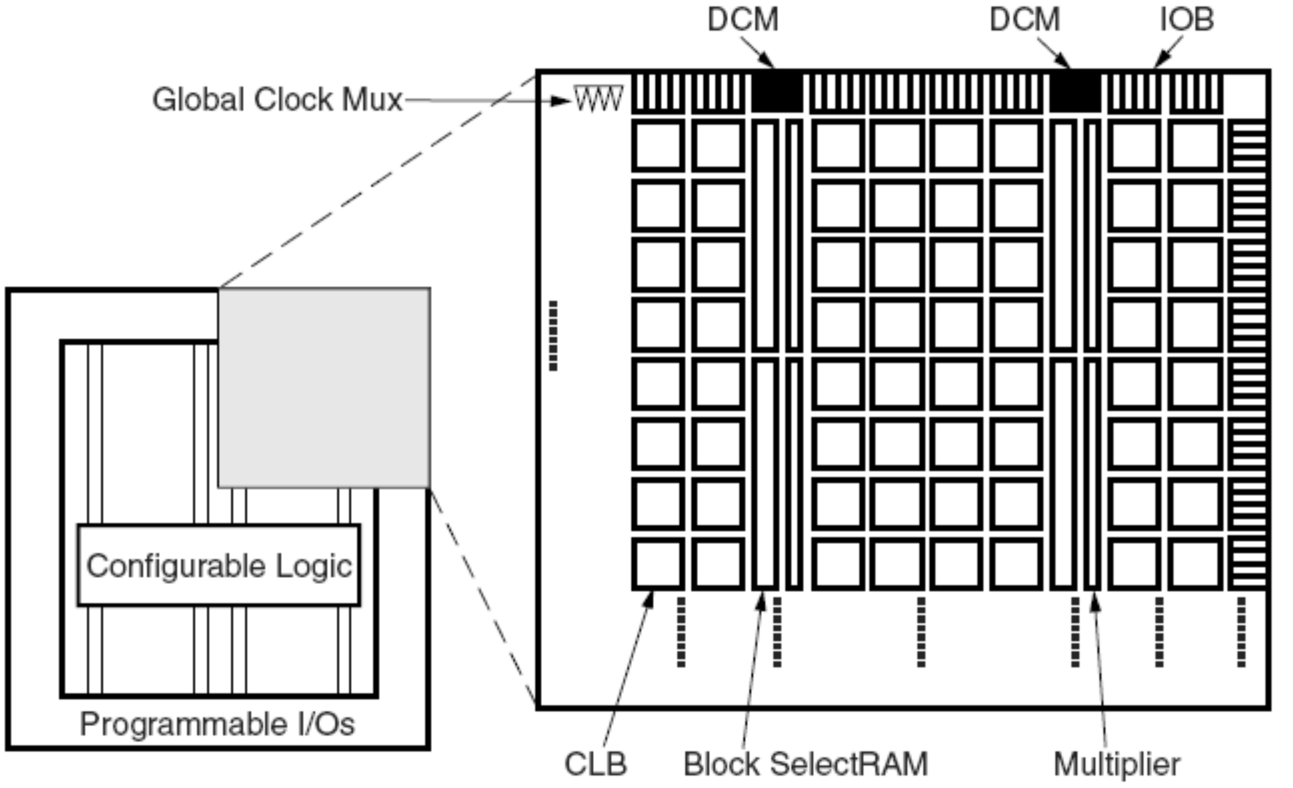
FPGAs & Synthesizable Verilog

- Quick tour of the Virtex 5
- Verilog
  - -- Modules
  - -- Combinational Logic (gates)
  - -- Sequential Logic (registers)
  - -- Memories
- Beehive Verilog tree

