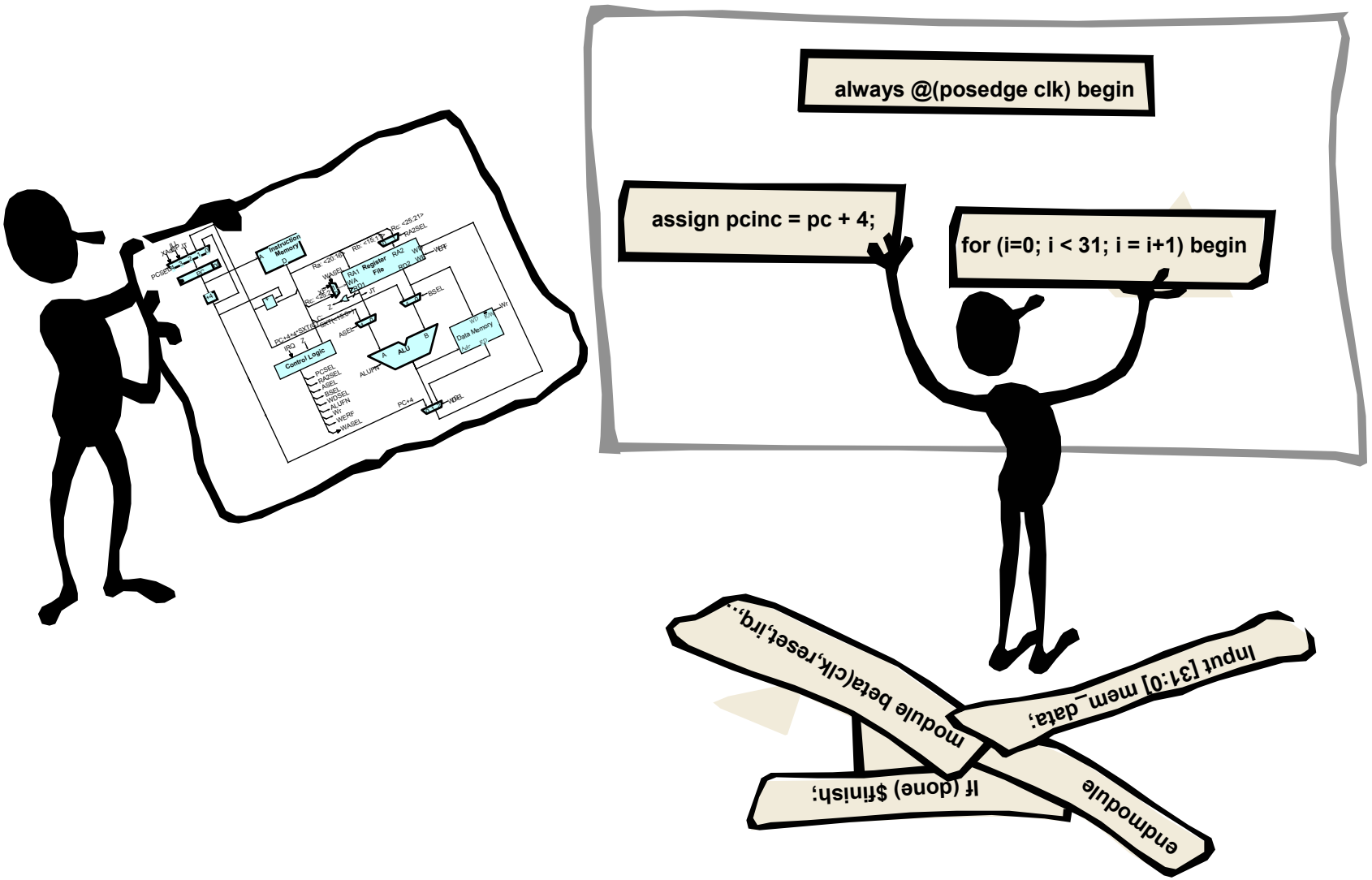
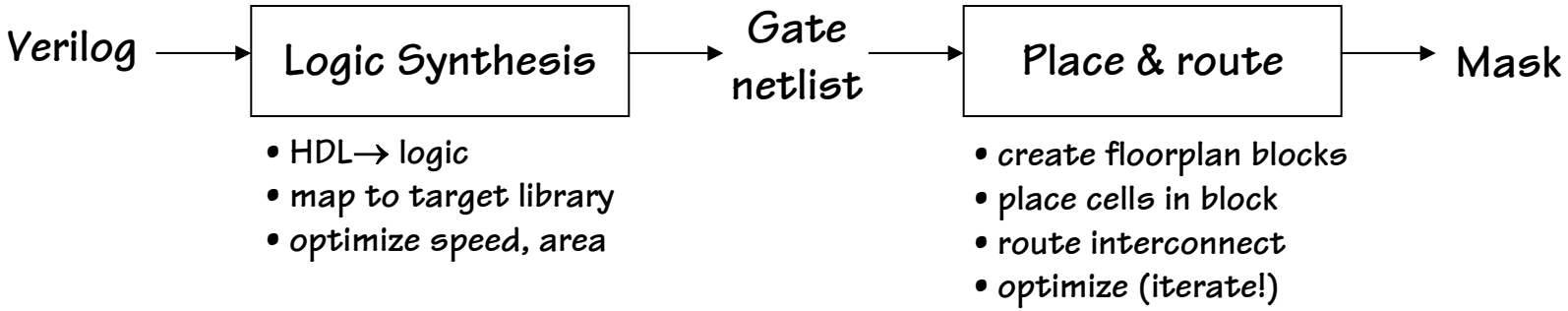
Architecture → Verilog
An Extended Example



