Talk about
design & verification for an IP
- Simple Memory -

2014 - 2020

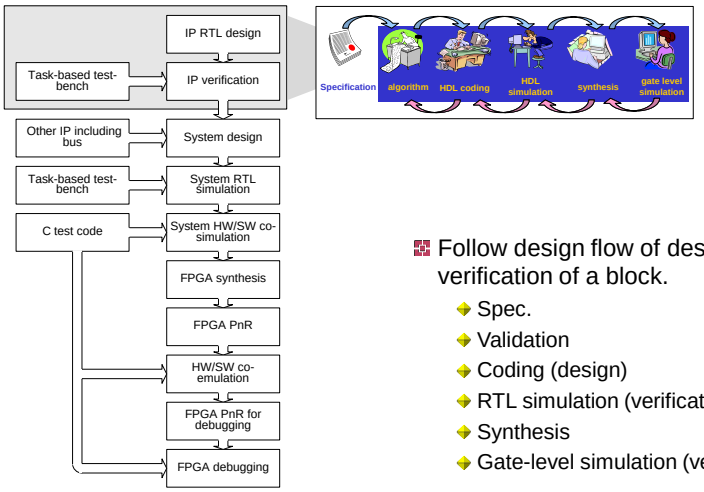
Ando Ki, Ph.D.
(adki@fugure-ds.com)

Agenda

- Walk through a complete design flow of a simple memory block.
  - Planning
  - Preparing
  - Verifying
- Contents
  - Design flow
  - Specification
  - Validation
  - Verification
  - Draw block and timing diagram
  - Design
  - Prepare test-bench
  - Simulation
  - Synthesis
  - Gate-level simulation
  - Projects

Design flow

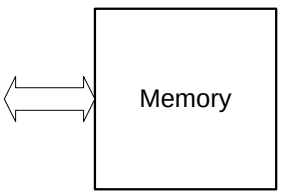
Design flows are the explicit combination of electronic design automation tools to accomplish the design of an integrated circuit.



- Follow design flow of design and verification of a block.
  - Spec.
  - Validation
  - Coding (design)
  - RTL simulation (verification)
  - Synthesis
  - Gate-level simulation (verification)

Ontogeny repeats phylogeny.

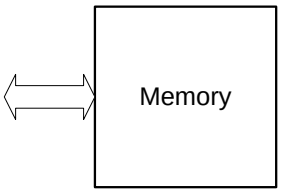
Specification: Know about the function to design



- What function to design
  - Memory
- Function 기능
  - storage: 저장기능
  - read: 읽기
  - write: 쓰기
- What to know or determine
  - data width → 32-bit
  - address width → 10-bit (1024 depth)
    - How many bytes?
  - access protocol (interface)
    - read protocol
    - write protocol
    - partial access

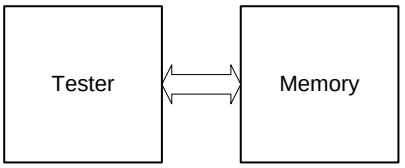
Quiz: Can you tell me the size of memory in bytes.

Validation: Is it possible?



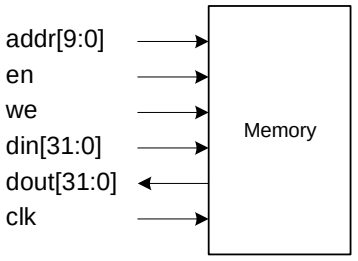
- Can I or we implement the function?
  - Time
    - 필요한 시점까지 만들 수 있나
  - Money
    - 설계하는 데까지 비용은 감당이 되나
  - Saleable
    - 결과물이 판매 가능한가
  - Skill
    - 설계할 수 있는 능력은 되는가
    - C, Verilog-HDL, VHDL, SystemC, ...
  - Practical
    - 설계가 가능한가
    - 1TByte on a chip?
    - 100GHz operation?
  - Environment
    - 설계에 필요한 환경은 있나
    - Simulator, Synthesizer
    - Fabrication
  - and so on