Xilinx Virtex V FPGA

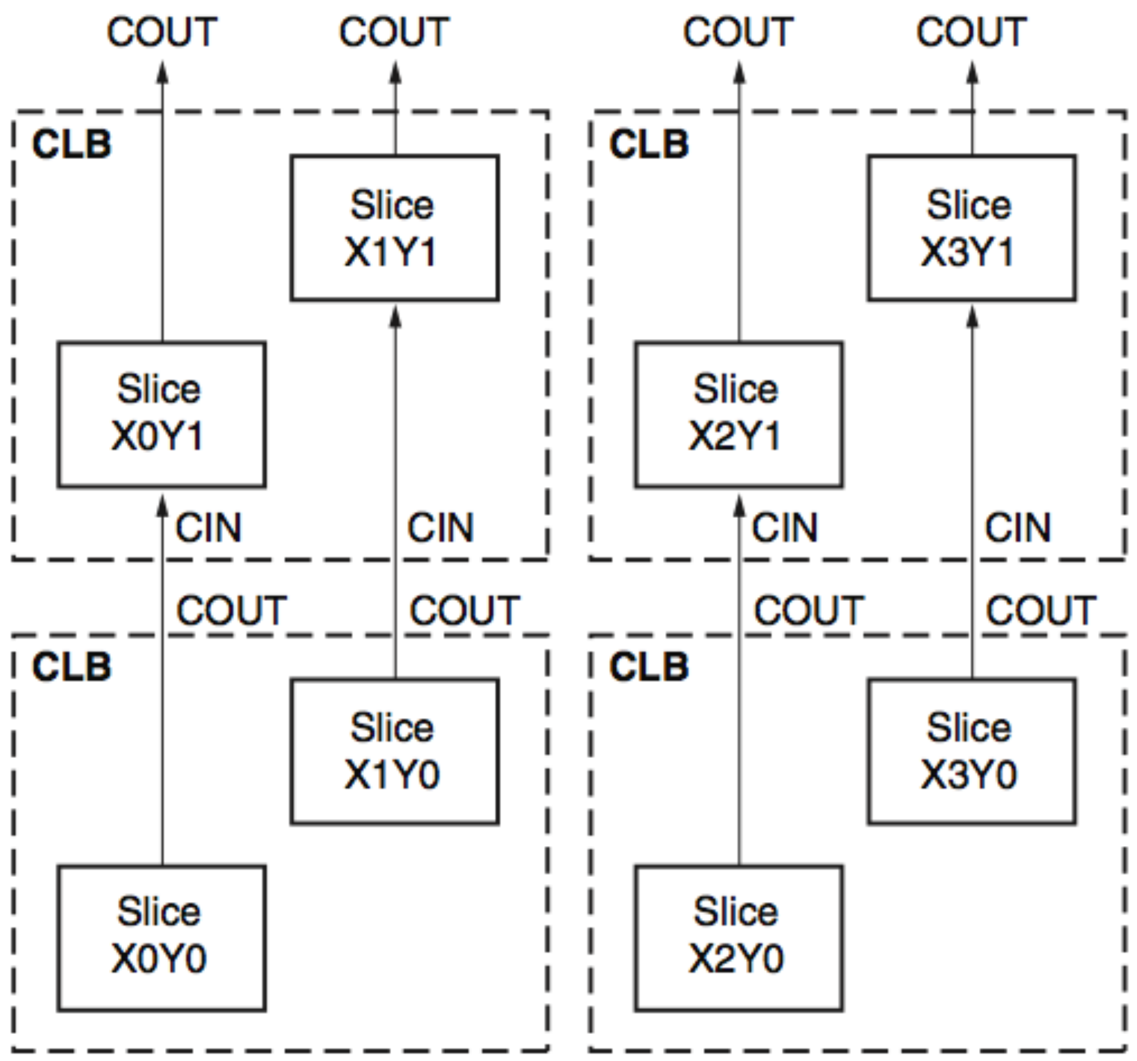


XC5VLX110T:

- 1136 pins, 640 IOBs
- CLB array: 54 cols x 160 rows = 69,120 LUTs
- 148 36Kbit BRAMs = 5.3Mbits
- 64 DSP48E (25x18 mul, 48-bit adder, acc)
- PCIe, 10/100/1000 Mb/s Ethernet MAC

