The Verilog hardware description language

For native speakers of VHDL...
Hardware Description Languages

• **VHDL** = VHSIC Hardware Description Language
  
  (VHSIC = Very High Speed Integrated Circuits)
  
  • Developed by DOD from 1983 – based on ADA
  • Based on the ADA language

• **Verilog** – created in 1984 by Philip Moorby of Gateway Design Automation (merged with Cadence)
  
  • Based on the C language
  • IEEE P1800 “System Verilog” in voting stage & will be merged with 1364
Some Verilog properties

- Syntax similar to C (terse)
- Modeling at various levels
  - switch level, gate level, RTL, behavioral level
- More of a chip-level design language than system level (like VHDL)

Verilog:  
- TestFixture instead of TestBench (just terminology)
- Case sensitive
- always
- module

VHDL:  
- TestBench
- case insensitive (very important in names)
- process (with begin and end)
- entity and architecture combined
Verilog modules

The **module** is the basic building block, describing a logic block

```verilog
module module_name (module_terminal_list_separated_by_commas);
  port and net declarations (IO plus wires and regs for internal nodes)
  IO terminals: input, output, inout
  wire: internal “net” - combinational logic (needs a driver)
  reg: data storage element (holds a value – acts as a “variable”)
  parameter: assign an identifier to a constant

  functional description

endmodule
```
Verilog syntax issues

• Like VHDL, whitespaces include space, tab, and newline

• Comments use same format as C and C++:
  
  // this is a one line comment to the end of line
  /* this is another single line comment */
  /* this is a multiple line comment */

• Identifiers: any sequence of
  * letters (a-z, A-Z), digits (0-9), $ (dollar sign) and _ (underscore).
  * the first character must be a letter or underscore  
    Identifier_15

• Verilog is case sensitive, VHDL is case insensitive
  
  Bob, BOB, bob  // three different identifiers in Verilog
Combinational logic example

**module** small_block (a, b, c, o1, o2);

**input** a, b, c;

**output** o1, o2;

**wire** s;

**assign** o1 = s | c ; // OR operation

**assign** s = a & b ; // AND operation

**assign** o2 = s ^ c ; // XOR operation

**endmodule**

Analagous to VHDL entity

Analagous to VHDL architecture, signal assignments
Combining statements

wire a;
assign a = b | (c & d);

// Equivalent to:
wire a = b | (c & d);
2-to-1 MUX example

```vhdl
module MUX2 (A,B,S,Z);
input A,B,S; //input ports
output Z; //output port
always //evaluate block continuously
  begin
    if (S == 0) Z = A;
    else Z = B;
  end
endmodule
```

Analagous to VHDL process, sensitive to S,A,B changes
Hierarchy example of 4-to-1 MUX (calling the previous 2-to-1 MUX)

module MUX4 (A,B,c,d,S0,S1,Z);
  input A,B,c,d,S0,S1;
  output Z;
  wire z1,z2;
  MUX2 M1(A,B,s0,z1); //instance M1 of MUX2
  MUX2 M2(z1,z2,S1,Z); //instance M2 of MUX2
  MUX2 M3(.S(S0), .Z(z2).A(c),.B(d)); //connect signal-port by name
endmodule

Note: no component declaration, just instantiation
But: must define MUX2 module in Verilog source before MUX4 module
Multi-bit signals (vectors)

// an example of four 2-to-1 MUXs
module MUX2ARR(A,B,S,Z);
  input [3:0] A,B; // note whitespace before & after array declaration
  input S;
  output [3:0] Z; // little-endian form, MSB = bit 3 (left-most)
  wire [0:3] G; // big-endian form, MSB = bit 0 (left-most)
  always
    begin
      if (S == 0) G = A;
      else G = B;
    end
  assign Z = G;
endmodule

A,B,Z,G
analagous to VHDL
std_logic_vector
## Constants and identifiers

- **Numbers:** *(bit width)‘(radix)(digits)*

<table>
<thead>
<tr>
<th>Verilog</th>
<th>VHDL</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>4’b1010</td>
<td>“1010” or B“1010”</td>
<td>a 4-bit binary value</td>
</tr>
<tr>
<td>12’ha5c</td>
<td>X“0a5c”</td>
<td>a 12-bit hexadecimal value</td>
</tr>
<tr>
<td>6’o71</td>
<td>O“71”</td>
<td>a 6-bit octal value</td>
</tr>
<tr>
<td>8’d255</td>
<td>255</td>
<td>an 8-bit decimal value</td>
</tr>
<tr>
<td>255</td>
<td>255</td>
<td>32-bit decimal value (default)</td>
</tr>
<tr>
<td>16’bZ</td>
<td>x”ZZZZZ”</td>
<td>a 16-bit floating value</td>
</tr>
</tbody>
</table>