Verification: How to verify your design

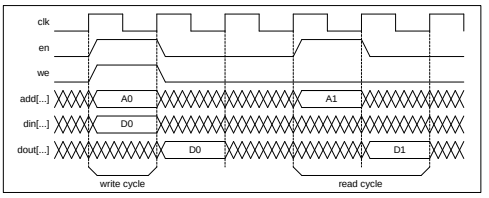


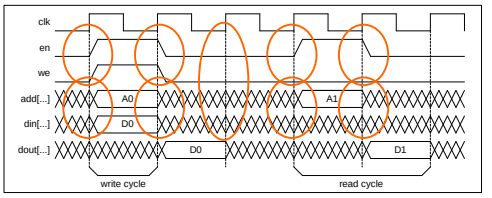
- Verification environment
  - Schematic diagram
- What to verify
  - remember the functions
    - read, write
    - partial or not
  - testing scenarios
    - what to check
    - how to check
    - compare actual value with expecting value
- How to verify the functions
  - RTL simulation
  - Post-synthesis simulation (gate-level simulation)
  - FPGA prototyping
  - Silicon proven

Draw block and timing diagram



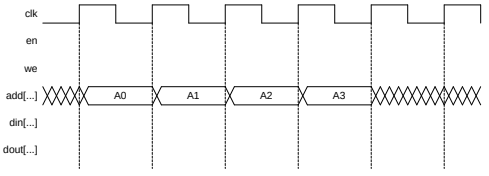
- Draw block diagram with ports
  - Note that 'addr[9:0]' is word-wise.
- Draw access timing diagram