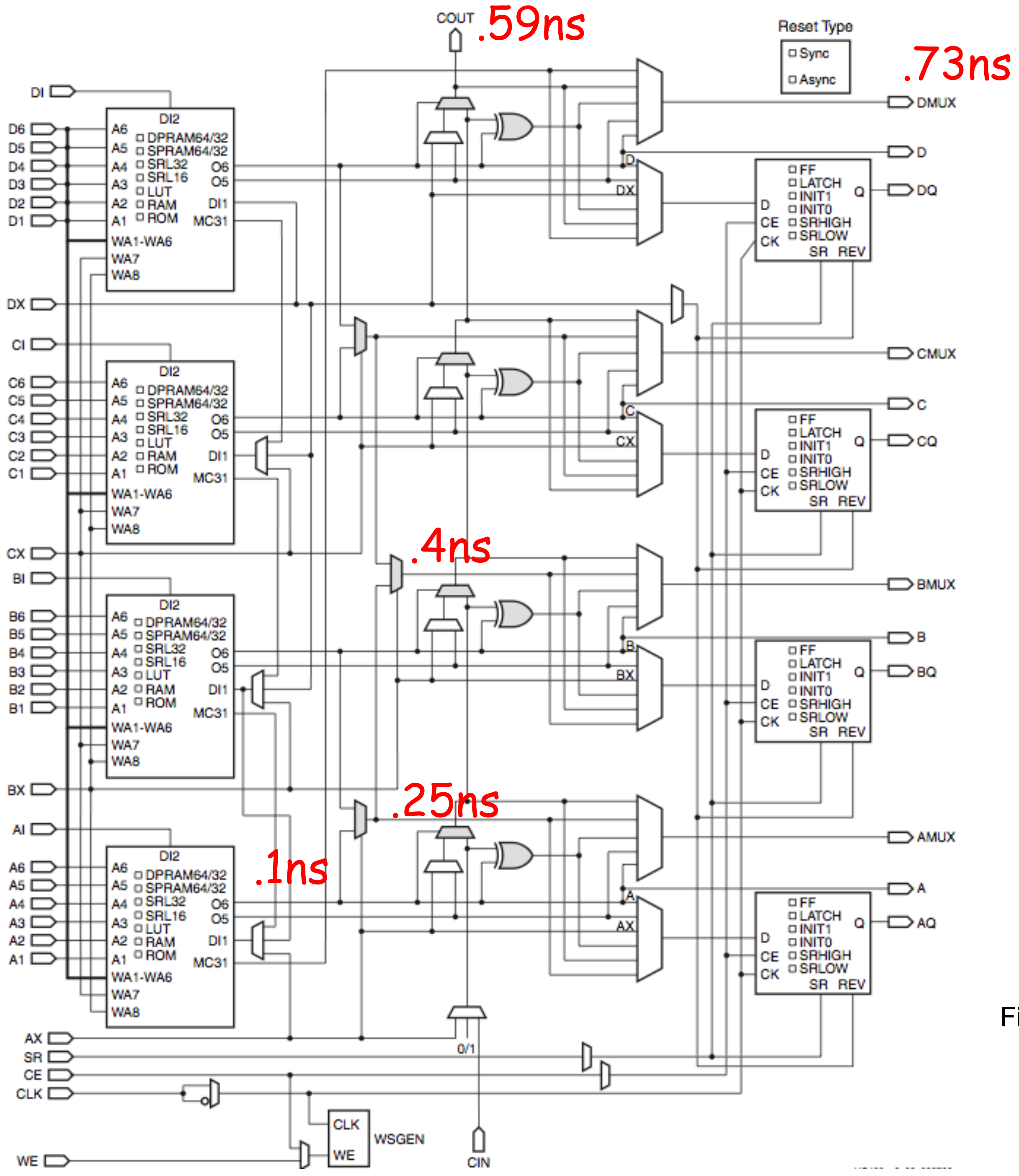
Figures from Xilinx Virtex II datasheet

Virtex V CLB



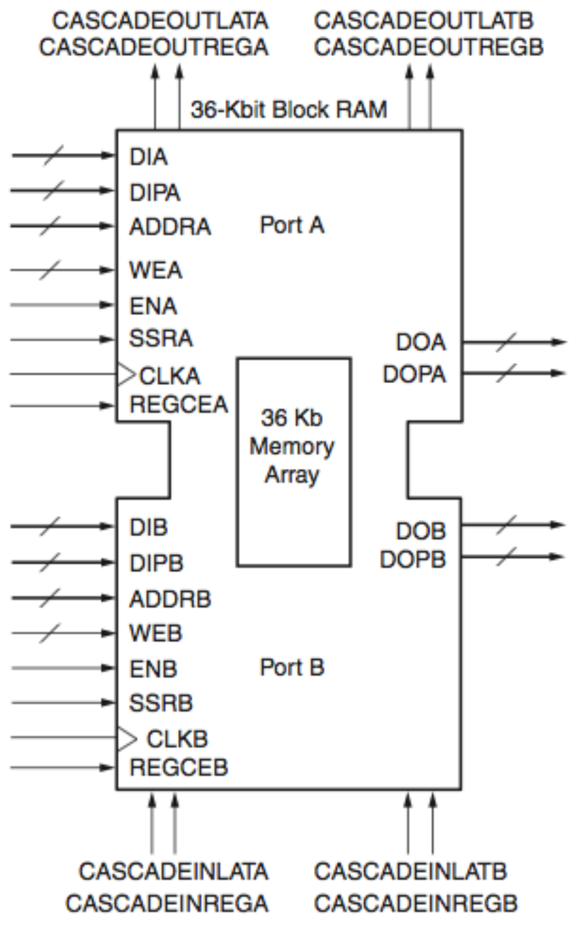
Figures from Xilinx Virtex V datasheet

Virtex 5 Slice Schematic

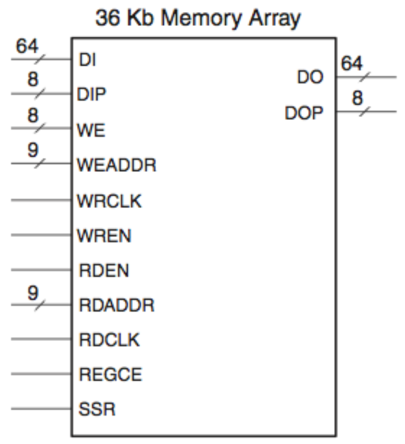


Figures from Xilinx Virtex V datasheet

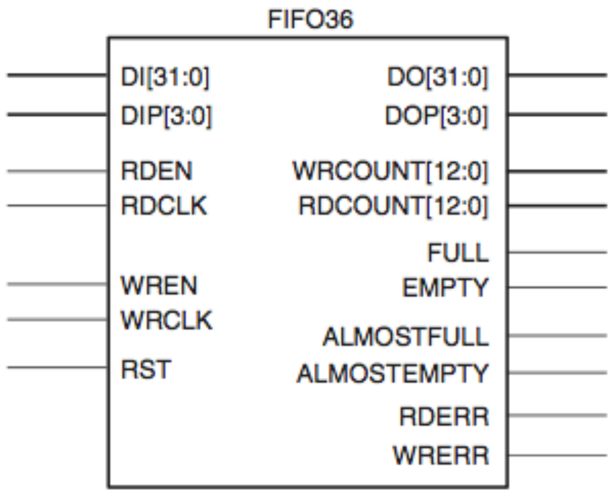
Block Memories (BRAMs)



