LE



Our Computational Substrate

(from Altera Cyclone II datasheet)

Muxes

- assign f =

```
  s[1] ?
    (s[0] ? a : b) :
    (s[0] ? c : d);

 always @(*)
  case (s)
   2'b00 : g = a;
   2'b01 : g = b;
   2'b10 : g = c;
   default: g = d; // 2'b11
  endcase
```

```
always @(*)
  if (s == 2'b00)
     h = a;
  else if (s == 2'b01)
     h = b;
  else if (s == 2'b10)
     h = c;
  else  // s == 2'b1
     h = d;
```

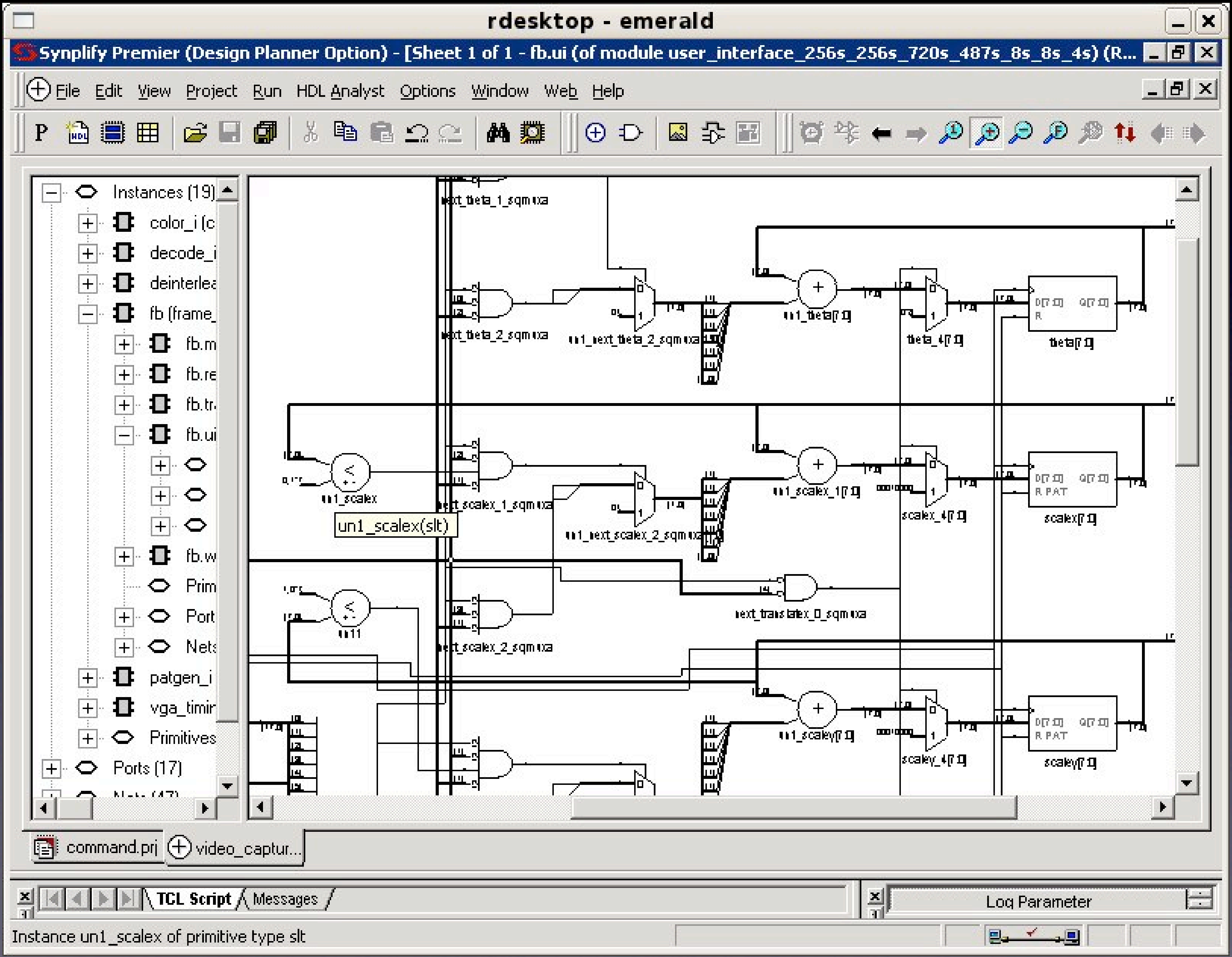
Adders/subtractors

- assign f = a + b;
  assign g = a - b;
- wire [8:0] s;
  wire [7:0] a, b;
  assign s = {0,a} + {0,b}; // pick up carry out
- How big are they? How fast?