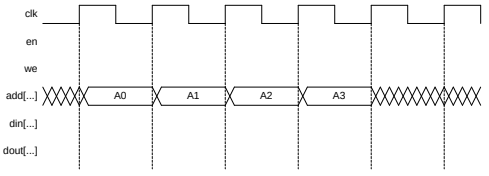


Draw timing diagrams

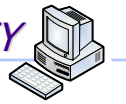
Consecutive four-read case



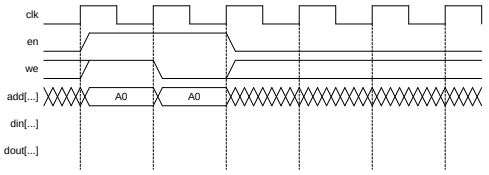
Consecutive four-write case



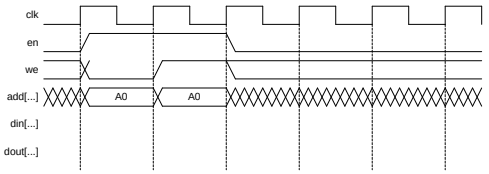
Draw timing diagrams



Read-after-write case

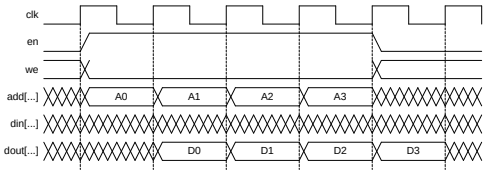


Write-after-read case

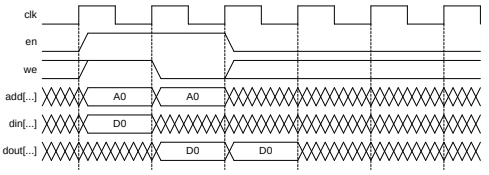


Draw timing diagrams

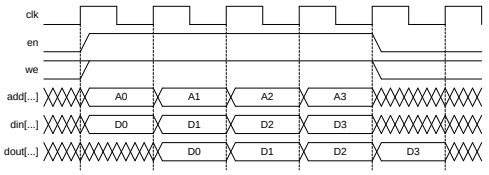
Consecutive four read case



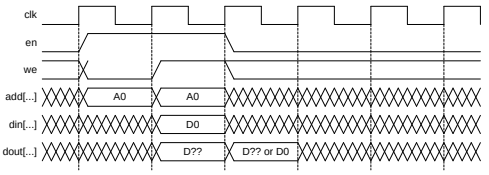
Read-after-write case



Consecutive four write case

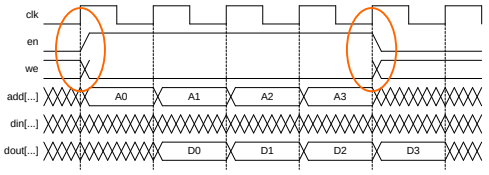


Write-after-read case

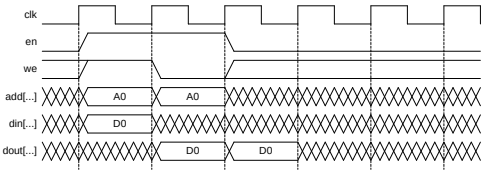


Draw timing diagrams

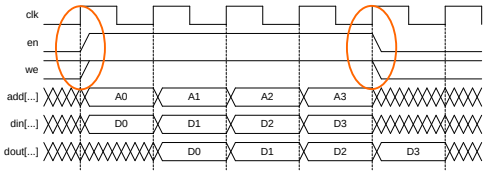
Consecutive four read case



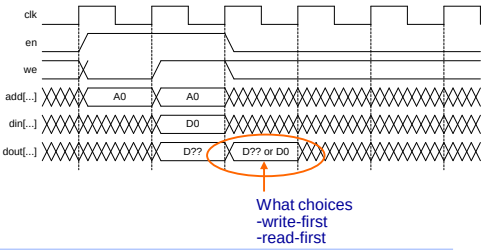
Read-after-write case



Consecutive four write case

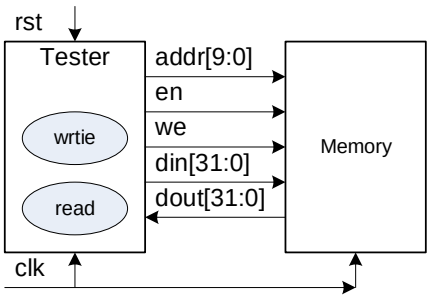


Write-after-read case



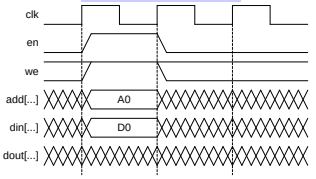
Tasks

DIY



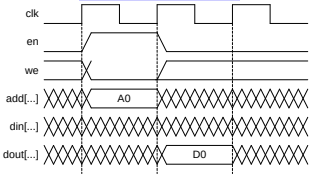

```
task write;
...
...
endtask
```

Write transaction



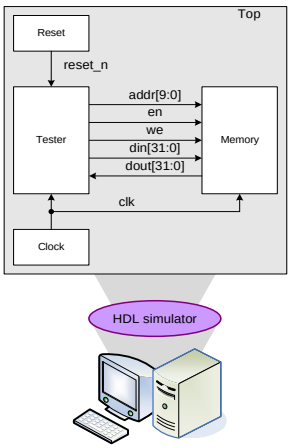

```
task read;
...
...
endtask
```

Read transaction

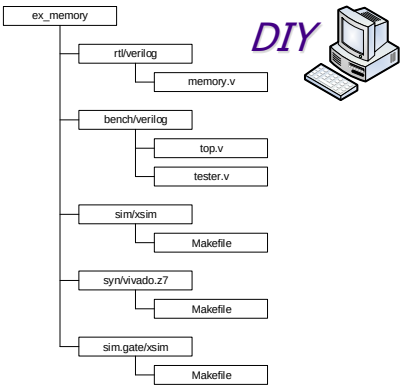


How to simulate your design

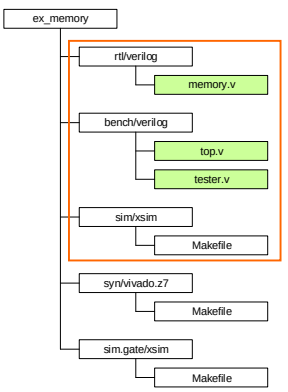
- Think about verification environment