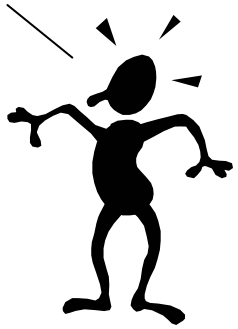
Reprise: Why use an HDL?

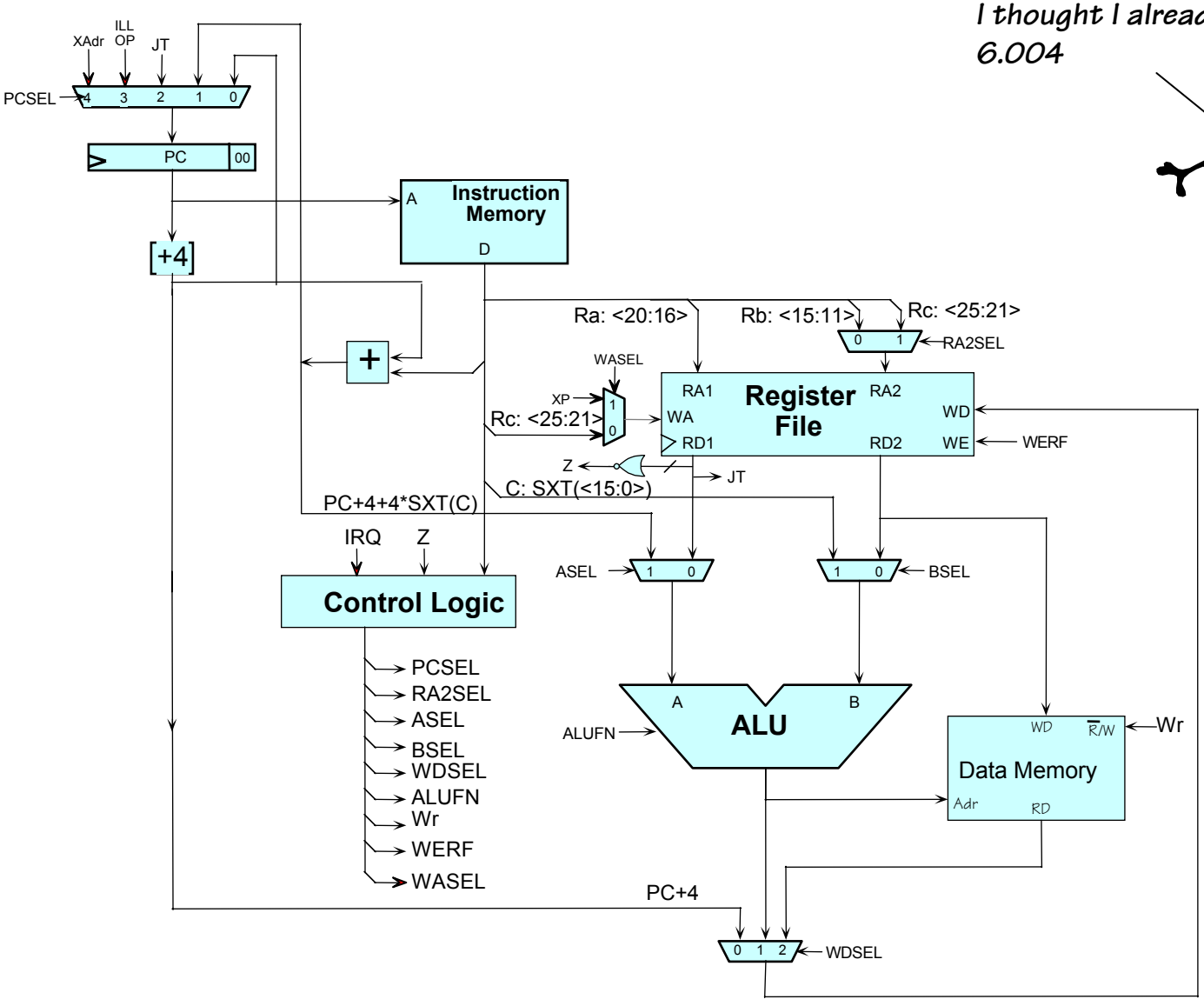
- Want an executable functional specification
  - Document exact behavior of all the modules and their interfaces
  - Executable models can be tested & refined until they do what you want
- Too much detail at the transistor and mask levels
  - Can't debug 1M transistors as individual analog components
  - Abstract away "unnecessary" details
  - Play by the rules: don't break abstraction with clever hacks
- HDL description is first step in a mostly automated process to build an implementation directly from the behavioral model



Beta redux!

I thought I already did 6.004





Goals for the Verilog description

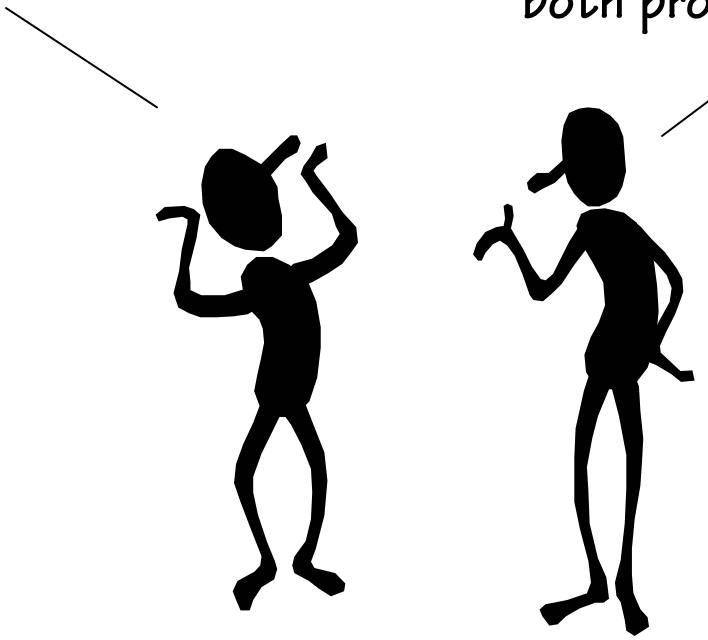
- Readable, correct code that clearly captures the architecture diagram – "correct by inspection"
- Partition the design into regions appropriate for different implementation strategies. Big issue: wires are "bad" since they take up area and have capacitance (impacting speed and power).
  - Memories: very dense layouts, structured wires pretty much route themselves, just a few base cells to design & verify
  - Datapaths: each cell contains necessary wiring, so replicating cells (for N bits of datapath) also replicates wiring. Data flows between columnar functional units on horizontal busses and control flows vertically.
  - Random logic: interconnect is "random" but library of cells can be designed ahead of time and characterized.
  - Think about physical partition: wires that cross boundaries can take lots of area; blocks have to fit into the floorplan without wasteful gaps.