Comparators

- assign isZero = (a == 0);
- assign isGreater = (a > b);   // unsigned!!!
  assign isLTZ = (a < 0);   // is this ever true?
- can do signed compares if ALL signals involved are declared "signed".
  wire signed [7:0] a, b;
- How big? How fast?

Check with RTL view



rdesktop - emerald

Synplify Premier (Design Planner Option) - [Sheet 1 of 1 - fb.ui (of module user_interface_256s_256s_720s_487s_8s_8s_4s) (R...

Instance un1_scalex of primitive type slt

Verilog tips and traps

HW Tools:
pain in digital form?

- We should teach ideas, not tools!
  But tools help express ideas
- HW tools often kind of suck, but
    don't blame tools for bad craftsmanship,
    do good craftsmanship with bad tools.
- Patience and care will win

= vs. <=

- Simple rule:
  - If you want sequential logic, use always @(posedge clk) with <=.
  - If you want combinational logic, use always @(*) with =.

Constants: 32 bits, decimal

- wire [7:0] foo = 127; // synthesis warning!
- wire [7:0] foo = 8'd127;
- wire [7:0] foo = 8'b11111111;
- wire [7:0] foo = 8'hff;
- wire [7:0] foo = 8'hFF;
- watch out: 1010 looks like 4'b1010!

Truncation

```
wire [7:0] a = 8'hAB;
wire b;                         // oops! forgot width
wire [7:0] c;

assign b = a;      // synthesis warning if lucky.

assign c = a;
```

reg vs. wire

-
  ```
  wire f;      reg g, h;

  assign f = a & b;

  always @(posedge clk)
    g <= a & b;

  always @(*)
    h = a & b;
  ```

Assign in one block

```
input wire a, b;
output reg f;

always @(posedge clk)
   if (a) f <= 1'b0; // race!

always @(posedge clk)
   if (b) f <= 1'b1; // race!
```

= vs. <=

- Simple rule:
  - If you want sequential logic, use always @(posedge clk) with <=.
  - If you want combinational logic, use always @(*) with =.

=   vs.   <=

-
  ```
  always @(posedge clk)
    begin
      f <= a + b;
      g <= f + c;
    end
  ```

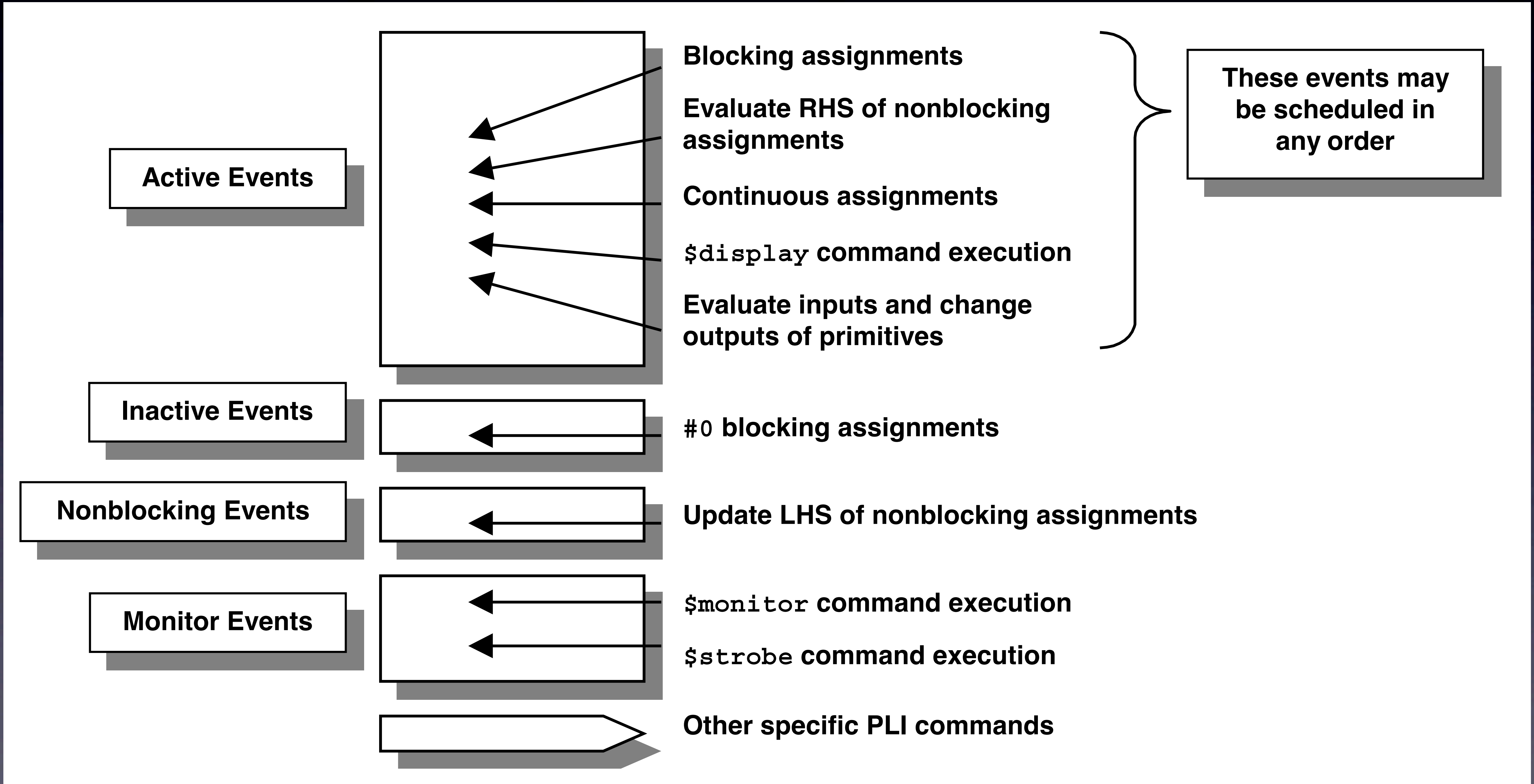
-
  ```
  always @(posedge clk)
    begin
      f = a + b;
      g = f + c;  // a + b + c
    end
  ```

-
  ```
  always@(posedge clk)
    begin
      f2 <= f1;
      f3 <= f2;

      f4 = f3;
      f5 = f4;   // f5 = f3 !!

      f7 = f6;
      f6 = f5;
    end
  ```

