ECE 385 – Digital Systems Laboratory

Lecture 20 – Memory, Testbench and Verification

Zuofu Cheng

Fall 2018

[Link to Course Website]

ECE ILLINOIS

ILLINOIS

On-Chip Memory

- Tightly coupled to FPGA logic (on the same die)
- **Synchronous** SRAM technology, organized internally as 256x36
- Can be rearranged (e.g. 256x36, 512x18, 1024x9, etc…)
- Single port or dual port, single or dual clock
- Can be used as addressed memory or FIFO
- Maximum clock = 274 MHz (e.g. can run as fast as logic in almost all cases)

On-Chip Memory Continued

- OCM must be **synchronous**
- M9K blocks cannot be used for asynchronous memory
- Latches and/or combinational logic will be used for **asynchronous** memories (typically considered bad design, latches are poor use of FPGA area)
- Check "total memory bits" in Place & Route report to see if memory successfully inferred
- This is why InvSubBytes has a clock (even though it is just a ROM)

On-Chip Memory Continued

- Can simply instantiate using SystemVerilog
- FPGA Synthesis will infer (automatically use) M9K blocks if it can understand HDL
- Refer to *Recommended HDL coding Styles* to make sure inference works
- Can also instantiate in Qsys for use in SoC (e.g. as program memory)
- Alternatively, you can use the GUI tools (Megafunction) which gives you SystemVerilog module instantiation template

```
module ram_32x8(
 output logic [7:0] q,
 input [7:0] d,
 input [4:0] write_address, read_address,
 input we, clk
);

logic [7:0] mem [32];

Always_ff @ (posedge clk) begin
      if (we)
      mem[write_address] <= d;
      q <= mem[read_address];
end
endmodule
```

On-Chip Memory Continued

- Do **not** copy the lab 6 test_memory.sv, it is designed for simulation, not synthesis
- Although it will work for small memories (such as in lab 6), it will cause **very long** compilation times for larger memories…why?

```
...
always_ff @ (posedge Clk or posedge Reset)
begin
if(Reset)     // Insert initial memory contents here
begin
      mem_array[   0 ] <=     opCLR(R0)                 ;       // Clear the
      register so it can be used as a base
      mem_array[   1 ] <=     opLDR(R1, R0, inSW)       ;       // Load switches
      mem_array[   2 ] <=     opJMP(R1)                 ;       // Jump to the start
      of a program
                                                                // Basic I/O test 1
      mem_array[   3 ] <=     opLDR(R1, R0, inSW)       ;       // Load switches
      mem_array[   4 ] <=     opSTR(R1, R0, outHEX)     ;       // Output
      mem_array[   5 ] <=     opBR(nzp, -3)             ;       // Repeat
      ...
```

On-Chip Memory Continued

- If you want to initialize OCM, you should use the $readmem as specified in *Recommended HDL Coding Styles*
- This will create a MIF (memory initialization file) which initializes the OCM at **programming**
- Instead, use $readmem as specified in document, or initialize using GUI Megafunction
- $readmemb, $readmemh, etc… are special commands recognized by the synthesis tool for binary, hex data
- Place in initial procedure block
- Even though initial is typically unsynthesizable, it doesn't actually synthesize the $readmem command, the synthesis tool simply reads the file at compile time and initializes the contents

```
...
logic [7:0] ram[16];
initial
begin
     $readmemh("ram.txt", ram);
end...

ram.txt:
24
34
2e
2e
2e
2e
```

Introduction to SDRAM

- SDRAM stands for ***synchronous*** *dynamic random access memory*
- Dynamic memory = made from small capacitors instead of logic gates
  - Dynamic memory has much higher density than static memory
  - Must be refreshed periodically to hold value (typically once every couple of milliseconds)
  - Traditionally addressed with multiplexed address lines (in a row/column fashion due to high density)
- Synchronous = clocked
  - Responds to commands as signaled by CLE/CAS/RAS/WE
  - Typically driven by a state machine which is called the *SDRAM controller*
  - This abstracts the commands so that SDRAM looks like standard memory

DRAM Basic Cell

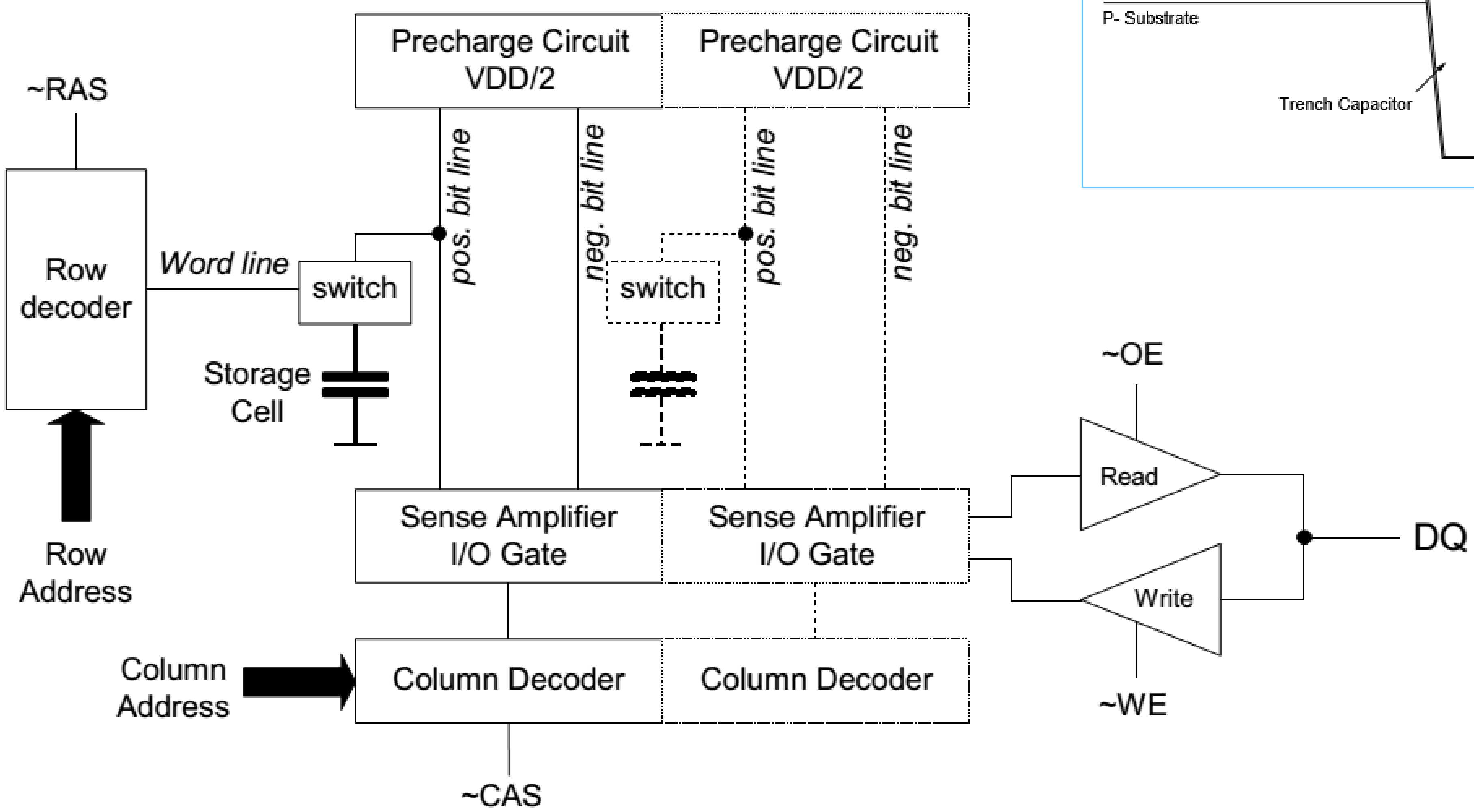
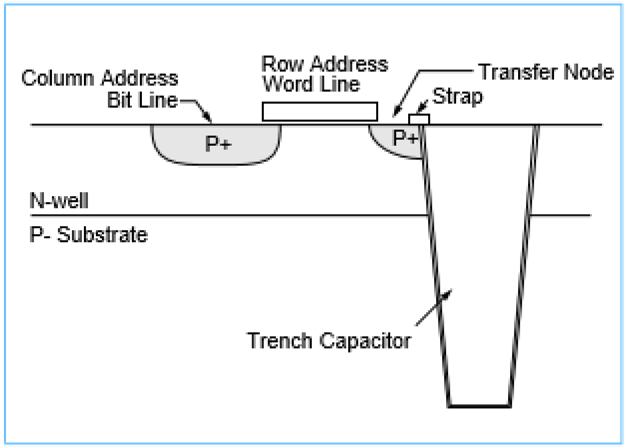
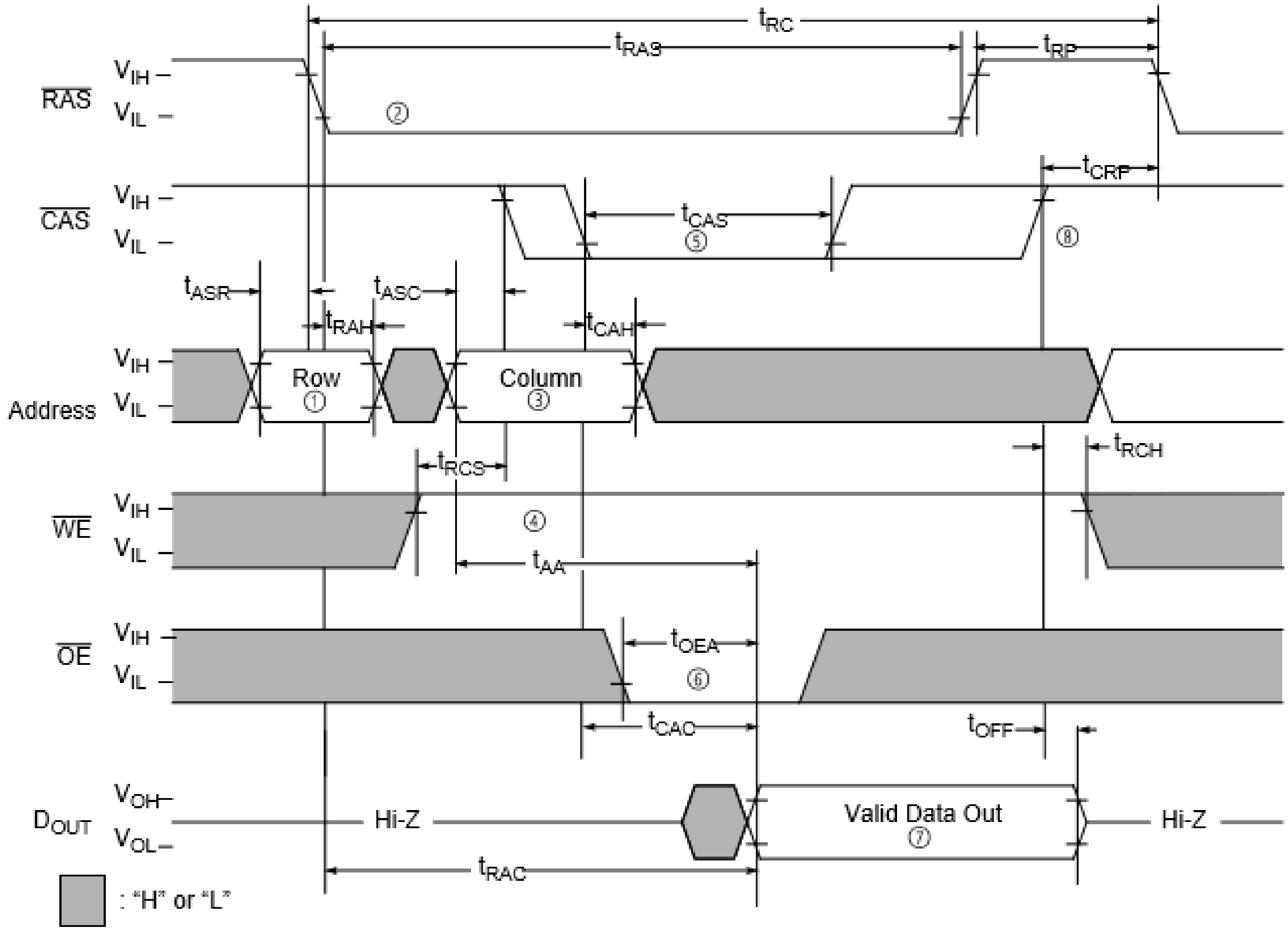