- Plan directory structure

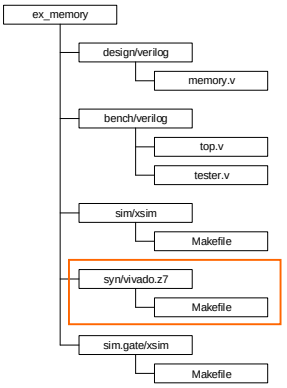


Design your memory block and verify it

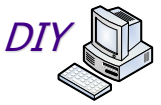


- Prepare Verilog files
  - 'memory.v'
  - 'tester.v'
    - make sure to use tasks
    - make sure to build test scenarios
  - 'top.v'
- Do verify your design using Vivado XSI
  - 'Makefile' for RTL simulation
- (Option) Do verify your design using ModelSim
  - 'Makefile' or 'RunMe.bat' or 'RunMe.sh' for RTL simulation.

Prepare gate-level design

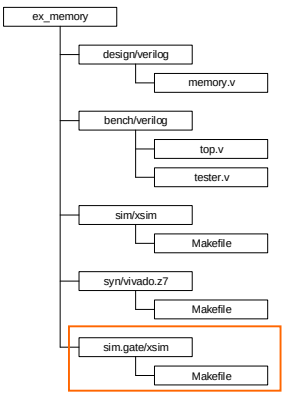


- Do logic synthesis of ‘memory.v’ using Vivado
  - ‘Makefile’
- Prepare gate-level Verilog of ‘memory.v’
  - by product of logic synthsis
  - ‘memory.vm’
- Do verify your gate-level design using ModelSim and Xilinx library
  - ‘Makefile’ or ‘RunMe.bat’ or ‘RunMe.sh’ for RTL simulation.
  - Note that Xilinx library is required
    - ‘$XILINX/verilog/src/unisims’

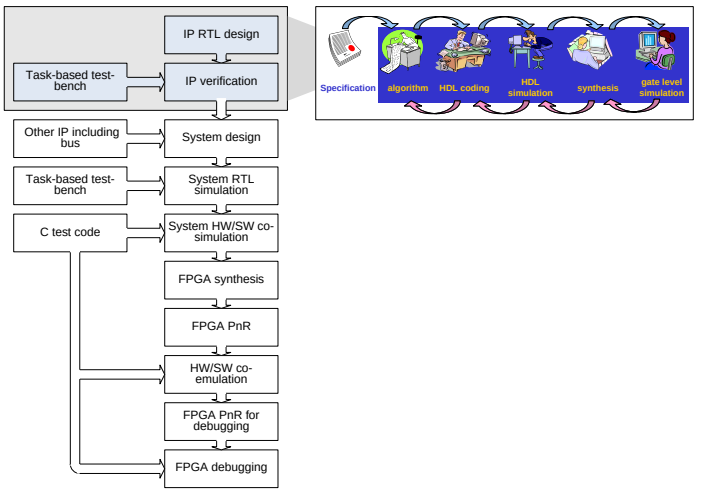


Gate-level simulation

- Do verify your gate-level design using XXIM and Xilinx library
  - ‘Makefile’

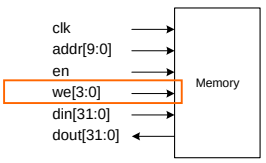


Summary

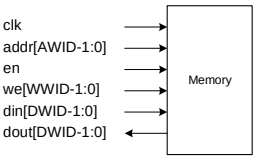


Project

- Partial access support
  - Modify the memory design to support partial access
    - E.g., one byte, two bytes, four bytes.
  - Verify your design with task-based RTL simulation