Themes: draw as few fets as possible; maximize use of design techniques that offer good wire management strategies; use special tools for each type of layout

Hey! What happened to abstraction?

Wasn't the plan to abstract-away the physical details so we could concentrate on getting the functionality right? Why are we worrying about wires and floorplans at this stage?

Because life is short! If you have the luxury of writing two models (the first to experiment with function, the second to describe the actual partition you want to have), by all means! But with a little experience you can tackle both problems at once.



Helping the Tools*

In an ideal world it shouldn't matter how you write the Verilog – optimization in the CAD tools will find the best solution. But the world is not ideal (yet…)

- Tools work best on smaller problems
  - Need to partition real problem into pieces for you and the tools (it's hard to think about 1M gates at one time). Decompose large problems into smaller problems and then connect the solutions
  - Hierarchy in Verilog ↔ partitions in physical layout

- Tools use your code as a starting point.
  - Your structure isn't completed eliminated (this is good…)
  - Little optimization will be done between top-level blocks

- Structure of the problem is often important
  - Finding a "good" way to think about the problem is key

Like optimizing compilers for C, tools are good for local optimizations but don't expect them to rewrite your code and change your algorithm. With practice you'll learn what works and what doesn't…

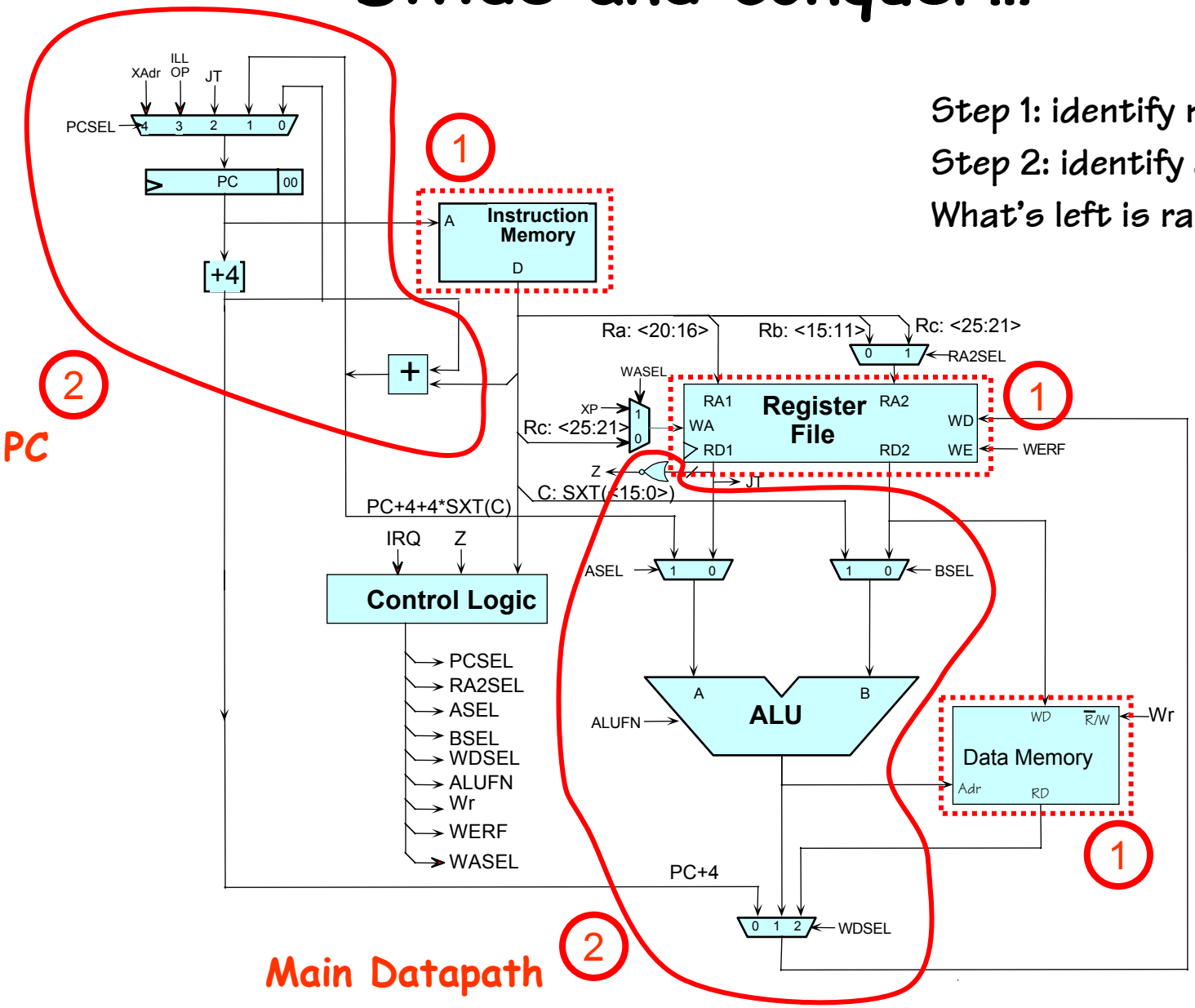
*adapted from a Stanford EE271 lecture by Mark Horowitz