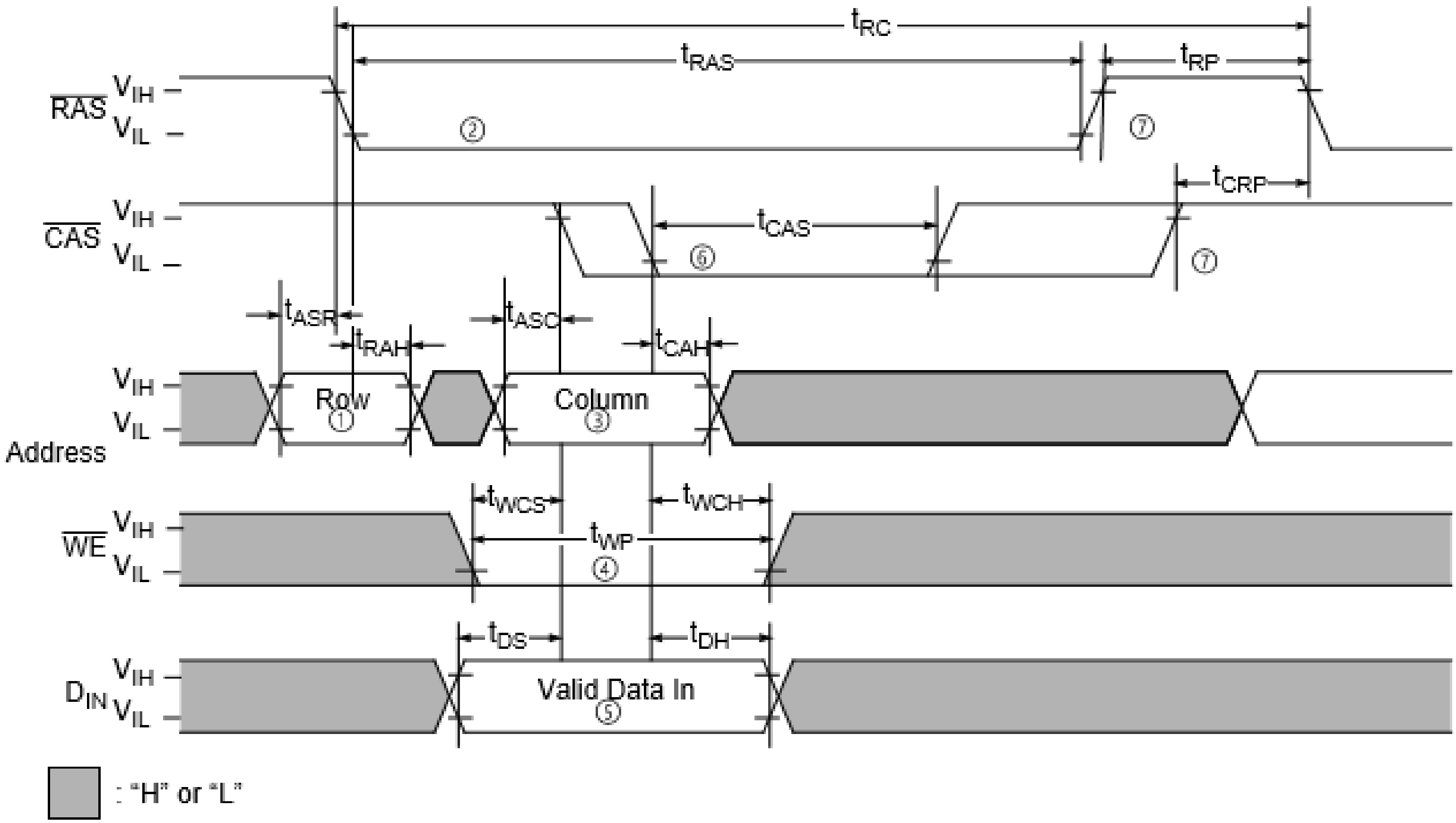
Figure 1: IBM Trench Capacitor Memory Cell



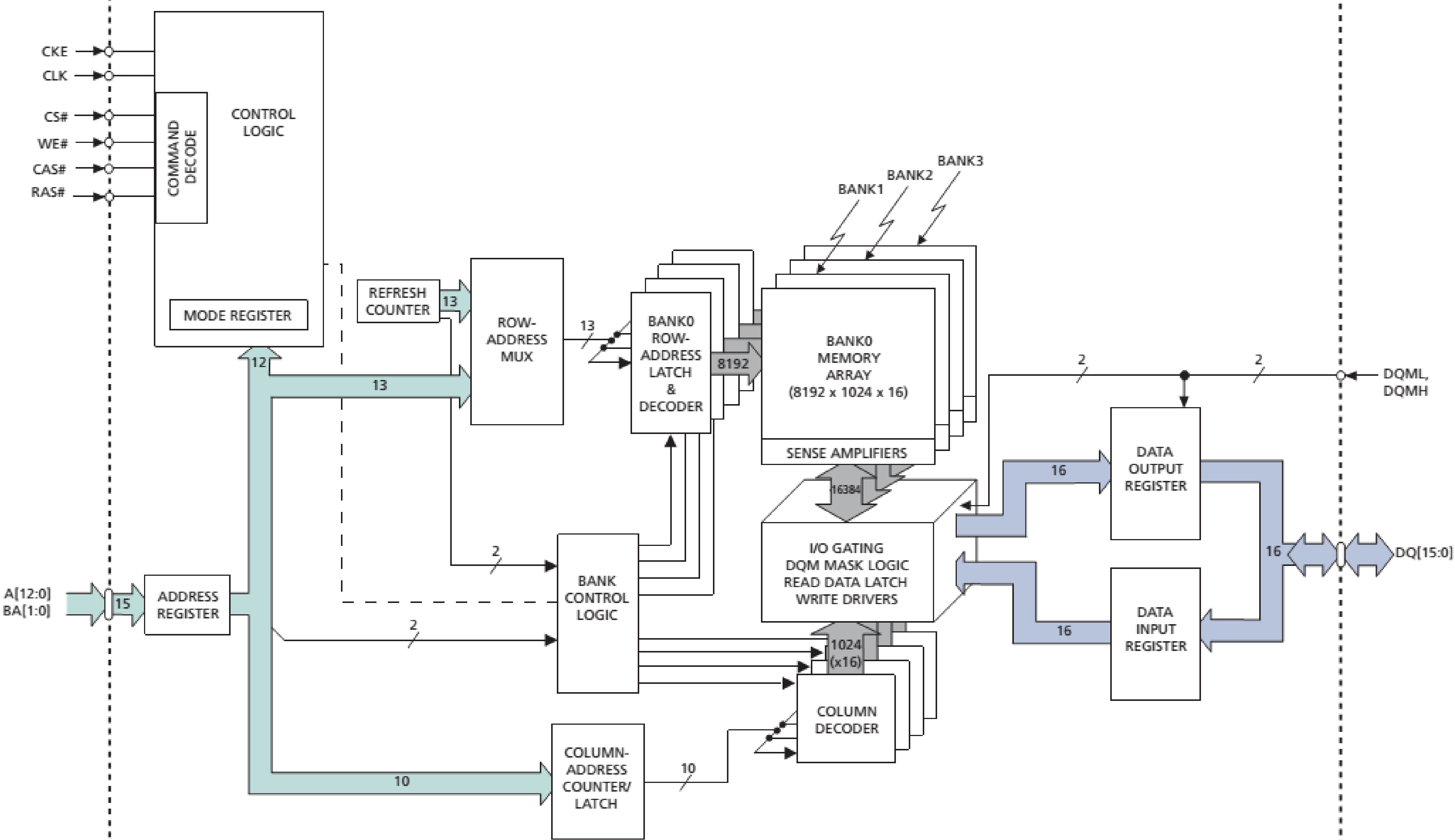
DRAM Read Cycle



DRAM Write Cycle



SDRAM Structure



SDRAM Commands

- Commands are combinations of CLE/CAS/RAS/WE
- Note that SDRAM addresses are multiplex, typically rows are selected before columns using the same address pins

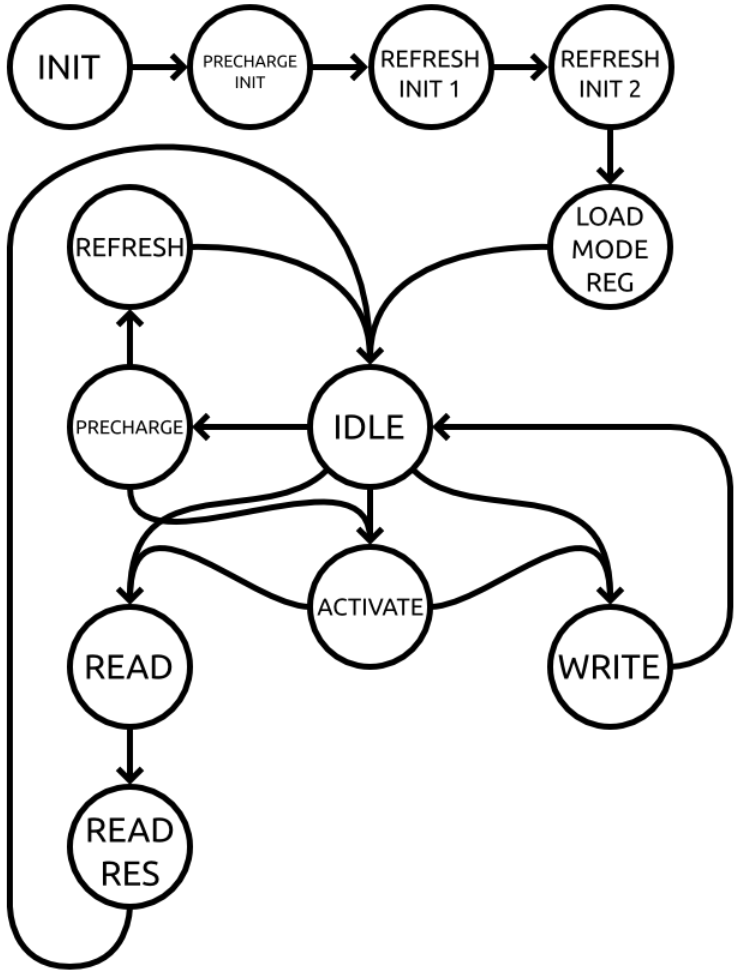
Table 13: Truth Table – Commands and DQM Operation

