Life Cycle of a Memory Request

1. Use AQR or AQW to place address in AQ

2. If A[31] == 0, check for hit in DCache

3. Read Hit: place cache word in RQ;
Write Hit: replace cache word with WQ

2.5. Miss: place new cache line read Address slot on ring.
If dirty, follow with 8 WriteData slots & old cache line write Address slot.

Monitor RDdest, on match copy RDreturn to cache. On 8th word, write tag. DCache will process a "hit" on next cycle.

Ring Example: 2 requests for lock 17

Both core #1 and core #2 want lock 17...

Core #1: waiting for Token
Core #2: waiting for Token
**Ring Example: 2 requests for lock 17**

Core #1: rewrites Token count to add a slot, skipping 00 slots
Core #2: waiting for Token

**Ring Example: 2 requests for lock 17**

Core #1: add Preq slot to train
Core #2: rewrites Token count to add a slot, skipping 01 slots
Ring Example: 2 requests for lock 17

Core #1: waiting for Preq or Pfail reply
Core #2: skipping slot, 00 remaining

Core #1: waiting for Preq or Pfail reply
Core #2: add Preq slot to train
Ring Example: 2 requests for lock 17

Core #1: waiting for Preq or Pfail reply
Core #2: waiting for Preq or Pfail reply

Core #1: waiting for Preq or Pfail reply
Core #2: waiting for Preq or Pfail reply
Ring Example: 2 requests for lock 17

Core #1: sees its Preq, grab lock 17, rewrite as Null
Core #2: waiting for Preq or Pfail reply

Core #1: Preq for lock it owns, rewrite as Pfail
Core #2: waiting for Preq or Pfail reply
Ring Example: 2 requests for lock 17

Core #1: idle (forwarding slots)
Core #2: sees its Pfail, fails lock request, rewrite as Null
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

HDL description → Logic Synthesis → Gate netlist → Place & route → CPLD FPGA

- HDL→ logic
- map to target library (LUTs)
- optimize speed, area

- create floor plan blocks
- place cells in block
- route interconnect
- optimize (iterate!)

Functional design → Physical design

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.

```verilog
// 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:

```verilog
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:

```verilog
{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:

```verilog
// 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.
**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

<table>
<thead>
<tr>
<th>Bitwise</th>
<th>Reduction</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>~a</td>
<td>!a</td>
<td>NOT</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>&amp;a</td>
<td>AND</td>
</tr>
<tr>
<td>a | b</td>
<td>|a</td>
<td>OR</td>
</tr>
<tr>
<td>a ^ b</td>
<td>^a</td>
<td>XOR</td>
</tr>
<tr>
<td>a ^~ b</td>
<td>^~a</td>
<td>XNOR</td>
</tr>
<tr>
<td>a ^~ b</td>
<td>a ^~ b</td>
<td></td>
</tr>
</tbody>
</table>

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

**Other operators**

- **Conditional**
  - `a ? b : c` if `a` then `b` else `c`
  - Synthesizes to a mux

- **Relational**
  - `a > b` greater than
  - `a >= b` greater than or equal
  - `a < b` Less than
  - `a <= b` Less than or equal
  - Synthesizes to a subtract

- **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*

* expensive in HW
** not synthesizable
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 two's-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

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

Verilog Links

- Quick reference manual for “modern” Verilog (Verilog-2001) w/ examples:

- Open-source Verilog simulation
When AQ[31]==1, AQ[2:0] select the target I/O unit. When unit is done (one or more cycles) it signals completion by reading from AQ, bringing the next address to the front of the queue.
Message Module

• Can send message of 0 to 63 words to destination core
  – Use one ring slot for message header
    • Ring[31:0] = source core (4), message type (4), length (6)
    • SrcDest[3:0] = destination core #
    • SlotType = Message
  – Use following <length> ring slots for message body
    • Ring[31:0] = next word of message
    • SrcDest[3:0] = destination core #
    • Slot Type = Message

• Can receive up to 1024 message words into a fifo
  – Monitor ring, copying Ring[31:0] to fifo when slot is a message directed to us.
  – Received Message slots are rewritten as Null.
  – 0-length messages are treated as directions for Debug Unit
  – CPU can poll to read next word from fifo (or get empty indication)