Divide and conquer…

Step 1: identify memories

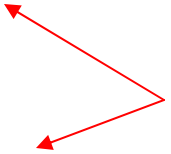
Step 2: identify datapaths

What's left is random logic…



Choosing the right style…

- Structural Verilog
  - Use for hierarchy (instantiating other modules)
  - Floorplanning tools often require that modules which include structural verilog not include other styles. In other words the leafs of the hierarchy are dataflow/behavioral modules, all other modules are pure structural verilog.
- Dataflow Verilog: `assign` ***`target = expression`***
  - Use for (most) combinational logic
  - Avoids problems with activation list omissions
- Behavioral Verilog: `always @(…) begin … end`
  - Use to model state elements (e.g., registers)
  - Sometimes useful for combinational logic expressed using `for` or `case` statements
  - Simulates much faster than dataflow statements since no waveforms are produced for signals internal to behavioral block. Here's where you can make the tradeoff between simulation speed and debugability.



These two styles are often mixed in a single module

Beta Module Hierarchy

- beta
  - control [random logic]
  - regfile [memory]
  - pc [datapath]
  - datapath
    - dp_misc [datapath]
    - dp_alu
      - dp_addsub [datapath]
      - dp_boole [datapath]
      - dp_shift [datapath]
      - dp_cmp [datapath]
      - dp_mux [datapath]
    - dp_wdata [datapath]

Beta verilog can be found in /mit/6.371/examples/beta.vl or Handouts webpage

Register File

```
// 2-read, 1-write 32-location register file
module regfile(ra1,rd1,ra2,rd2,clk,werf,wa,wd);
  input [4:0] ra1;          // address for read port 1 (Reg[RA])
  output [31:0] rd1;        // read data for port 1
  input [4:0] ra2;          // address for read port 2 (Reg[RB], Reg[RC] for ST)
  output [31:0] rd2;        // read data for port 2
  input clk;
  input werf;               // write enable, active high
  input [4:0] wa;           // address for write port (Reg[RC])
  input [31:0] wd;          // write data

  reg [31:0] registers[31:0];   // the register file itself (local)

  // read paths are combinational
  // logic to ensure R31 reads as zero is in main datapath
  assign rd1 = registers[ra1];
  assign rd2 = registers[ra2];

  // write port is active only when WERF is asserted
  always @(posedge clk)
    if (werf) registers[wa] <= wd;
endmodule
```