Note 1 applies to all parameters and conditions

Name (Function)	CS#	RAS#	CAS#	WE#	DQM	ADDR	DQ	Notes
COMMAND INHIBIT (NOP)	H	X	X	X	X	X	X	
NO OPERATION (NOP)	L	H	H	H	X	X	X	
ACTIVE (select bank and activate row)	L	L	H	H	X	Bank/row	X	2
READ (select bank and column, and start READ burst)	L	H	L	H	L/H	Bank/col	X	3
WRITE (select bank and column, and start WRITE burst)	L	H	L	L	L/H	Bank/col	Valid	3
BURST TERMINATE	L	H	H	L	X	X	Active	4
PRECHARGE (Deactivate row in bank or banks)	L	L	H	L	X	Code	X	5
AUTO REFRESH or SELF REFRESH (enter self refresh mode)	L	L	L	H	X	X	X	6, 7
LOAD MODE REGISTER	L	L	L	L	X	Op-code	X	8
Write enable/output enable	X	X	X	X	L	X	Active	9
Write inhibit/output High-Z	X	X	X	X	H	X	High-Z	9

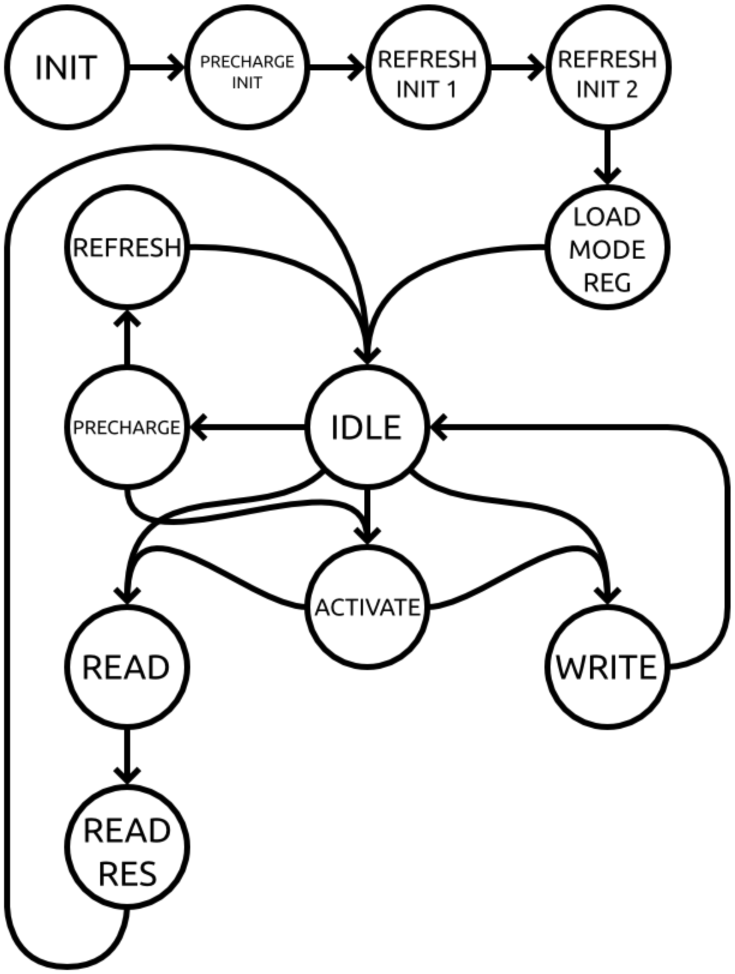
SDRAM Controller State Machine

- SDRAM operates using commands
- Controller needs internal timer to know when to send refresh operation
- SDRAM chip needs to be initialized to get into the IDLE state
- All SDRAM operations happen into a single row
- ACTIVE command initializes row for R/W



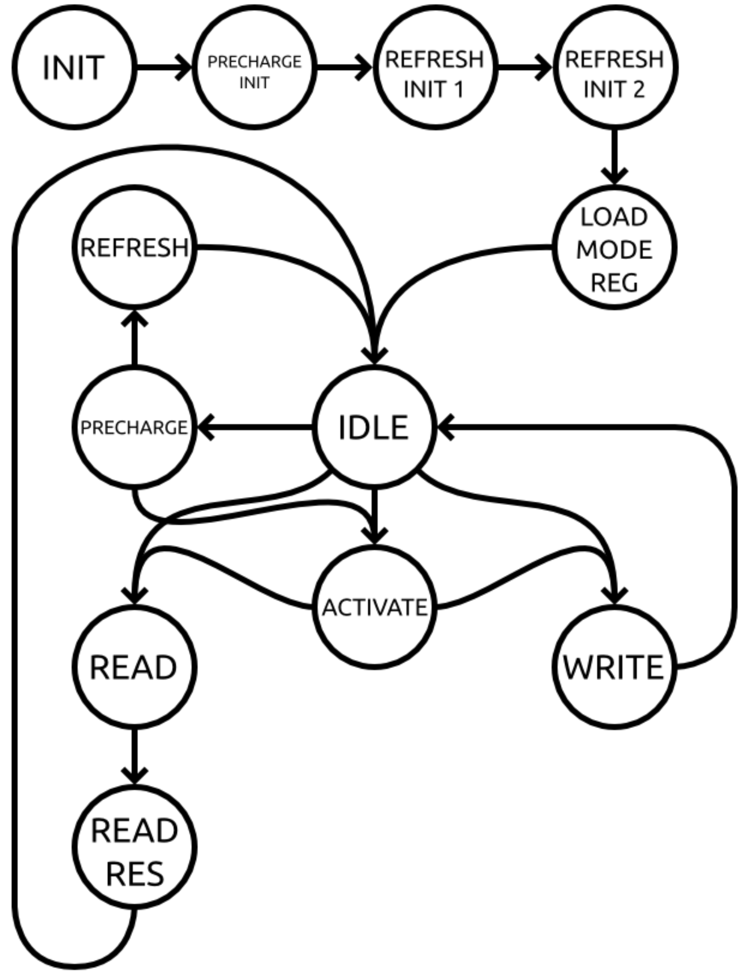
SDRAM Controller State Machine

- SDRAM can read/write words (columns) into a open row quickly
- SDRAM can also automatically read/write successive words without the need for a column address (burst operation)
- In order to read into another closed row, currently open row must be pre-charged to close



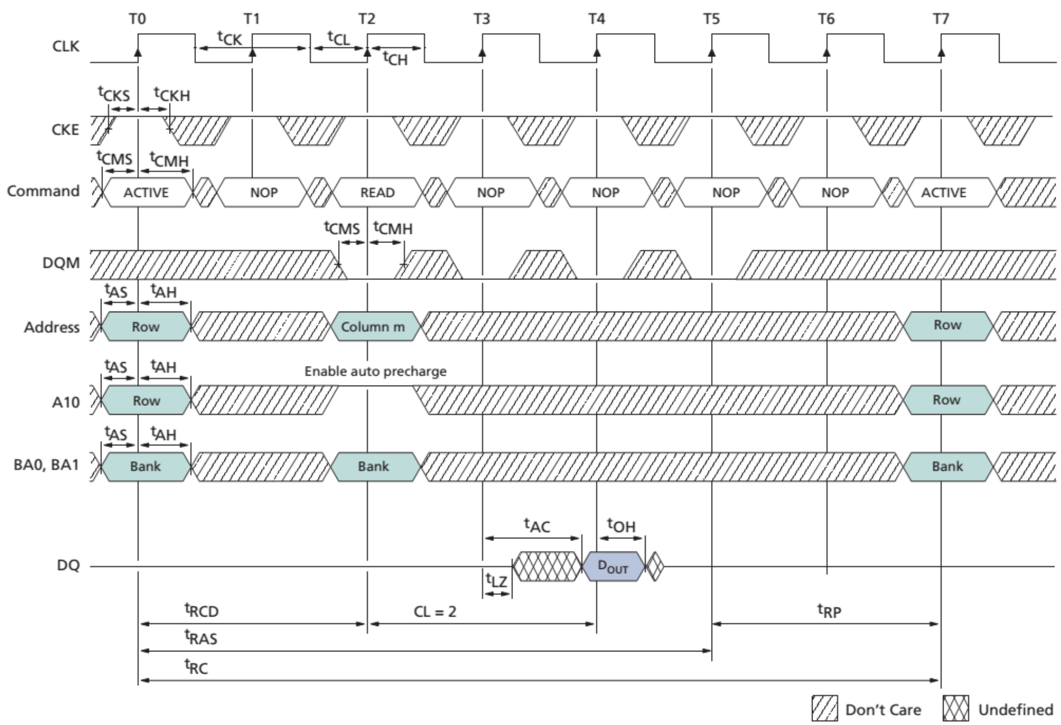
SDRAM Controller State Machine

- Controller must also manage refresh, which is handled per row
- Typically, each row needs to be refreshed every couple of milliseconds <100ms
- This means controller must keep track of when row was last refreshed and which row to refresh next



SDRAM Timing

- Because the SDRAM is synchronous, it requires a certain relationship between block and data to not violate setup and hold times
- That relationship is difficult to maintain when the SDRAM chip is physically off chip from FPGA (differences in PCB routing, FPGA I/O buffer delays, etc.)
- Use PLL (phase locked loop) to shift phase of the SDRAM's clock relative to data (settings from Experiment 7 tutorial were provided by the manufacturer)



Using SDRAM in FPGA Designs

- DE2-115 has 2*32Mx16 SDRAM chips wired in parallel
- Behaves like single 32Mx32 SDRAM
- You can use Altera's SDRAM controller if you export the Avalon MM port from Qsys
- Will need to place NIOS II memory somewhere else if this is the case, simple to use on-chip memory
  - Remember to make OCM sufficient size for your ELF file (~64KB should be enough)
  - Relocate the reset vector so your code executes out of OCM
  - Performance should be higher as well than execution from SDRAM
- Check out Avalon Interface Specifications to do this, you will need to create a Avalon MM master for your FPGA logic to use SDRAM

Design Trends in Size (gates of logic and datapath)