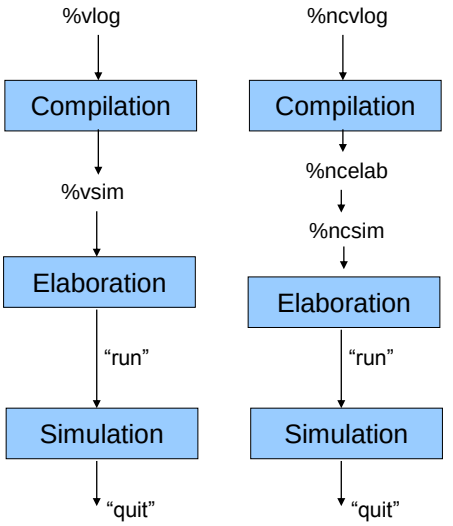


- Parameterized design
  - Modify the memory design to support various configuration using parameter
    - Memory depth, i.e., address width
    - Data width



Phases on HDL Simulator

- Compilation phase
  - Syntax checking
  - Lexical parsing
  - Building internal data structure
  - And so on.
- Elaboration phase
  - Preparing simulation
  - When the simulator starts, it first goes through an elaboration phase during which the **entire design is loaded** and **connected and initial values are set**.
  - During this phase all foreign **shared libraries (such as VPI) are loaded** and the **initialization functions of all foreign architectures are executed**.
- Simulation phase
  - The simulation phase begins when the first *run* command is executed and continues until a *quit* or *restart* command is executed.



ModelSim commands

- 'ModelSim Command Reference Manual' or 'Questa® SIM Command Reference Manual' at 'C:\ModelSim\questasim_10.3\docs\pdfdocs'
- 'vlib': This command creates a design library.

```
vlib -help
vlib [-short | -dos | -long | -unix] [-format { 1 | 3 | 4 }]
[-type {directory | archive | flat}]
[{-lock | -unlock} <design_unit>] [-locklib | -unlocklib]
[-unnamed_designs <value>]
<library_name>
```

- 'vlog': This command compiles Verilog source code into a specified working library (or to the **work** library by default).

```
vlog [options] <filename> [<filename> ...]
[options]
[+define+<macro_name>[=<macro_text>]] [-f <filename>]
[-work <library_name>]
```

- 'vsim': This command invokes the VSIM simulator

```
vsim [options]
[options]
[-c] [-do "<command_string>" | <macro_file_name>]
```

ISE commands

- Select ‘Start > Programs > Xilinx ISE Design Suite 12.4 > Documentation’

- XST User Guide for Virtex-4, Virtex-5, Spartan-3, and Newer CPLD Devices, UG627.
- ‘xst’: Xilinx Synthesis Technology (XST) is a Xilinx application that synthesizes Hardware Description Language (HDL) designs to create Xilinx specific netlist files called NGC files.

```
xst –ifn project_file -ofn log_file
```

- Command Line Tools User Guide, UG628
- ‘netgen’: NetGen is a command line executable that reads Xilinx design files as input, extracts data from the design files, and generates netlists that are used with supported third-party simulation, equivalence checking, and static timing analysis tools.

```
netgen -sim -ofmt [verilog|vhdl] [options] input_file[.ncd]
```


Xilinx libraries

- Library Guide
- refer to ‘http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_simulation_libraries.htm’
- Look at ‘$XILINX/verilog/src’ directory
  - UNISIM library for functional simulation of Xilinx primitives
  - UniMacro library for functional simulation of Xilinx macros
  - XilinxCoreLib library for functional simulation of Xilinx cores
  - Xilinx EDK library for behavioral simulation of Xilinx Embedded Development Kit (EDK) IP components
  - SIMPRIM library for timing simulation of Xilinx primitives
  - SmartModel/SecureIP simulation library for both functional and timing simulation of Xilinx Hard-IP, such as PPC, PCIe®, GT, and TEMAC IP.

How to use Xilinx library with ModelSim

- Use following options with vlog (ModelSim Verilog compiler).

```
+libext+.v
-y $XILINX/verilog/src
-y $XILINX/verilog/src/unisims
-y $XILINX/verilog/src/XilinxCoreLib
  $XILINX/verilog/src/glbl.v
```

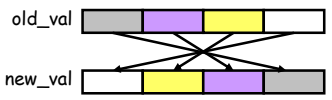

Task and function (1/2)

- Task
  - Declared within a module
  - Referenced only by a behavioral within the module
    - Can be referenced only from within a cyclic (always) or single-pass behavior (initial).
  - Parameters passed to task as inputs and inouts and from task a outputs or inputs
  - Local variables can be declared
  - Recursion not supported although nesting permitted
- Function
  - Implement combinational behavior
  - <u>No timing control</u>
    - '#' or '@' or 'wait' is not permitted
  - May call other functions with no recursion
  - Reference in an expression, e.g., RHS
  - No 'output' or 'inout' allowed
  - No non-blocking assignment
  - The purpose of a *function* is to respond to an input value by returning a single value.
    - A function can be used as an operand in an expression.
    - The value of that operand is the value returned by the function.