32k x 1 – 1k x 36
cascadable

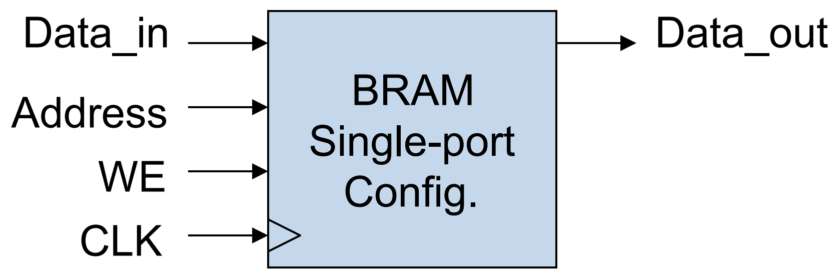


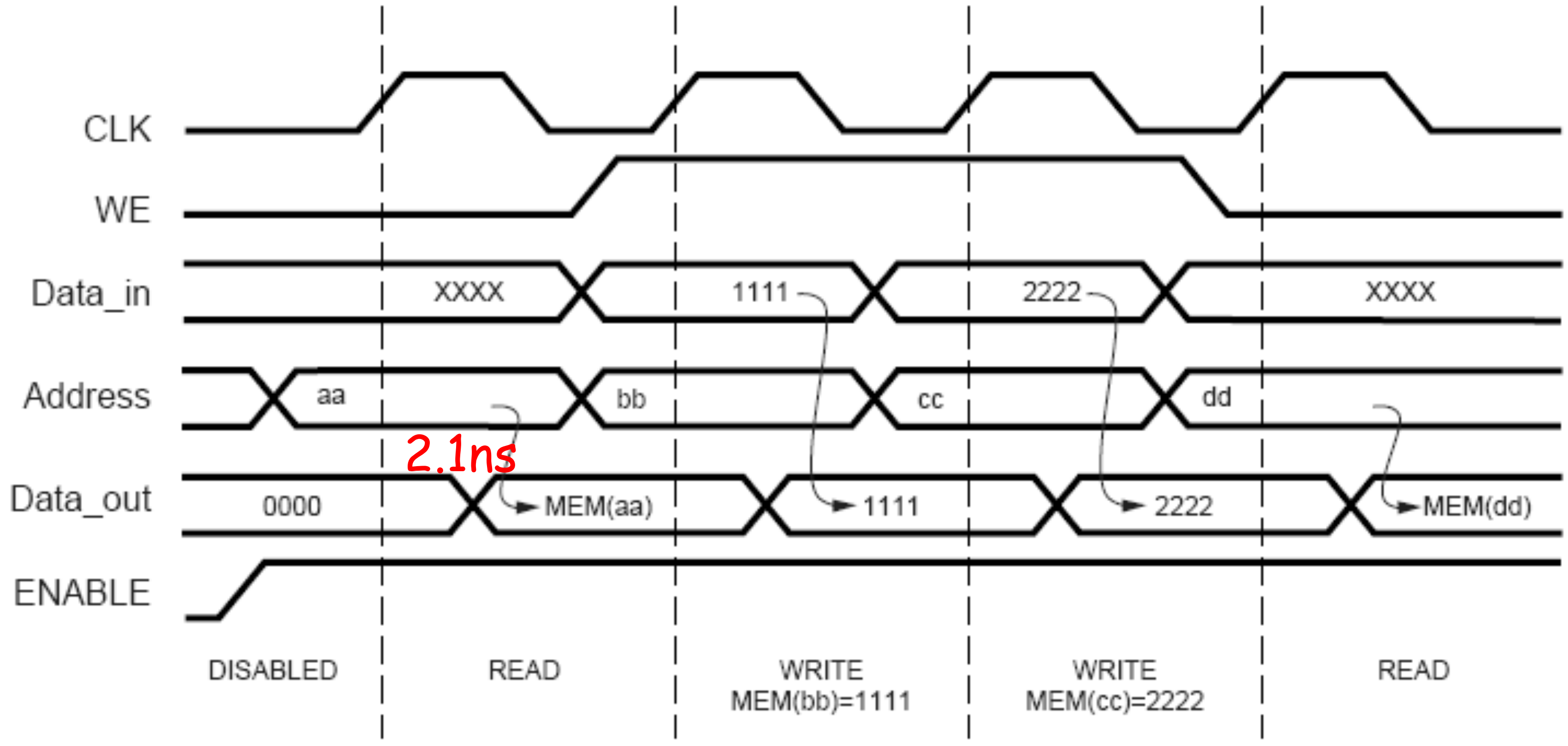
512 x 72



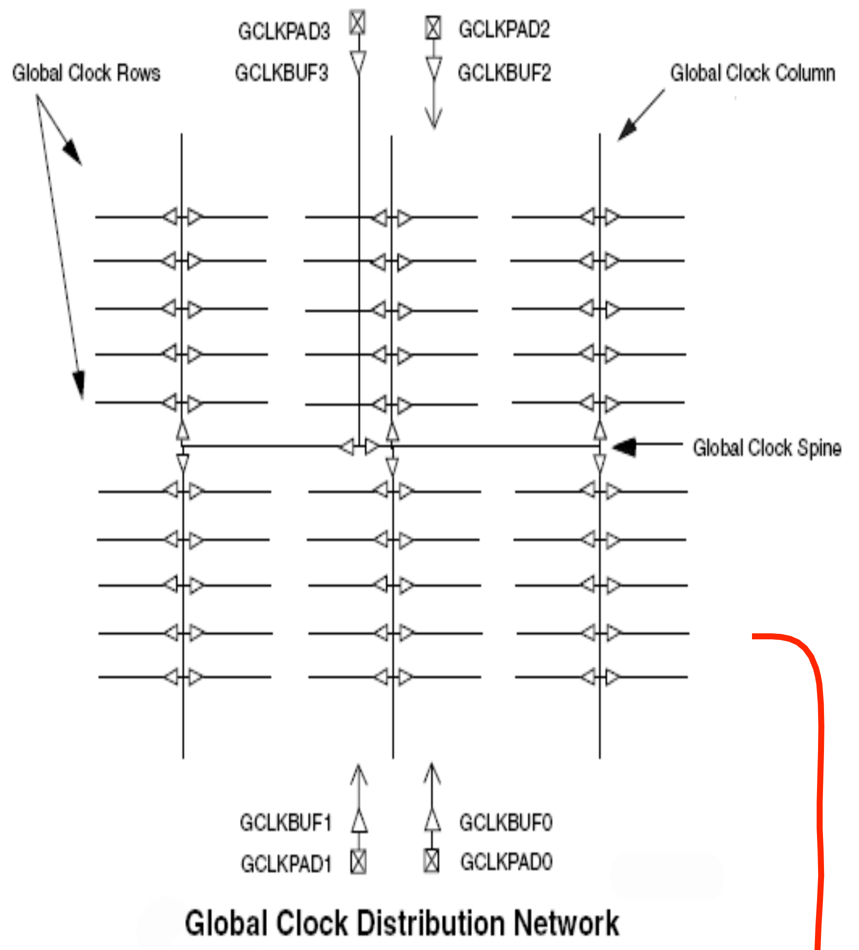
8k x 4 – 512 x 72

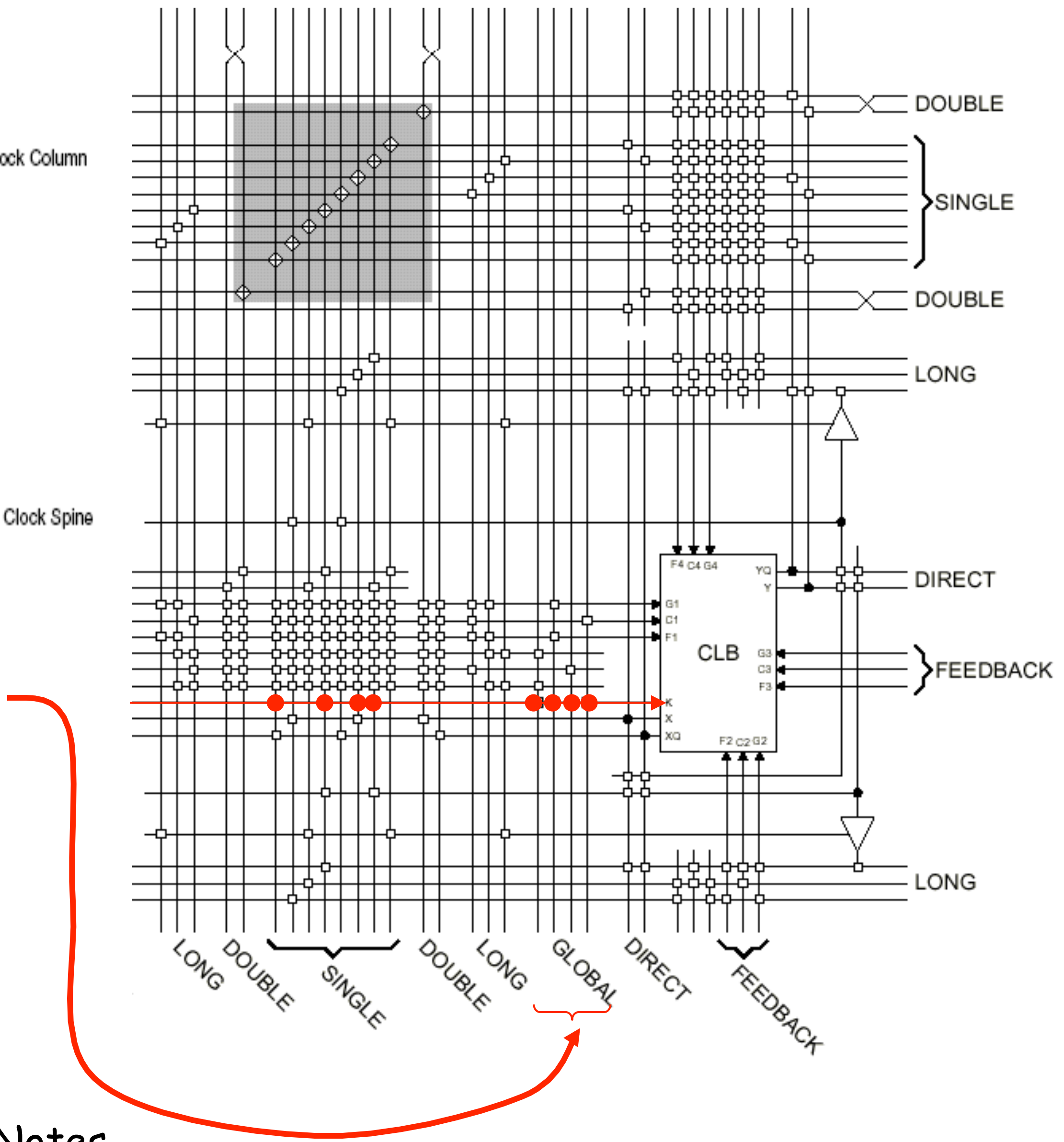
BRAM Operation





Wiring in FPGAs





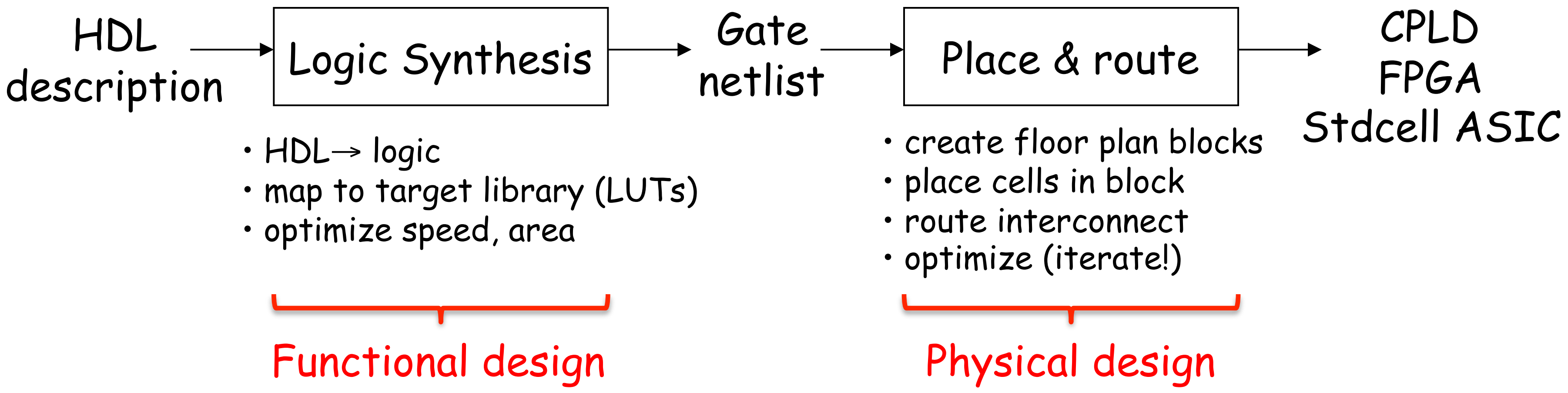
Figures from Xilinx App Notes

Using an HDL description

Using Verilog you can write an executable functional specification that

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

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



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`) which are used to communicate with the module.

Don't forget this ";"

```
// single-line comments
/* multi-line
   comments
*/
module name(input a,b,input [31:0] c,output z,output reg [3:0] s);