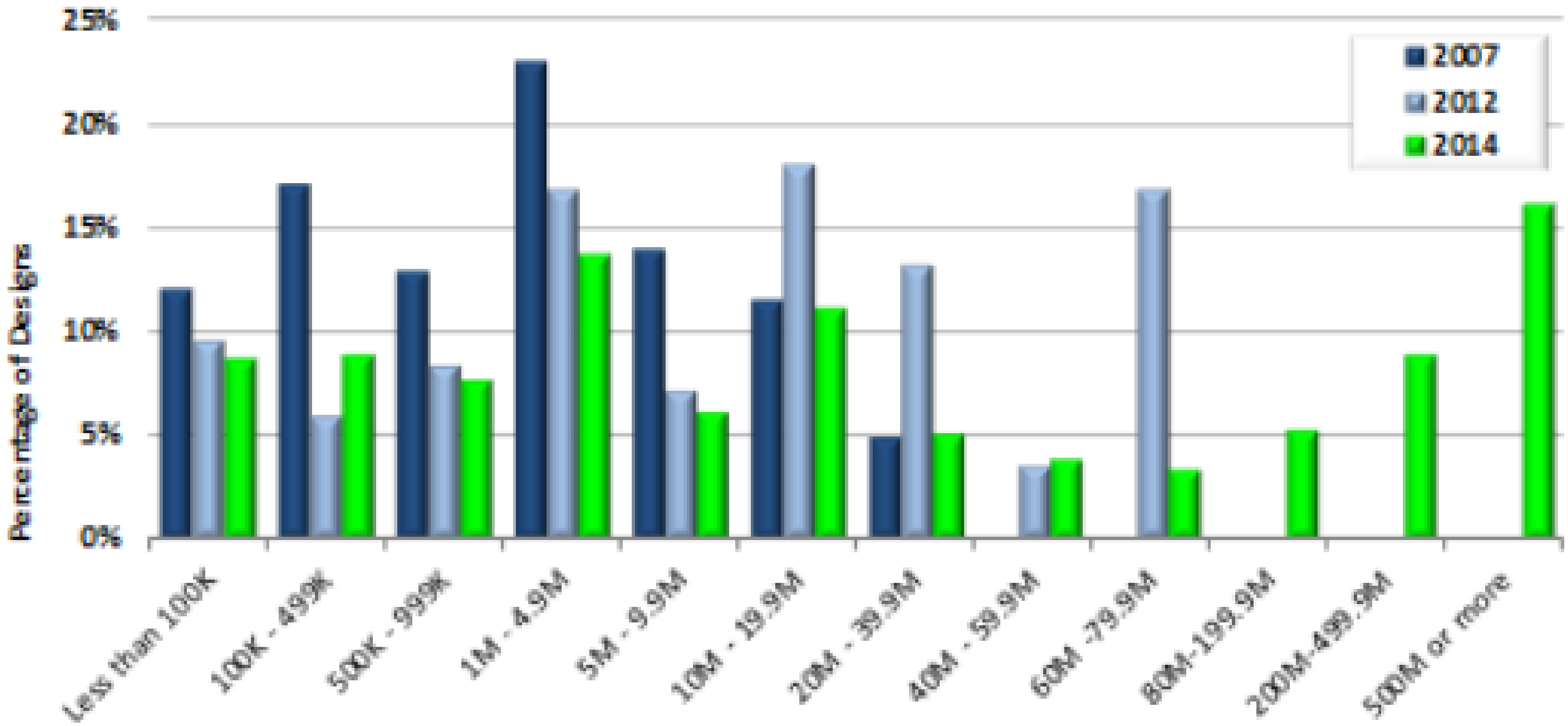


Figure 1. Design Sizes

H. Foster, Trends in functional verification: A 2014 industry Study, DAC'14

Time Spent in Verification

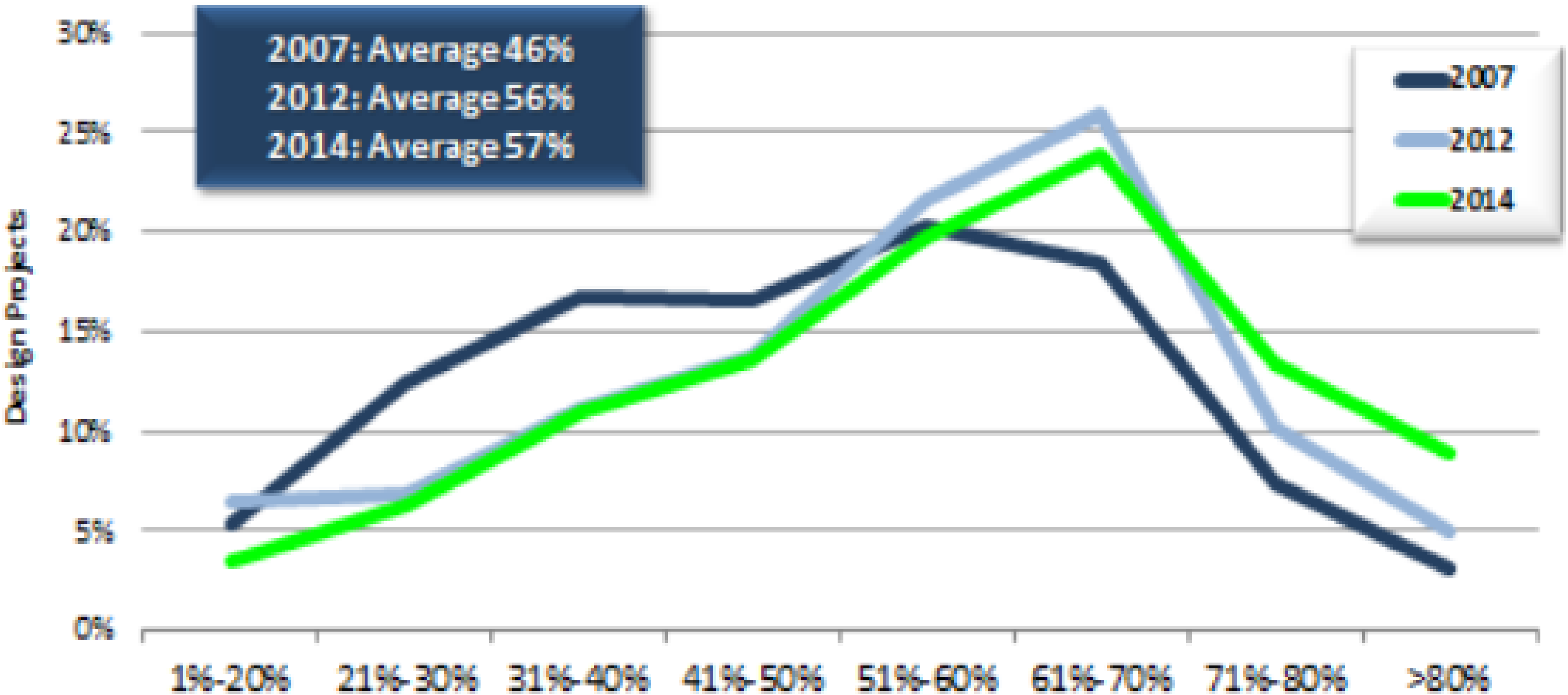


Figure 3. Percentage of Project Time Spent in Verification

More Verification Engineers!

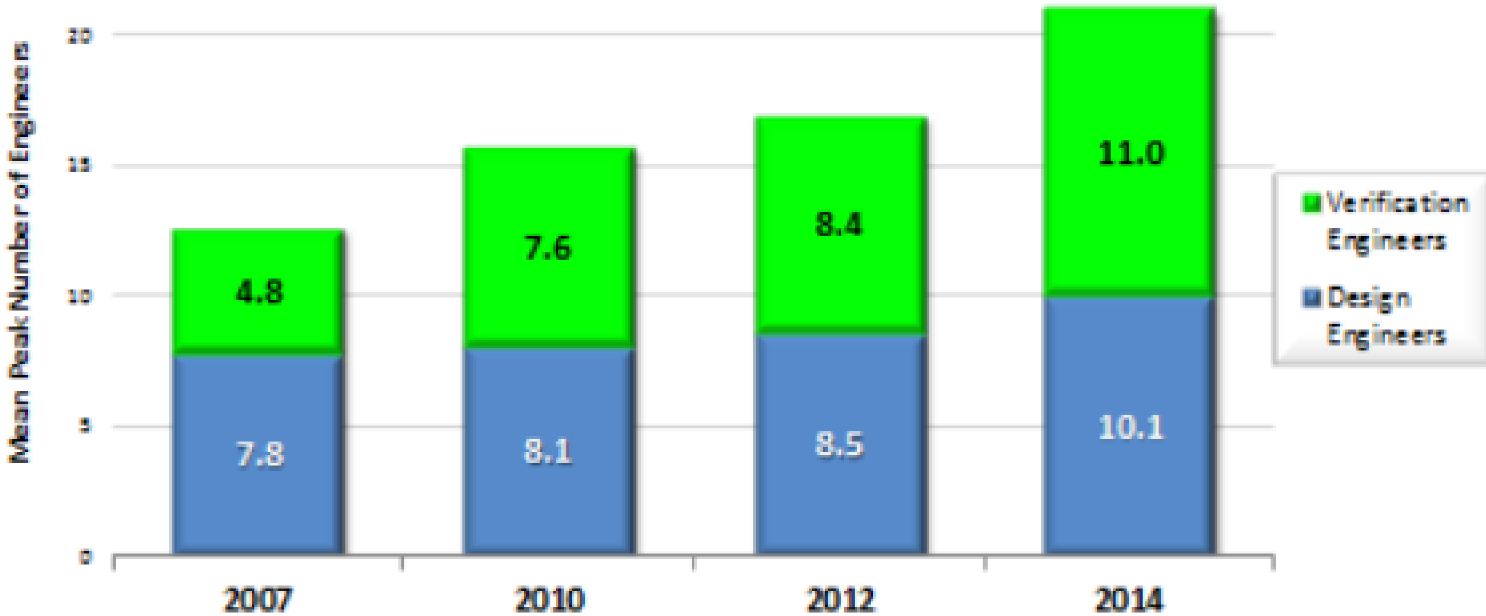


Figure 4. Mean Number of Peak Engineers per Project

Languages Used

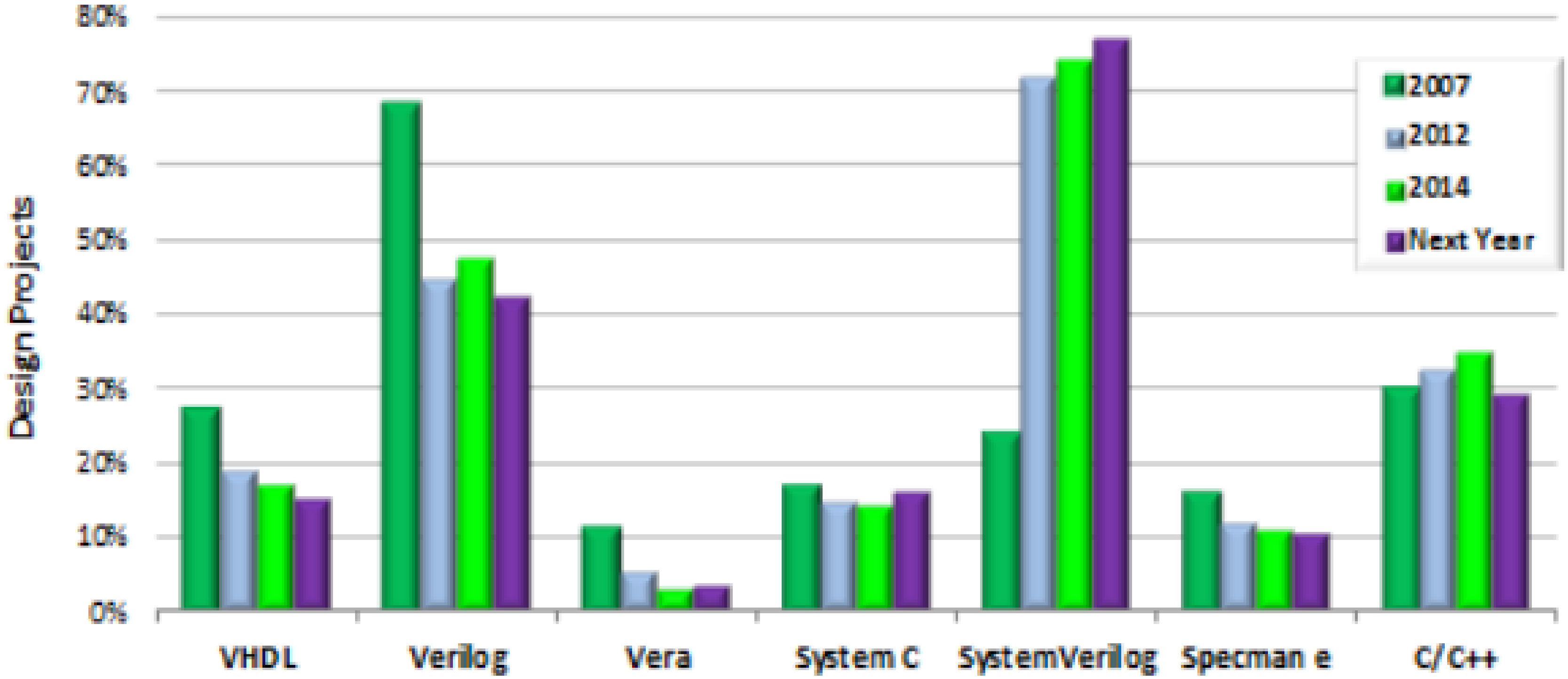


Figure 11. Languages Used for Verification (Testbenches)

How about Assertions?