Task and function (2/2)

- Task
  - A task can contain time-controlling statements.
  - A task can enable other tasks and functions.
  - A task can have zero or more arguments of any type.
  - A task shall not return a value, since it creates a hierarchical organization of the procedural statements within a Verilog behavior.
- Function
  - A function shall execute in one simulation time unit.
  - A function cannot enable a task.
  - A function shall have at least one **input** type argument and shall not have an **output** or **inout** type argument.
  - A function shall return a single value, which can be scalar (single-bit) or vector (multi-bit).

Task and function usage example




```
// task is not allowed in continuous
// assignment

initial
  switch_byte_task(old_val, new_val);

always
  switch_byte_task(old_val, new_val);
```

```
// function can be used in continuous assignment
wire [31:0] new_val = switch_byte_function(old_val);


 initial
       new_val = switch_byte_function(old_val);

 always
       new_val <= switch_byte_function(old_val);
```

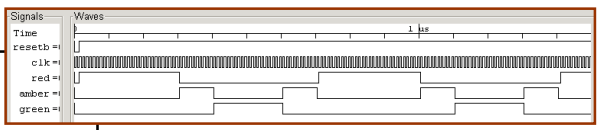
Form of task

```
module xxx (…);
…

task task_name;
   [input_output_inout_arguments;]
   statement;
endtask
…
endmodule
```

```
module xxx (…);
…

task task_name (in_out_inout_arguments);
   statement;
endtask
…
endmodule
```

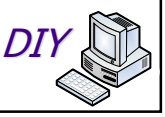

Task example

- ‘codes/verilog/task_function/traffic’




```
// traffic.v
module traffic_lights(clk, resetb);
  input clk, resetb;
  reg   red, amber, green;
  parameter on = 1, off = 0, red_tics = 30,
            amber_tics = 10, green_tics = 20;
  // initialize colors.
  initial red   = off;
  initial amber = off;
  initial green = off;
  always @ (clk or resetb) begin
     if (resetb==1'b1) begin
        red = on; // turn red light on
        light (red, red_tics); // and wait.
        amber = on; // turn amber light on
        light (amber, amber_tics); // and wait.
        green = on; // turn green light on
        light (green, green_tics); // and wait.
        amber = on; // turn amber light on
        light (amber, amber_tics); // and wait.
     end
  end
```

```
  // task to wait for 'tics' positive edge clocks
  // before turning 'color' light off.
  task light;
    output color;
    input [31:0] tics;
    begin
       repeat (tics) @ (posedge clk);
       color = off; // turn light off.
    end
  endtask
endmodule
```



Form of function

```
module xxx (...);
...

function function_name;
   input input_arg;
   [input_arguments;]
   begin
     [necessary_statements;]
     function_naem = expression;
   end
endfunction
...
endmodule
```

Default 1-bit output.

At least one input argument

A function definition shall include an assignment of the function result value to the internal variable that has the same name as the function name.

```
module xxx (...);
...

function [m:n] function_name;
   input input_arg;
   [input_arguments;]
   [necessary_statements;]
   function_name = expression;
endfunction
...
endmodule
```


Function example

- 'codes/verilog/task_function/factorial'

```
module top;

 function automatic integer factorial_auto;
   input [31:0] operand;
   integer i;
   if (operand >= 2) factorial_auto = factorial_auto(operand - 1) * operand;
   else factorial_auto = 1;
 endfunction

function integer factorial;
   input [31:0] operand;
   integer i;
   if (operand >= 2) factorial = factorial(operand - 1) * operand;
   else factorial = 1;
 endfunction

 integer result, n;
 initial begin
   for(n = 0; n <= 7; n = n+1) begin
      result = factorial_auto(n);
      $display("%0d factorial_auto=%0d", n, result);
      result = factorial(n);
      $display("%0d factorial=%0d", n, result);
   end
  end
endmodule
```

All items declared inside automatic functions are allocated dynamically for each invocation.
It is required to be enabled for recursive call.