  // declarations of internal signals, registers

  // combinational logic: assign

  // sequential logic: always @ (posedge clock)

  // module instances

endmodule
```

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.

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
```

LHS must be of type `wire`

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.

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

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 ~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 >= 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

Numeric Constants

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

```
123          // default: decimal radix, 32-bit 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.

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.

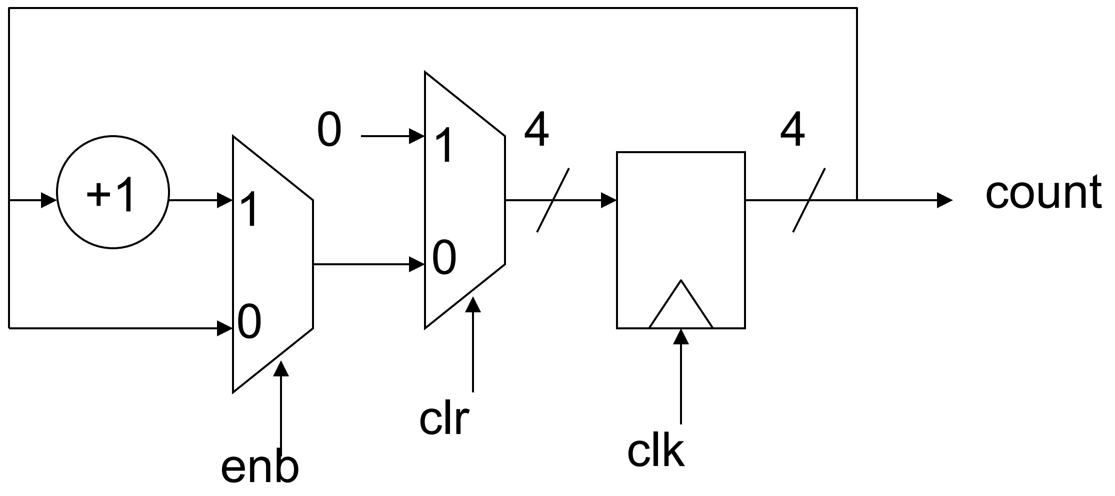
This type of hierarchical behavioral model is called "structural" since we're building up a structure of instances connected by wires. We often mix dataflow and structural modeling when describing a module's behavior.

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

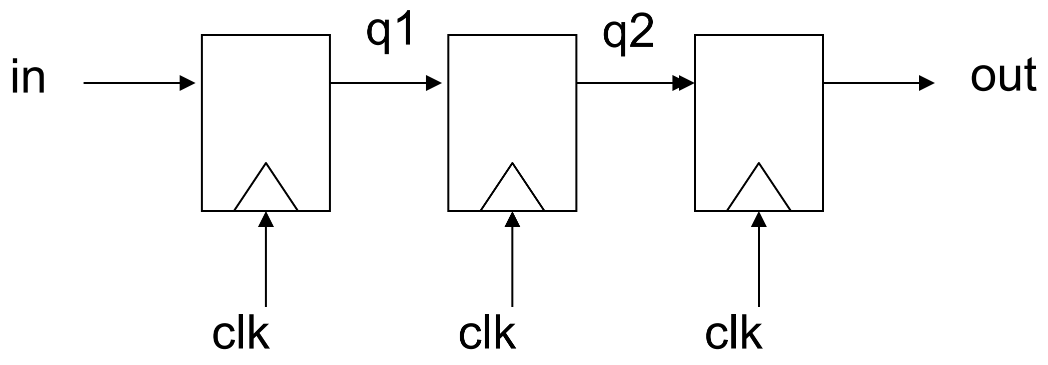
Example: A Simple Counter



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

Inside always: LHS must be of type reg, always use <=

Example: Shift Register



```
// shift register
reg q1,q2,out;
always @(posedge clk) begin
  q1 <= in;
  q2 <= q1;
  out <= q2;
end
```

```
// shift register
reg q1,q2,out;
always @(posedge clk) q1 <= in;
always @(posedge clk) q2 <= q1;
always @(posedge clk) out <= q2;
```