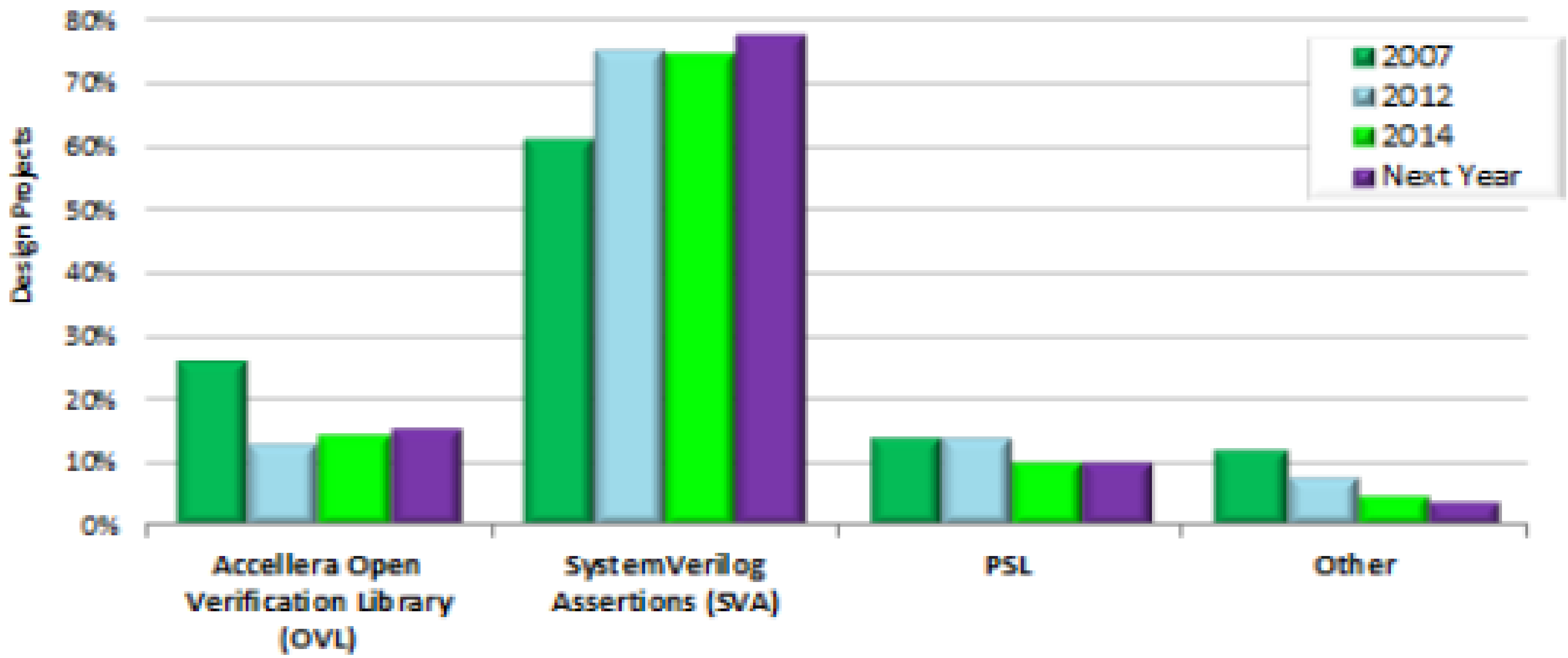


Figure 13. Assertion Language Adoption

Completion Schedule

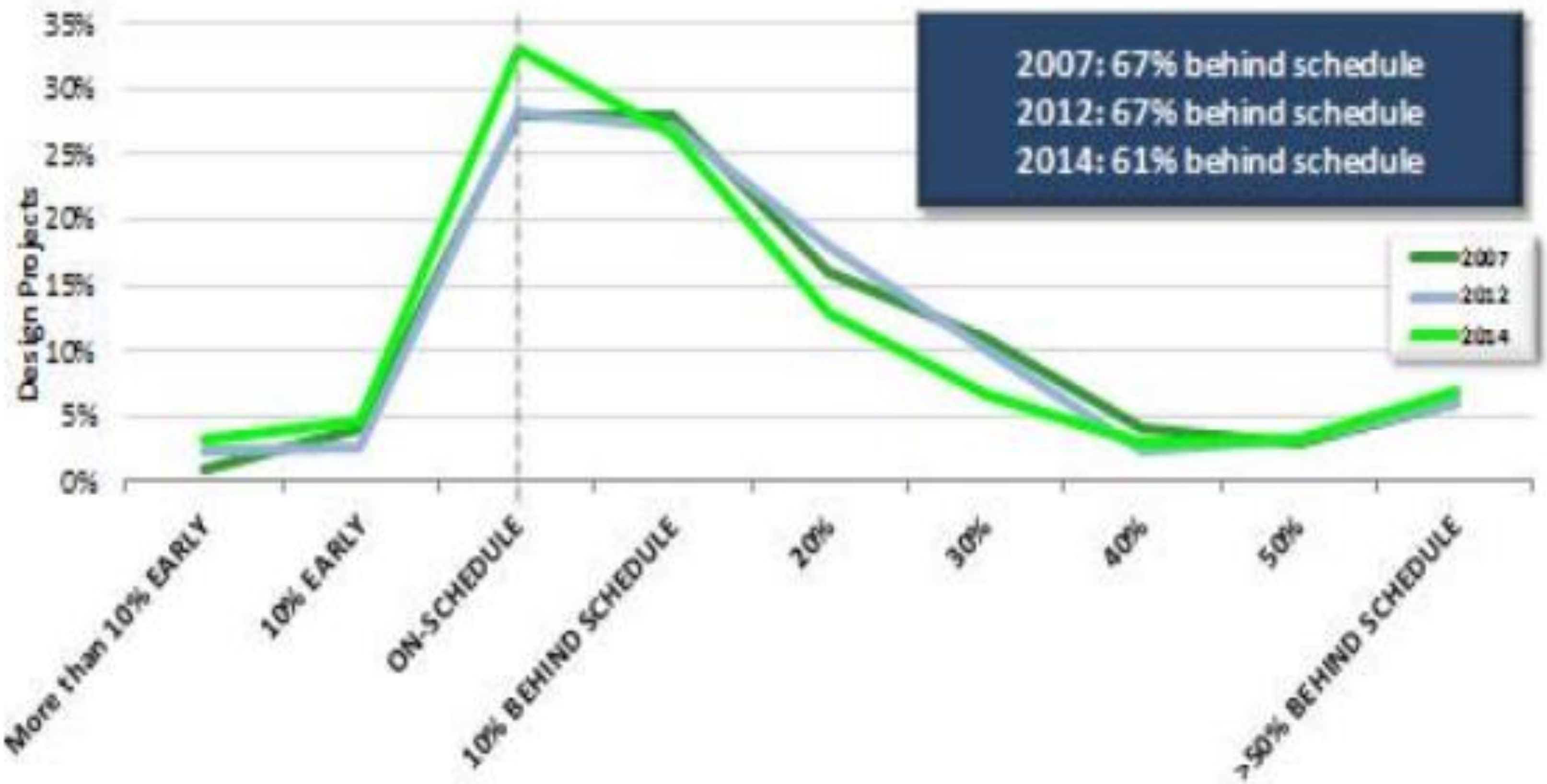


Figure 14. Design Completion Compared to Original Schedule

Number of Respins

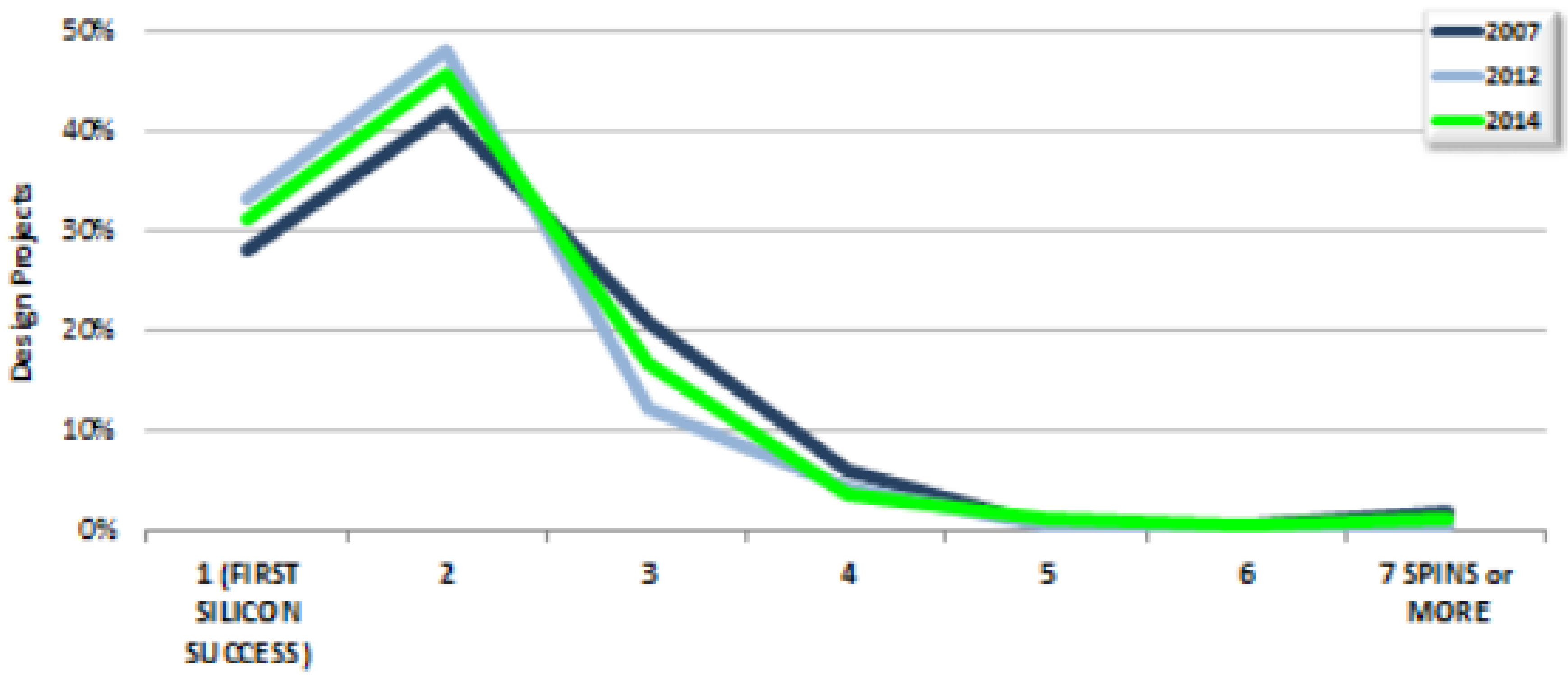


Figure 15. Required Number of Spins

Representative Flaws

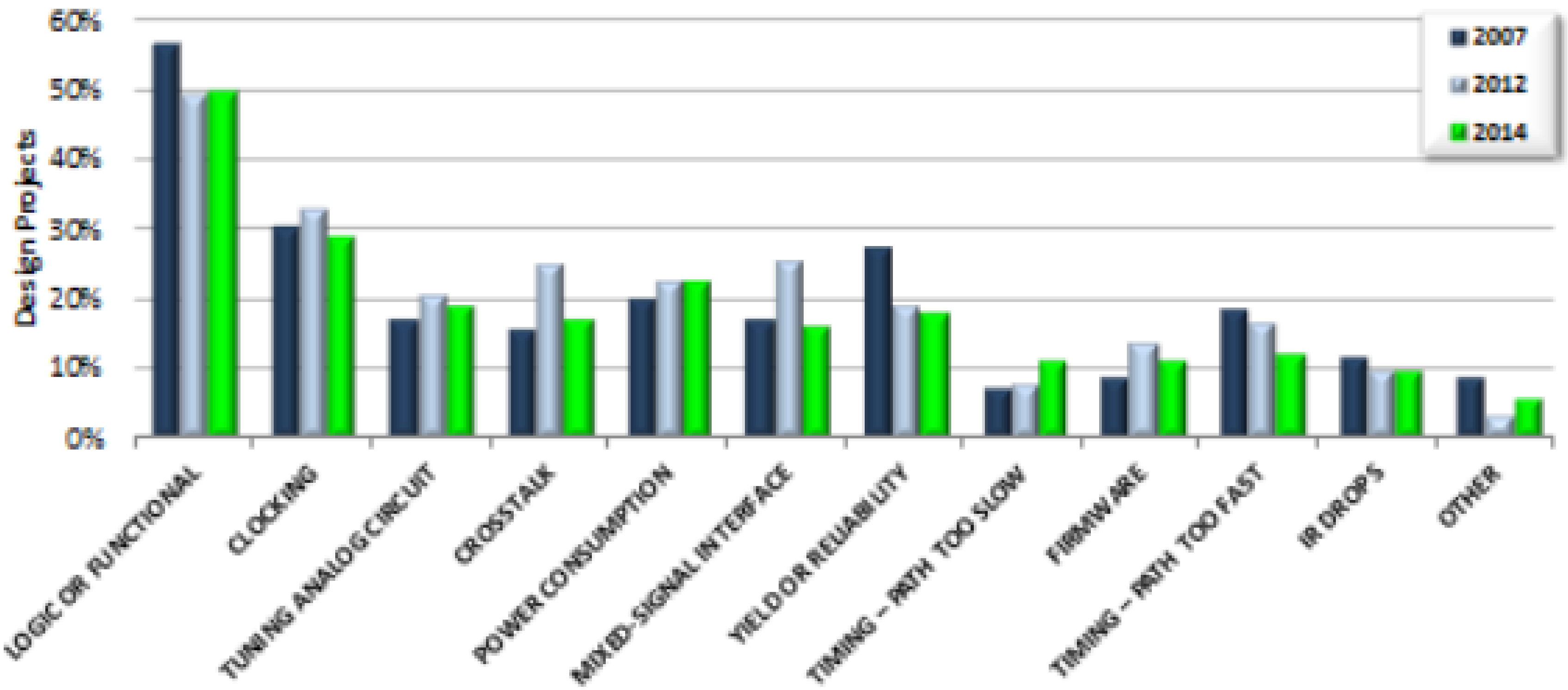


Figure 16. Types of Flaws Resulting in Respins

Root Cause of the Flaws

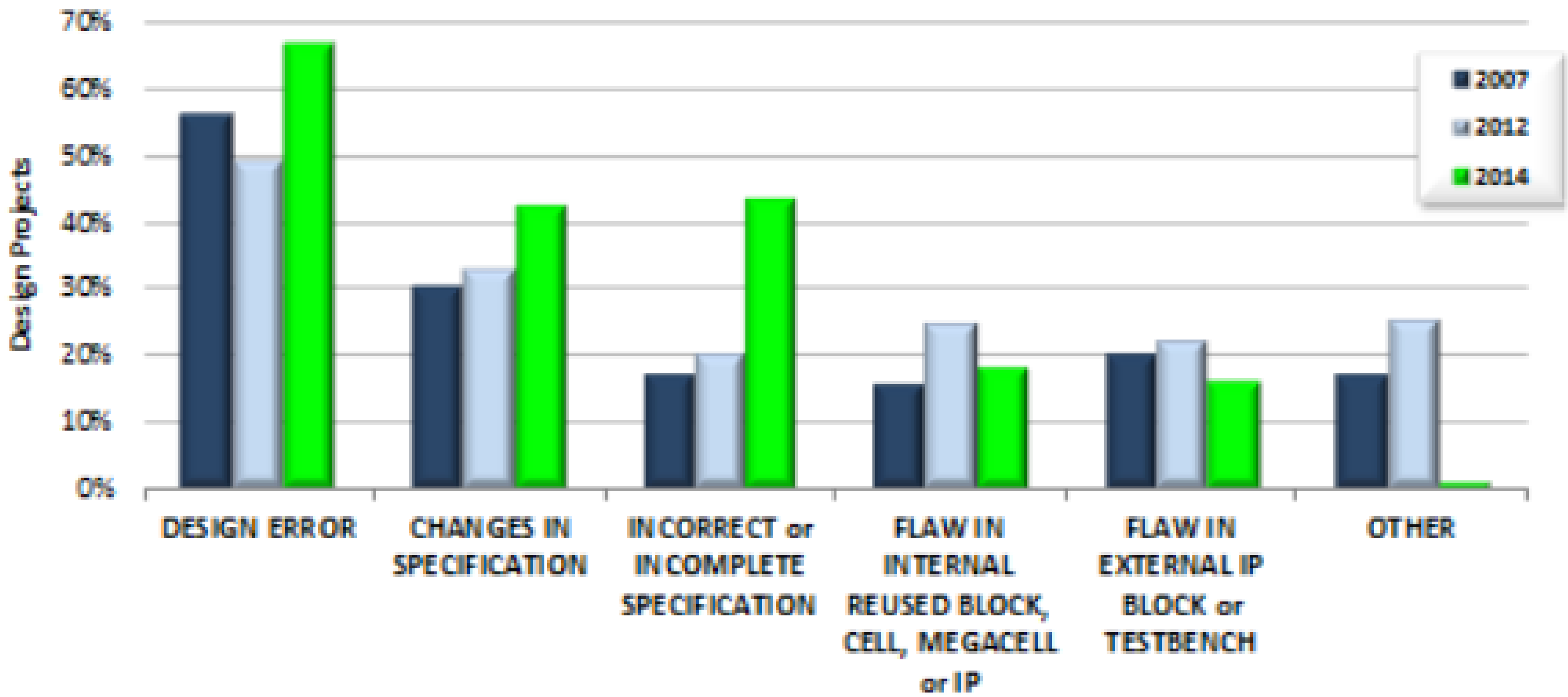
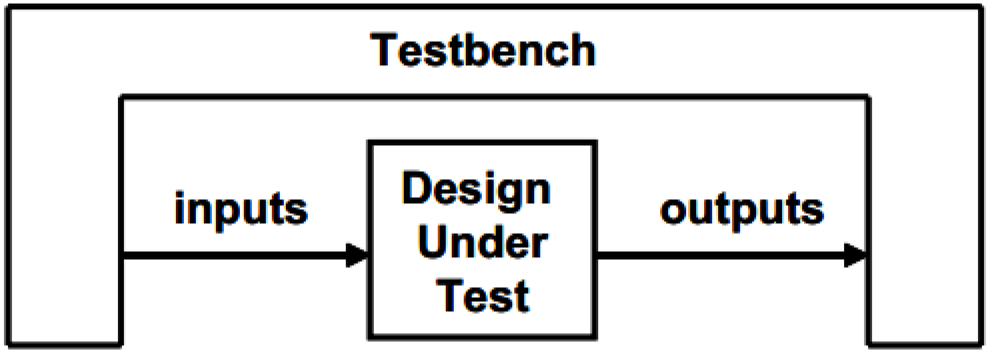


Figure 17. Root Cause of Functional Flaws

SystemVerilog Test-benches

- Test-bench is used to determine the correctness of the Design Under Test (DUT)
  - Generate stimulus
  - Apply stimulus to the DUT
  - Capture the response
  - Check for correctness
  - Measure progress against the overall verification goals
- Test-bench wraps around the DUT just as a hardware tester connects to a physical chip.



[1] C. Spear, SystemVerilog for Verification. New York, NY: Springer, 2008.

Flat vs. Layered Test-bench

- Flat test-bench

```
initial begin
    Reset = 0;
    LoadA = 1;

    #2 Reset = 1;

    #2 LoadA = 0;
    #2 LoadA = 1;
    ……
end
```

Like writing a circuit in a single module.

- Layered test-bench
  - Break down into tasks
  - Modularized
  - Scalable

Example Flat Test-bench (Register File)