More specifically,

```
initial
  state = 0;

always @(posedge clk)
  begin
    if (state == 0) state = 1;
    if (state == 1) state = 2;
    if (state == 2) state = 0;
  end
```

Aargh.



Active Events
Inactive Events
Nonblocking Events
Monitor Events
Blocking assignments
Evaluate RHS of nonblocking assignments
Continuous assignments
$display command execution
Evaluate inputs and change outputs of primitives
These events may be scheduled in any order
#0 blocking assignments
Update LHS of nonblocking assignments
$monitor command execution
$strobe command execution
Other specific PLI commands

Figure 1 - Verilog "stratified event queue"

from Cliff Cummings' "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!"

Incomplete sensitivity lists

- always @(a or b)   // it's or, not ||

  ```
  f = a & b;
  ```
- always @(a)

  ```
  f = a & b;
  ```
- always

  ```
  f = a & b;
  ```
- Just use always@(*) for combinational logic

Enables and Latches

- always @(posedge clk)
```
  if (a == 1)
    f <= 1;
  else if (a == 2)
    f <= 2;
  else if (a == 3)
    f <= 3;
```

- implicitly:
```
  else
    f <= f;
```

- always @(*)
```
  if (a == 1)
    f = 1;
  else if (a == 2)
    f = 2;
  else if (a == 3)
    f = 3;
```

- implicitly:
```
  else
    f = f;
```
  this is memory!

=   vs.   <=

- Simple rule:
  - If you want sequential logic, use always @(posedge clk) with <=.
  - If you want combinational logic, use always @(*) with =.

Combinational and Sequential

```
input wire a, b, s;
output reg f, g, h;

always @(posedge clk)
  begin

    f <= (a & ~s) | (b & s);                if (s)
                                              h <= a;
    g <= s ? a : b;                         else
                                              h <= b;

                                          end
```

Displaying things

- works for most stuff:
  $display(“the answer is %h.”, ans);
- for nonblocking assignments, you may sometimes want:
  $strobe(“the answer is %h.”, ans);
  (see Aargh. for reason)

X’s

- X’s are for undefined values:
  ```
  wire a;
  $display(a);   // prints an X
  ```
- Pins that aren’t hooked up will be X’s: Often, 32’hxxxxxxf4 indicates an Active-HDL bus with default width.
- 1’b1 & 1’bX yields 1’bX
  1’b1 + 1’bX yields 1’bX

Z’s

- Z’s are for bus sharing. You won’t need this.
- a <= 1’bZ;   b <= 1’bZ;
  a <= 2’b0;   b <= 1’b1;
  // a will be 0 and b will be 1
- Z’s turn into X’s sometimes:
  1’b1 & 1’bZ yields 1’bX.
  1’b1 + 1’bZ yields 1’bX.

Initial values

- Synthesis sometimes ignores (!?!), so better include a reset line.
- Maybe:
  ```
  reg foo = 1'b1;
  ```
- Maybe:
  ```
  initial begin
    foo = 1'b1;
  end
  ```

-
  ```
  module fooReg;
    input wire newFoo;
    output reg foo;

    initial #0 foo = 1'b0;

    always @(posedge clk)
      if (reset == 1'b1)
        foo <= 1'b0;
    else
        foo <= newFoo;
  endmodule
  ```

Whew.

Questions?