Non-blocking assignment (<=) semantics:
1) evaluate all RHS expressions in all active blocks
2) after evals complete, assign new values to LHS

FPGA Memories

- Distributed memory (built using LUTs as RAMs)
  - Combinational (w/o clock) read + sync (w/ clock) write
  - 32/64/128/256 x 1 using a one or more LUTs
  - Wider using multiple LUTs in parallel
  - Multiple read ports; fake by building multiple copies of memory
  - `(* ram_style = “distributed” *)` pragma
- Block memory (built using BRAMs)
  - True dual port: two read/write ports
  - Both reads and writes are synchronous (need clock edge!)
  - Widths of 1 to 36 bits, Depths of 32k to 1k
  - Special 512 x 72 hack
  - FIFO support built-in
  - `(* ram_style = “block” *)` pragma

Example: register file

```
// 16-entry 32-bit register file
(* ram_style = “distributed” *)
reg [31:0] regfile[15:0];
wire [4:0] a_addr,b_addr,w_addr;
wire [31:0] a_data,b_data,w_data;
wire weRF;

// async read
assign a_data = regfile[a_addr];
assign b_data = regfile[b_addr];

// sync write
always @(posedge clk)
  if (weRF & w_addr != 0) regfile[w_addr] <= w_data;
```

Example: instruction cache

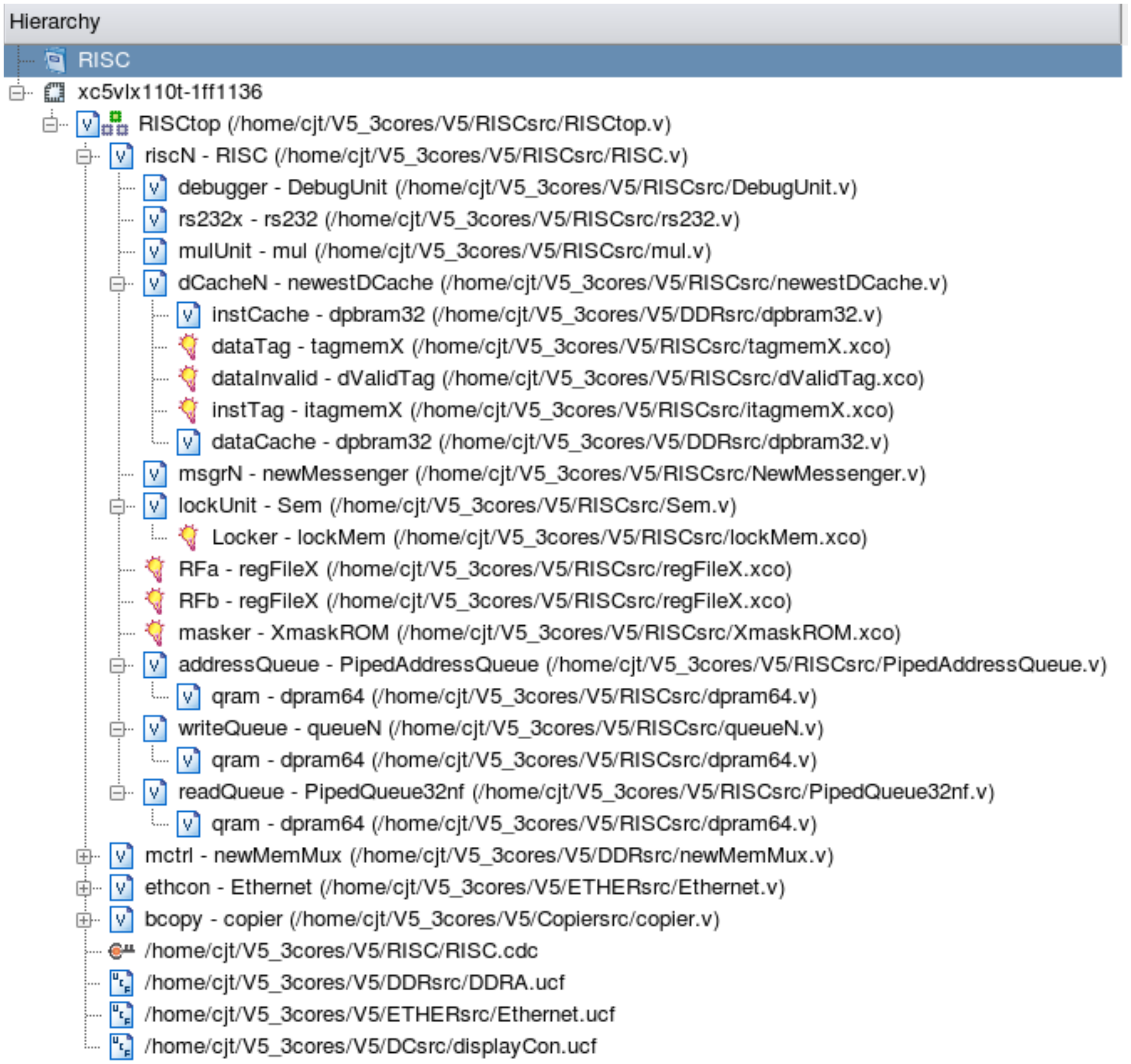
```
(* ram_style = "block" *)
reg [31:0] instCache[1023:0];  // 1k x 32 bram
reg [9:0] instAddr;
always @(posedge clock) begin
   if (~stall | ~Ihit) instAddr <= Iaddr;
   if (~Dmiss & (RDdest == whichCore))
     instCache[{pcx[9:3], cacheAddr}] <= RDreturn;
end
assign instx = instCache[instAddr];


(* ram_style = "distributed" *)
reg [20:0] instTag[127:0];  // 128 x 21 distributed mem
assign Itag = instTag[pcx[9:3]];
always @(posedge clock) begin
  if (writeItag) instTag[pcx[9:3]] <= pcx[30:10];
end;
```