```
module RegisterFileTest();

timeunit 10ns;// Half clock cycle at 50 MHz
             // This is the amount of time represented by #1

timeprecision 1ns;

// Internal signals
logic Clk = 0;
logic [2:0] SR1, SR2, DR;
logic Reset, LD_REG;
logic [15:0] D, SR1_OUT, SR2_OUT;

// A counter to count the instances where simulation results
// do not match with expected results
integer ErrorCnt = 0;

// Instantiating the DUT
RegisterFile r0(.*);

// Toggle the clock
// #1 means wait for a delay of 1 timeunit
always #1 Clk = ~Clk;

initial begin
       Reset = 1;
       SR1 = 3'd0;
       SR2 = 3'd0;
       DR = 3'd0;
       LD_REG = 1'b0;
       D = 0;
       #2 Reset = 0;

       DR = 3'd0;
       D = -1;
       #2 LD_REG = 1'b1;
       #2 LD_REG = 1'b0;

       DR = 3'd1;
       D = -2;
       #2 LD_REG = 1'b1;
       #2 LD_REG = 1'b0;

       DR = 3'd2;
       D = -3;
       #2 LD_REG = 1'b1;
       #2 LD_REG = 1'b0;

       DR = 3'd3;
       D = -4;
       #2 LD_REG = 1'b1;
       #2 LD_REG = 1'b0;

       #2 SR1 = 3'd0; SR2 = 3'd1;
       #2 SR1 = 3'd2; SR2 = 3'd3;
       #2 SR1 = 3'd4; SR2 = 3'd5;
       #2 SR1 = 3'd6; SR2 = 3'd7;
       #2 SR1 = 3'd0; SR2 = 3'd0;

       if (ErrorCnt == 0)
              $display("Success!");
       else
              $display("Try again!");
       $stop;
end
endmodule
```