DIY

```
# 0 factorial_auto=1
# 0 factorial=1
# 1 factorial_auto=1
# 1 factorial=1
# 2 factorial_auto=2
# 2 factorial=1
# 3 factorial_auto=6
# 3 factorial=1
# 4 factorial_auto=24
# 4 factorial=1
# 5 factorial_auto=120
# 5 factorial=1
# 6 factorial_auto=720
# 6 factorial=1
# 7 factorial_auto=5040
# 7 factorial=1
```

Verilog Parameter

- Parameters are constants, not variables.
  - Parameter represents constant since it cannot be modified at simulation time.
- Parameter can have default value.
- Parameter can be assigned by expression.
- Parameter can be modified in the module instance statement.
  - The order is important.

```
module my_memory
    #(parameter ADD_WIDTH=1, DATA_WIDTH=8, DELAY=0)
(
      input  wire [ADD_WIDTH-1:0]  addr
    , input  wire [DATA_WIDTH-1:0] datai
    , output wire [DATA_WIDTH-1:0] datao
    , input  wire                  rw
    , input  wire                  clk
);
//-----------------------------------------
 localparam MEM_DEPTH = 1<<ADD_WIDTH;
 reg [DATA_WIDTH-1:0] mem[0:MEM_DEPTH-1];
 always @ (posedge clk) begin
   if (rw)  #(DELAY) datao = mem[addr];
   else                mem[addr] = datai;
 end
 endmodule
//-----------------------------------------
 module top;
...
 my_memory          UmemA (addA, datA, rw, clk);
 my_memory #(3,4)   UmemB (addB, datB, rw, clk);
 my_memory #(3,4,2) UmemC (addC, datC, rw, clk);
 my_memory #(,,3)   UmemC (addD, datD, rw, clk);
...
 endmodule
```

Comma-separate list is possible.

Equation is possible

Default parameter

Parameter overriding

Verilog Parameter

- Parameters are constants, not variables.
  - Parameter represents constant since it cannot be modified at simulation time.
- Parameter can have default value.
- Parameter can be assigned by expression.
- Parameter can be modified in the module instance statement.
  - The order is important.

```
module my_memory(addr, datai, datao, rw, clk);
parameter ADD_WIDTH  = 1, DATA_WIDTH = 8;
parameter DELAY=0;
input   [ADD_WIDTH-1:0]   addr;
input   [DATA_WIDTH-1:0] datai;
output [DATA_WIDTH-1:0] datao;
input    rw;
input    clk;
//-----------------------------------------
parameter MEM_DEPTH = 1<<ADD_WIDTH;
reg [DATA_WIDTH-1:0] mem[0:MEM_DEPTH-1];
always @ (posedge clk) begin
  if (rw)  #(DELAY) datao = mem[addr];
  else                mem[addr] = datai;
end
endmodule
//-----------------------------------------
module top;
...
my_memory          UmemA (addA, datA, rw, clk);
my_memory #(3,4)   UmemB (addB, datB, rw, clk);
my_memory #(3,4,2) UmemC (addC, datC, rw, clk);
my_memory #(,,3)   UmemC (addD, datD, rw, clk);
...
endmodule
```

Comma-separate list is possible.