Verilog Links

- Quick reference manual for "modern" Verilog (Verilog-2001) w/ examples:
    - http://www.sutherland-hdl.com/online_verilog_ref_guide/verilog_2001_ref_guide.pdf

- Open-source Verilog simulation
    - http://www.icarus.com/eda/verilog/
    - http://gtkwave.sourceforge.net/

Beehive Verilog



```
Hierarchy
RISC
xc5vlx110t-1ff1136
  RISCtop (/home/cjt/V5_3cores/V5/RISCsrc/RISCtop.v)
    riscN - RISC (/home/cjt/V5_3cores/V5/RISCsrc/RISC.v)
      debugger - DebugUnit (/home/cjt/V5_3cores/V5/RISCsrc/DebugUnit.v)
      rs232x - rs232 (/home/cjt/V5_3cores/V5/RISCsrc/rs232.v)
      mulUnit - mul (/home/cjt/V5_3cores/V5/RISCsrc/mul.v)
      dCacheN - newestDCache (/home/cjt/V5_3cores/V5/RISCsrc/newestDCache.v)
        instCache - dpbram32 (/home/cjt/V5_3cores/V5/DDRsrc/dpbram32.v)
        dataTag - tagmemX (/home/cjt/V5_3cores/V5/RISCsrc/tagmemX.xco)
        dataInvalid - dValidTag (/home/cjt/V5_3cores/V5/RISCsrc/dValidTag.xco)
        instTag - itagmemX (/home/cjt/V5_3cores/V5/RISCsrc/itagmemX.xco)
        dataCache - dpbram32 (/home/cjt/V5_3cores/V5/DDRsrc/dpbram32.v)
      msgrN - newMessenger (/home/cjt/V5_3cores/V5/RISCsrc/NewMessenger.v)
      lockUnit - Sem (/home/cjt/V5_3cores/V5/RISCsrc/Sem.v)
        Locker - lockMem (/home/cjt/V5_3cores/V5/RISCsrc/lockMem.xco)
      RFa - regFileX (/home/cjt/V5_3cores/V5/RISCsrc/regFileX.xco)
      RFb - regFileX (/home/cjt/V5_3cores/V5/RISCsrc/regFileX.xco)
      masker - XmaskROM (/home/cjt/V5_3cores/V5/RISCsrc/XmaskROM.xco)
      addressQueue - PipedAddressQueue (/home/cjt/V5_3cores/V5/RISCsrc/PipedAddressQueue.v)
        qram - dpram64 (/home/cjt/V5_3cores/V5/RISCsrc/dpram64.v)
      writeQueue - queueN (/home/cjt/V5_3cores/V5/RISCsrc/queueN.v)
        qram - dpram64 (/home/cjt/V5_3cores/V5/RISCsrc/dpram64.v)
      readQueue - PipedQueue32nf (/home/cjt/V5_3cores/V5/RISCsrc/PipedQueue32nf.v)
        qram - dpram64 (/home/cjt/V5_3cores/V5/RISCsrc/dpram64.v)
    mctrl - newMemMux (/home/cjt/V5_3cores/V5/DDRsrc/newMemMux.v)
    ethcon - Ethernet (/home/cjt/V5_3cores/V5/ETHERsrc/Ethernet.v)
    bcopy - copier (/home/cjt/V5_3cores/V5/Copiersrc/copier.v)
    /home/cjt/V5_3cores/V5/RISC/RISC.cdc
    /home/cjt/V5_3cores/V5/DDRsrc/DDRA.ucf
    /home/cjt/V5_3cores/V5/ETHERsrc/Ethernet.ucf
    /home/cjt/V5_3cores/V5/DCsrc/displayCon.ucf
```

Simulating Beehive

- ssh beehive@ra.csail.mit.edu
- mkdir yourname
- cd yourname
- tar xfz ../beehive_sim.tgz
- cd beehive
- make | more

Verilog Assignment #1

- Current behavior of lock unit:
  - P
    - Read i/o space with AQ[2:0] = 5, AQ[8:3] = lock #
    - Returns 2 if core already has lock
    - Otherwise sends Preq message on ring, converted to Pfail if another core owns lock
      - If Preq makes it all around the ring, set lock, return 1
      - If Preq converted to Pfail, return 0
  - V
    - Write i/o space with AQ[2:0] = 5, AQ[8:3] = lock #
    - If core owns lock, clear lock bit
    - If core doesn't own lock, send Vreq message on ring, which causes owner to clear their lock bit
- New behavior
  - Return 2 if core already has lock, or if it was the previous owner of the lock and no Preq messages have been seen for that lock (in which case set the lock bit). Hint: need more than two lock states…

Verilog Assignment #2

- Implement broadcast Messages
- Suggestions
  - Use a message destination of 0 to indicate broadcast
  - Modify messenger to receive messages destined for either its core number or 0
  - Modify messenger to remove broadcast messages it placed on the ring