Program block

- Analogous to the role of (top level) modules in synthesizable implementation
- Encapsulates all test-bench related items
- Creates a clear separation between design and verification
- Provides an entry and exit point to the execution of test-benches
- Acts as a scope for data defined within the block
- Provides syntactic context that specifies scheduling in the reactive region (executing the test-bench code)

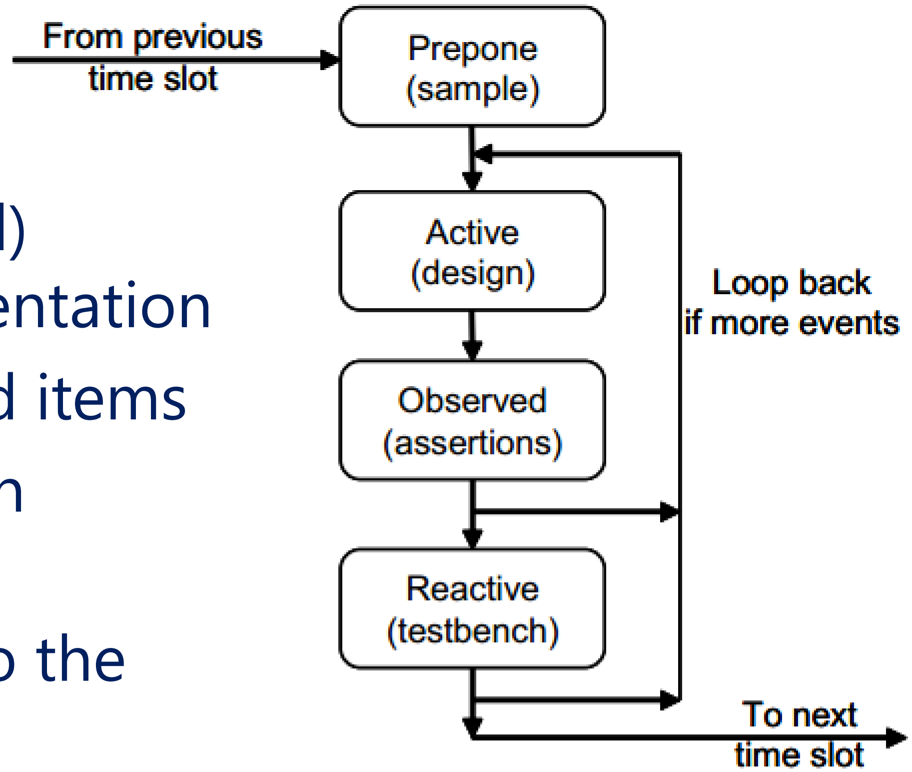


Figure: single time step

Delay Token (#)

- Delay token (#) is used to model delays in simulation
- Delay token is not synthesizable (it is simply ignored by synthesis tool, will still successfully synthesize)
- Important to understand simulation workflow
- **Note: different procedures (always_comb, always_ff, always) run concurrently in unknown order!**
- Each simulation timeunit (is divided into several queues)
- First queue: (can happen in any order, but always before the second queue)
  - All RHS of non-blocking assignments are evaluated
  - All RHS and LHS of blocking and continuous (assign procedure) assignment
  - Inputs and outputs evaluated
  - $display and $write processed
- Second queue: (happens after first queue)
  - Change LHS of all non-blocking assignment

- See (Understanding Verilog Blocking Understanding Verilog Blocking and Non and Non -blocking Assignments – S. Sutherland)

When do the assignments happen?

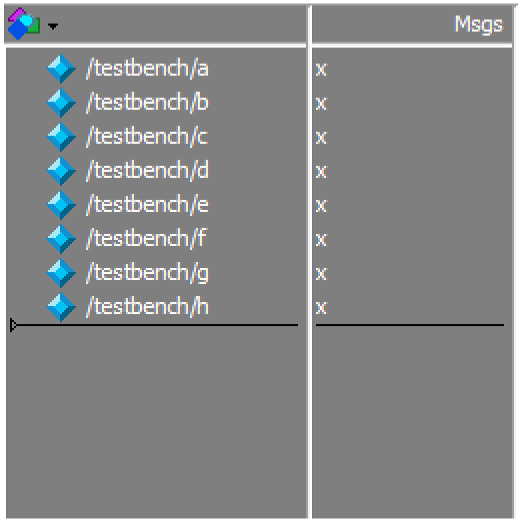
- Which one is different, and why?

```
initial begin: TEST_A
    #10 a = 1'b1;
    #20 b = 1'b0;
end

initial begin: TEST_B
    #10 c <= 1'b1;
    #20 d <= 1'b0;
end

initial begin: TEST_C
    e = #10 1'b1;
    f = #20 1'b0;
end

initial begin: TEST_D
    g <= #10 1'b1;
    h <= #20 1'b0;
end
```



Monitoring Internal Signals

- Sometimes useful to be able to monitor/force internal signals
- Prevents the need to break out “test signals” from interior modules
- Hierarchical references are addressed using . (period)

```
logic ALU_out;
always_comb begin: INTERNAL_MONITORING
    ALU_out = processor0.F_A_B;
end

initial begin: INTERNAL_FORCES
...
#1  force processor0.F_A_B = 1'b0;
```

Procedural Statements

- Many operators and statements in SystemVerilog are similar to C and C++. They are the non-synthesizable part of the language.

```
initial                                 // Procedural block
   begin : example
   integer array[10], sum, j;
   // Declare i in for statement
   for (int i=0; i<10; i++)             // Increment i
      array[i] = i;
   // Add up values in the array
   sum = array[9];
   j=8;
   do                                   // do...while loop
      sum += array[j];                  // Accumulate
   while (j--);                         // Test if j=0
   $display("Sum=%4d", sum);            // %4d - specify width
end : example                           // End label
```

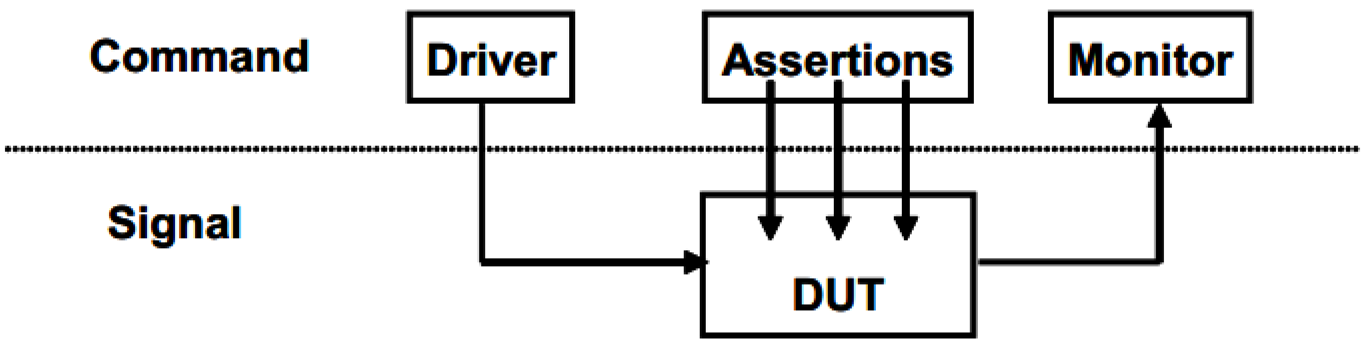
Tasks and Functions

- Tasks and functions can be declared inside classes or standalone
- Tasks can consume time, functions cannot. That is, things like ‘**delay’**, ‘**#100’** (time units), blocking statements (**posedge** clk, **wait**), can only exist in tasks for simulation
- You can pass an argument by reference using type ‘**ref**’ instead of copying its value. You can use ‘**return**’ to control the flow. (Non-synthesizable).
- Functions return value, while void functions do not

```
task load_array(int len, ref int array[] a);    // pass by reference
   if (len <= 0) begin
      $display("Bad len");
      return;                                   // return here
   end else begin
      int sum = 0;
      for (int i=0; i<a.size; i++)
        sum += a[i];
      $display("The sum of the arrays is ", sum);
   end
endtask
```

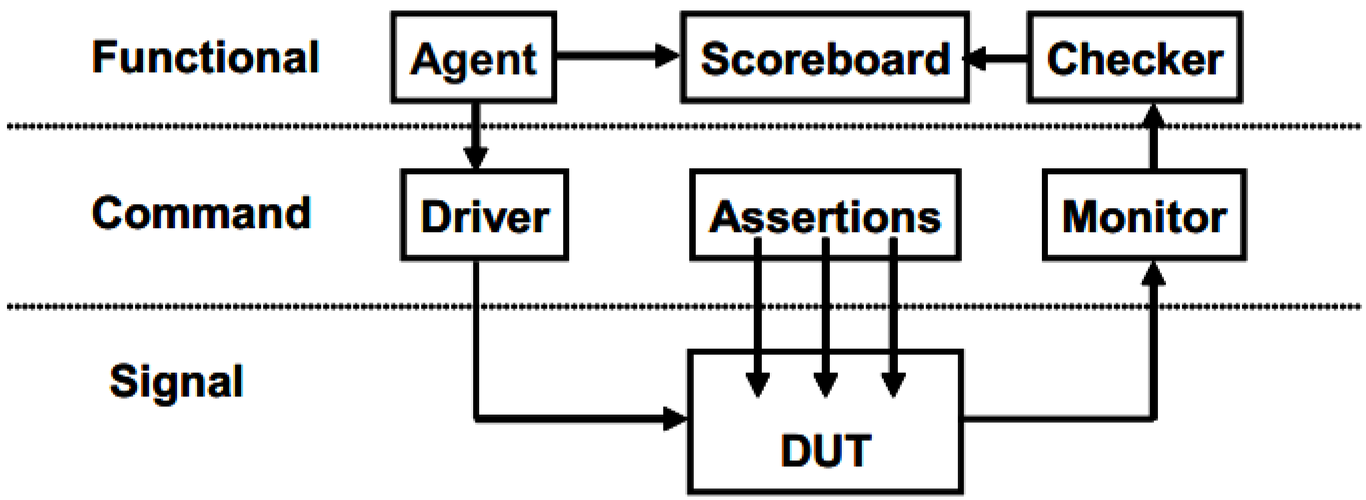
VMM (Verification Methodological Manual)

- Verification methodological manual is a collection of guidelines to write predictable and reusable verification suites
- Split up into multiple layers – signal, command, functional…
- Command layer (Level 1)
  - **Driver** – runs single commands such as bus read or write
  - **Monitor** – takes signal transitions and groups them together into commands
  - **Assertions** – Look at individual signals for any change across an entire command