Equation is possible

Default parameter

Parameter overriding

Verilog Parameter: example (1/3)

- ‘code/verilog/module/parameter’ example

```
// mem_generic.v
`timescale 1ns/1ns
module mem_generic(add, datr, datw, en, we, clk, rstb);
  parameter ADD_WIDTH=8, DAT_WIDTH=8;
  parameter DELAY=0;
  //-----------------------------------------------
  input  [ADD_WIDTH-1:0] add;  wire [ADD_WIDTH-1:0] add;
  output [DAT_WIDTH-1:0] datr; reg  [DAT_WIDTH-1:0] datr;
  input  [DAT_WIDTH-1:0] datw; wire [DAT_WIDTH-1:0] datw;
  input                  en;   wire                 en;
  input                  we;   wire                 we;
  input                  clk;  wire                 clk;
  input                  rstb; wire                 rstb;
  //-----------------------------------------------
  localparam DEPTH = 1<<ADD_WIDTH;
  //-----------------------------------------------
  reg [DAT_WIDTH-1:0] mem[0:DEPTH-1];
  //-----------------------------------------------
  always @ (posedge clk or negedge rstb) begin
    if (rstb==1'b0) begin
      datr  <= {DAT_WIDTH{1'b0}};
    end else begin
      if (en==1'b1) begin
        if (we==1'b1) begin
          datr          <= #(DELAY) datw;
          mem[add] <= datw;
        end else begin
          datr     <= #(DELAY) mem[add];
        end
      end
    end
  end
endmodule
```

Generic memory model, where address width, data width and response time are specified by parameters

Width specified by parameters

Internal local parameter

Storage elements

Assignment delay specified by parameter.

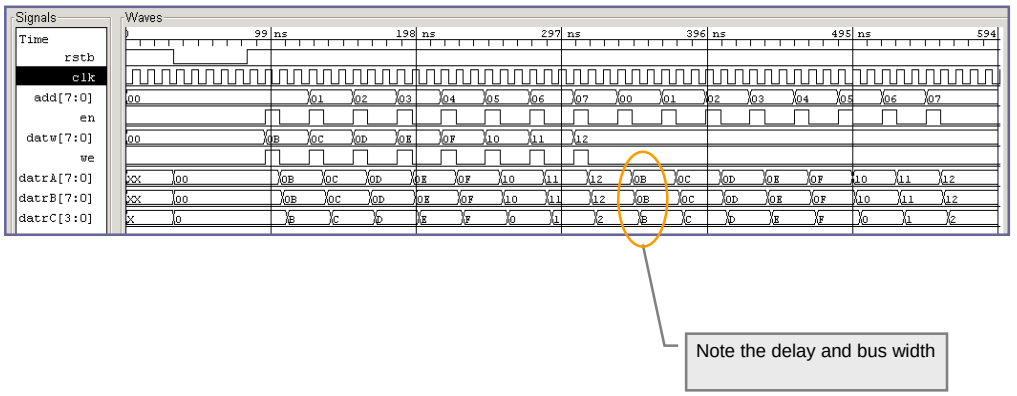
Verilog Parameter: example (2/3)

- ‘code/verilog/module/parameter’ example

```
// top.v
`timescale 1ns/1ns
module top;
  parameter AW=8, DW=8;
  reg  [AW-1:0] add;
  reg  [DW-1:0] datw;
  wire [DW-1:0] datrA, datrB;
  wire [3:0]        datrC;
  reg          en, we;
  reg          clk, rstb;
  integer x;
  //-----------------------------------------------
  mem_generic              UA (add, datrA, datw, en, we, clk, rstb);
  mem_generic #(8,8,2) UB (add, datrB, datw, en, we, clk, rstb);
  mem_generic #(5,4,5) UC (add[4:0], datrC[3:0], datw[3:0], en, we, clk, rstb);
  //-----------------------------------------------
  always @ (*) #5 clk <= ~clk;
  initial begin
     add  <= {AW{1'b0}};
     datw <= {DW{1'b0}};
     en   <= 1'b0; we   <= 1'b0;
     clk  <= 1'b0; rstb <= 1'b1;
     #33 rstb <= 1'b0;
     #50 rstb <= 1'b1;
  end
  //-----------------------------------------------
  .. details are not shown ..

endmodule
```

Different memory models are instantiated using parameter assignmemt.

Verilog Parameter: example (3/3)



Note the delay and bus width