= vs. <= inside begin … end

```
module main;
  reg a,b,clk;

initial begin
    clk = 0; a = 0; b = 1;
    #10 clk = 1;
    #10 $display("a=%d b=%d\n",a,b);
    $finish;
  end
endmodule
```

A

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

B

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

C

```
always @(posedge clk) a = b;
always @(posedge clk) b = a;
```

D

```
always @(posedge clk) a <= b;
always @(posedge clk) b <= a;
```

E

```
always @(posedge clk) begin
  a <= b;
  b = a;    // urk! Be consistent!
end
```

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

PC

```
module pc(clk,reset,pcsel,offset,jump_addr,
          branch_addr,pc,pc_plus_4);
  input clk;
  input reset;                  // forces PC to 0x80000000
  input [2:0] pcsel;            // selects source of next PC
  input [15:0] offset;          // inst[15:0]
  input [31:0] jump_addr;       // from Reg[RA], used in JMP instruction
  output [31:0] branch_addr;    // send to datapath for LDR instruction
  output [31:0] pc;             // used as address for instruction fetch
  output [31:0] pc_plus_4;      // saved in regfile during branches, JMP, traps

  reg [31:0] pc;
  wire [30:0] pcinc;
  wire [31:0] npc;

  // the Beta PC increments by 4, but won't change supervisor bit
  assign pcinc = pc + 4;
  assign pc_plus_4 = {pc[31],pcinc};

  // branch address = PC + 4 + 4*sxt(offset)
  assign branch_addr = {0,pcinc + {{13{offset[15]}},offset[15:0],2'b00}};

  assign npc = reset ? 32'h80000000 :
               (pcsel == 0) ? {pc[31],pcinc} :                  // normal
               (pcsel == 1) ? {pc[31],branch_addr[30:0]} :      // branch
               (pcsel == 2) ? {pc[31] & jump_addr[31],jump_addr[30:0]} :  // jump
               (pcsel == 3) ? 32'h80000004 : 32'h80000008;      // illop, trap

  // pc register, pc[31] is supervisor bit and gets special treatment
  always @(posedge clk) pc <= npc;
endmodule
```

Main Datapath

```
module datapath(inst,rd1,rd2,pc_plus_4,branch_addr,mem_data,
                rd1zero,rd2zero,asel,bsel,wdsel,alufn,
                wdata,mem_addr,mem_wdata,z);
  input [15:0] inst;         // constant field from instruction
  input [31:0] rd1;          // Reg[RA] from register file
  input [31:0] rd2;          // Reg[RB] from register file (Reg[RC] for ST)
  input [31:0] pc_plus_4;    // incremented PC
  input [31:0] branch_addr;  // PC + 4 + 4*sxt(inst[15:0])
  input [31:0] mem_data;     // memory read data (for LD)
  input rd1zero;             // RA == R31, so treat RD1 as 0
  input rd2zero;             // RB/RC == R31, so treat RD2 as 0
  input asel;                // select A operand for ALU
  input bsel;                // select B operand for ALU
  input [1:0] wdsel;         // select regfile write data
  input [5:0] alufn;         // operation to be performed by alu
  output [31:0] wdata;       // regfile write data (output of WDSEL mux)
  output [31:0] mem_addr;    // alu output, doubles as data memory address
  output [31:0] mem_wdata;   // data memory write data
  output z;                  // true if Reg[RA] is zero, used during branches

  wire [31:0] alu_a,alu_b;   // A and B inputs to ALU

  // compute A and B inputs into alu, also Z bit for control logic
  dp_misc misc(rd1zero,rd2zero,asel,bsel,inst,rd1,rd2,branch_addr,
               alu_a,alu_b,mem_wdata,z);
  // where all the heavy-lifting happens
  dp_alu alu(alufn,alu_a,alu_b,mem_addr);
  // select regfile write data from PC+4, alu output, and memory data
  dp_wdata wdata(wdsel,mem_data,mem_addr,pc_plus_4,wdata);
endmodule
```