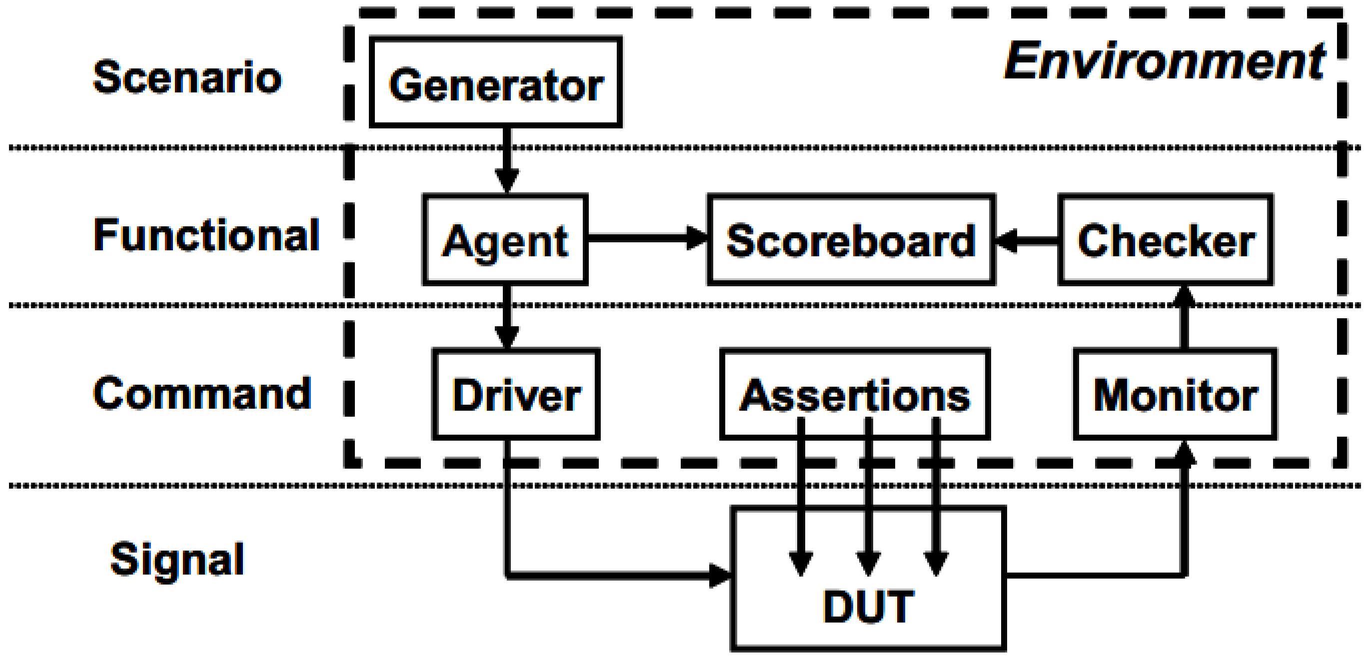
Functional – Level 2

- Functional Layer
  - **Agent** – receives high-level transactions such as DMA read or write and breaks them into individual commands
  - **Scoreboard** – predicts the results of the transaction
  - **Checker** – compares the commands from the monitor with those in scoreboard



Scenario – Level 3

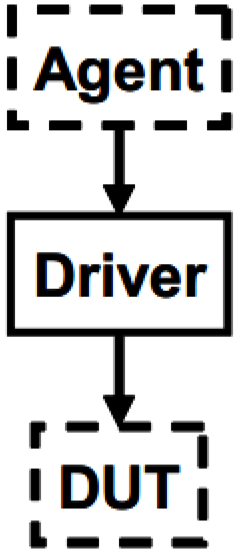
- Scenario Layer
  - **Generator** – generates a scenario for the simulation (e.g. play music from storage; download new music from a host; respond to input from the user, such as volume and track controls).



Simple Test-bench Driver

- Driver receives commands from the agent, then breaks the command down into individual signal changes

```
task run();
    done = 0;
    while (!done) begin
        // Get the next transaction
        // Make transformations
        // Send out transactions
    end
endtask
```



Connecting Everything Together

```
module top;
 bit clk;
 always #5 clk = ~clk;

 arb_if arbif(clk);
 arb a1 (arbif);
 test t1(arbif);
endmodule : top
```

Top level module

```
interface arb_if(input bit
clk);
   ...
endinterface: arb_if
```

Interface (data wrap)

```
module arb (arb_if arbif);
   ...
endmodule
```

Arbiter (circuit)

```
program test (arb_if arbif);
   ...
endprogram : test
```

Testbench

SystemVerilog Assertions

- An assertion validates the behavior in the circuit
- Check the intent of the design is met over simulation time
- Asks "Is the circuit working correctly?"
- Can be specified by both the designer (in the circuit module) or the verification engineer (in the testbench)
- Largely reduces the complexity of the testbench
- Two types of assertions: **Immediate** assertion and **Concurrent** assertion
- Four levels of reporting levels:

  `$fatal` – Runtime fatal. Simulation terminated

  `$error` – Runtime error (default). Simulation continues

  `$warning` – Runtime warning . Simulation continues

  `$info` – No severity. Simulation continues

Syntax and terminology

- Cycle delay – number of clock cycles to wait before the next expression is evaluated

  `##`*n* – a fixed number of clock cycles

  `##[`*min*`:`*max*`]` – a range of clock cycles

  `##[min:$]` – next expression must hold true eventually
- Implication operators – evaluation of a sequence
  - `|->` - Overlapped implication operator

    If the condition is true, sequence evaluation starts at the same clock tick
  - `|=>` - Non-overlapped implication operator

  If the condition is true, sequence evaluation starts at the next clock tick
- System functions –detect if a value changed between two adjacent clock ticks
  - `$rose` – returns true if the LSB of the expression changed to 1
  - `$fell` – returns true if the LSB of the expression changed to 0
  - `$stable` – returns true if the value of the expression did not change

Syntax and terminology II

- Repetition operators– arbitrary number of clock cycles a signal has to stay at a certain value
  - Consecutive repetition

    e.g. *"after 'start' raises, 'a' has to stay high for 3 continuous cycles before 'stop' is high"*

```
    @ (posedge clk) $rose(start) |-> ##1 a ##1 a ##1 a ##1 stop
=>  @ (posedge clk) $rose(start) |-> ##1 a[*3] stop
```

  - Go to repetition

    e.g. *"after 'start' raises, 'a' has to stay high for a* ***total*** *of 3 cycles before 'stop' is high, with the last cycle of 'a' happens right before 'stop'"*

```
@ (posedge clk) $rose(start) |-> ##1 a[->3] stop
```

Verilog assertion vs. SVA

```
// Sample Verilog checker
always @ {posedge a)
begin
repeat (1) @(posedge clk);
fork: a_to_b
   begin
   @ (posedge b)
   $display
   ("SUCCESS: b arrived in time\n",
$time);
   disable a_to_b;
   end

   begin
   repeat (3) @(posedge clk);
   $display
   ("ERROR:b did not arrive in time\n",
$time);
   disable a_to_b;
   end
join
end
```

```
// SVA Checker
a_to_b_chk:  assert property
©(posedge clk) $rose(a) |->
                  ##[1:3] $rose(b);
```

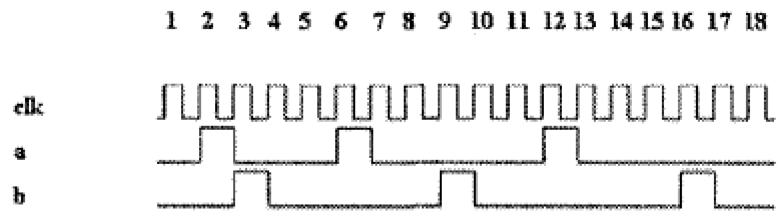


Figure 1: Waveform of the signals

```
SUCCESS: b arrived in time 127
vtosva.a_to_b_chk:
started at 125s succeeded at 175s

SUCCESS: b arrived in time 427
vtosva.a_to_b_chk:
started at 325s succeeded at 475s

ERROR: b did not arrive in time 775
vtosva.a_to_b_chk:
started at 625s failed at 775s
Offending '$rose(b)'
```

Figure 2: Simulation output

Immediate Assertion

- Evaluates immediately, not temporal in nature (no clocking)
- Test of an expression based on simulation events
- Used in **initial** and **always** procedural blocks
- Similar to an *if* statement
- Syntax:

  [name :] **assert** (expression) [pass_statement] [**else** fail_statement]

  (optional) (optional) (optional)

- e.g. "*Expects A always equal to B*"

```
always @(posedge clk)
   my_assert: assert (A == B)
      $display ("Good!")
   else
      $error ("Very bad!")
```

Concurrent Assertion I

- Evaluated at clock edges, test a sequence of events, temporal in nature
- Used in procedural blocks, modules, interfaces
- Specifies the expressions using *properties*

- *e.g. "Whenever a and b are both high, c should be high within 1 to 3 clock cycles"*

```
property p12;
   @ (posedge clk) (a && b) |-> ##[1:3] c;
endproperty

a12 : assert property(p12);
   $display ("Good!")
else
   $error ("Very bad!")
```

Concurrent Assertion II

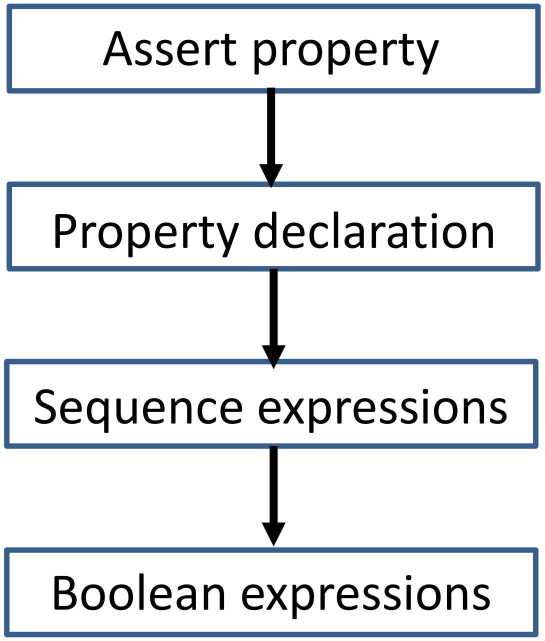
```
assert property(@ (posedge clk) (a && b) |-> ##[1:3] c;)
```

Boolean

Sequence

Property

Assertion




```
assertion_name : assert property
( name_of_property );

property name_of_property;
< test expression >;    or
< name_of_sequence >;
Endproperty

sequence name_of_sequence;
< test expression >;
endsequence

< test expression >;
```

Connecting SVA to the Design

- Use assertion checker in the module definition
- Or define a separate assertion checker module for reusability

```
module inline( input  logic        clk, a, b,
               input  logic [7:0] dl, d2,
               output logic [7:0] d ) ;
always @ (posedge clk)
begin
   if (a)
      d <= dl;
   if (b)
      d <= d2;
end
property p_mutex;
   @(posedge clk) not (a && b);
endproperty

a_mutex: assert property(p_mutex);

endmodule
```

Randomized Testing

- Verilog random functions
  - Easy to use in flat test-benches
  - Seed is optional and is usually only assigned in the first use
  - Same seed returns the same random sequence
- Example: test multiplier with different values

$random (seed) – 32-bit value uniform distribution

$urandom (seed) – 32-bit unsigned

$urandom_range (seed, min, max)

$dist_uniform (seed, start, end): returns an integer value; start < end.

$dist_normal (seed, mean, standard_deviation): returns a number that is on average approach to the mean argument. The standard_deviation is to determine shape of density

$dist_exponential (seed, mean): returns integer. $rdist_exponential returns a real value.

$dist_poisson (seed, mean)

$dist_chi_square (seed, degree_of_freedom)

$dist_t (seed, degree_of_freedom): degree_of_freedom determines the shape

$dist_erlang (seed, k_stage, mean)