Adder

```
module dp_addsub(fn,alu_a,alu_b,result,n,v,z);
  input fn;              // 0 for add, 1 for subtract
  input [31:0] alu_a;    // A operand
  input [31:0] alu_b;    // B operand
  output [31:0] result; // result
  output n,v,z;          // condition codes computed from result

  reg n,v,z;
  reg [31:0] result;

  always @(fn or alu_a or alu_b) begin: ripple
    integer i;           // FOR loop index, not in hardware
    reg cin,p,g;         // expanded at compile time into many signals
    reg [31:0] xb;       // hold's complement of ALU_B during subtract

    // simple ripple-carry adder for now
    xb = fn ? ~alu_b : alu_b;   // a - b == a + ~b + 1
    cin = fn;                   // carry-in is 0 for add, 1 for sub
    // remember: this FOR is expanded at *compile* time
    for (i = 0; i < 32; i = i + 1) begin
      p = alu_a[i] ^ xb[i];     // carry propagate
      g = alu_a[i] & xb[i];     // carry generate
      result[i] = p ^ cin;
      cin = g | (p & cin);      // carry into next stage
    end

    n = result[31];             // negative
    z = ~|result;               // zero
    v = (alu_a[31] & xb[31] & !n) | (~alu_a[31] & ~xb[31] & n);  // overflow
  end
endmodule
```

What's wrong with:

```
assign result = fn ? a - b : a + b;
```

Instruction decode (fragment)

```
…
reg [17:0] ctl;                    // local
always @(inst or interrupt) begin           // control rom
  if (interrupt)
                ctl = 16'b0100100000000000;  // interrupt
  else case (inst[31:26])
    //                      ppp   aaaaaaww
    //                    bcccw  llllllddd
    //                    tsssaabuuuuuuss
    //                    eeeesssffffffeex
    //                    sllleeennnnnnllw
    //                    t21011154321010r
    default:    ctl = 16'b0011100000000000;  // illegal opcode
    6'b011000: ctl = 16'b0000001000000100;  // LD
    6'b011001: ctl = 16'b0000001000000101;  // ST
    6'b011011: ctl = 16'b0010000000000000;  // JMP
…
    6'b111101: ctl = 16'b0000001100001010;  // SHRC
    6'b111110: ctl = 16'b0000001100011010;  // SRAC
  endcase
end

assign werf = ~ctl[0];
assign mem_we = !reset & ctl[0];
assign wdsel = ctl[2:1];
assign alufn = ctl[8:3];
assign bsel = ctl[9];
assign asel = ctl[10];
assign wa = ctl[11] ? 5'b11110 : inst[25:21];
assign pcsel = ((ctl[14:12] == 3'b001) & (ctl[15] ^ z)) ? 3'b000 : ctl[14:12];
…
```

Test Jig

```
module main;
 reg clk,reset,irq;

 wire [31:0] inst_addr,inst_data;
 wire [31:0] mem_addr,mem_rd_data,mem_wr_data;
 wire mem_we;

 reg [31:0] memory[1023:0];

 beta beta(clk,reset,irq,inst_addr,inst_data,
           mem_addr,mem_rd_data,mem_we,mem_wr_data);

 // main memory (2 async read ports, 1 sync write port)
 assign inst_data = memory[inst_addr[13:2]];
 assign mem_rd_data = memory[mem_addr[13:2]];
 always @(posedge clk)
   if (mem_we) memory[mem_addr[13:2]] <= mem_wr_data;

 always #5 clk = ~clk;
 initial begin
   $dumpfile("beta_checkoff.vcd");
   $dumpvars;
   $readmemh("beta_checkoff",memory);
   clk = 0; irq = 0; reset = 1;
   #10
   reset = 0;
   #3000    // 300 cycles
   $finish;
 end
endmodule
```

About 20 megabytes for 300 cycles!

```
@0  // starting address
77df000a
77ff0003
6ffe0000
c3e00000
77ffffe
d01e0004
77e00002
...
```