- **Logic values:** 0, 1, x, z *(note lowercase x and z, for undefined logic and tri-state values, respectively)*
Clocked circuits

@ (posedge CLK)  trigger on rising edge of CLK (0->1, 0->X, X->1)
@ (negedge CLK) trigger on falling edge of CLK (1->0, 1->X, X->0)
@ (CLK) sensitivity list – trigger on either edge of CLK

//Example: simple rising edge triggered flip-flop:
always @ (posedge CLK)
begin
    Q = D;
end

//Example: falling edge triggered flip-flop with sync preset and clock enable:
always @ (negedge CLK)
begin
    if (PR == 1) Q = 1; //synchronous set
    else if (CE == 1) Q = D; //clock enable
end

Analagous to VHDL process with CLK in sensitivity list
DFF example – with asynchronous reset

module dff (q,d,clk,reset)
    output q;
    input d,clk,reset;
    reg q; //"reg" since q stores the flip flop state
always @(posedge clk or posedge reset) //sensitive to clk or reset change
    if (reset) //load prevented if reset active
        q = 1'b0;
    else
        q = d'; //load if reset not active
endmodule
module dff (q,d,clk,reset)  
    output q; 
    input d,clk,reset; 
    reg q; //"reg" since q stores the flip flop state
    always @(posedge clk) //sensitive to rising edge of clk
        if (reset) //reset takes precedence on rising clock edge
            q = 1'b0; 
        else //load if reset not active
            q = d'; 
endmodule
D latch (level sensitive)

module d_latch (q,d,en)
    input d,en;
    output q;
    reg q; //q output holds a value
    always @(en or d) //execute whenever en or d change
        if (en) q = d;
endmodule
Operators (in increasing order of precedence):

|| logical OR
&& logical AND
| bitwise OR
^ bitwise XOR
& bitwise AND
== logical equality
!= logical inequality
< less than
<= less than or equal
also > greater than
>= greater than or equal
<< shift left
>> shift right
+ addition
- subtraction
* multiply
/ divide
% modulus
Unary operators:

Examples:

- `!` logical negation
- `~` bitwise negation
- `&` reduction AND
- `~&` reduction NAND
- `|` reduction OR
- `~&` reduction NOR
- `^` reduction XOR
- `~^` reduction XNOR

- `~4'b0101` is `4'b1010`
- `& 4'b1111` is `1'b1`
- `~& 4'b1111` is `1'b0`
- `| 4'b0000` is `1'b0`
- `~| 4'b0000` is `1'b1`
- `^ 4'b0101` is `1'b0`
- `~^4'b0101` is `1'b1`

*reduction operator is applied to bits of a vector, returning a one bit result*
Arithmetic operations

- Verilog recognizes standard arithmetic operators
- Synthesis tools will generate arithmetic circuits

```verilog
module Adder_8 (A, B, Z, Cin, Cout)
    input [7:0] A, B;
    input Cin;
    output [7:0] Z;
    output Cout;
    assign {Cout, Z} = A + B + Cin; //extra output bit for carry
endmodule
```
Optimizing circuits (1)

//synthesis tool infers two adders from the following
module Add1 (sel, a, b, c, d, y);
   input a, b, c, d, sel; output y; reg t1, t2, y;
   always @(sel or a or b or c or d) begin
      if (sel == 0)  y <= a + b;
      else               y <= c + d;
   end
endmodule
Optimizing circuits (2)

//synthesis tool infers a single adder from the following
//indicate that a mux selects adder inputs
module Add2 (sel, a, b, c, d, y);
    input a, b, c, d, sel; output y; reg t1, t2, y;
    always @(sel or a or b or c or d) begin
        if (sel == 0) begin t1 = a; t2 = b; end
        else begin t1 = c; t2 = d; end
        y = t1 + t2;
    end
endmodule
Verilog built-in primitive gates

• Verilog has 7 gate types that are primitive components: 
  and, or, nand, nor, xor, xnor, not

• Format:
  
gate GATE_INST_NAME (Z,I1,I2,...IN);  // the output comes first, followed by inputs

module carry_out(A,B,Cin,Cout)
  input A,B,Cin;
  output Cout;
  wire w1,w2,w3;
  and   A1  (w1,A,B); //primitive and gate instances
  and   A2  (w2,A,Cin);
  and   A3  (w3,B,Cin);
  or    O1  (Cout,w1,w2,w3); //primitive or gate instance
endmodule
Lists of assign/gate instance statements

• Can specify a comma-separated list of gates of one type
• Likewise for “assign” statements

module carry_out(A,B,Cin,Cout)
  input A,B,Cin;
  output Cout;
  wire w1,w2,w3,w4,w5;
  and    A1  (w1,A,B), // list of three and gate instances
          A2  (w2,A,Cin),
          A3  (w3,B,Cin);
  assign w4 = w1 & w2, // list of two assign statements
         Cout = w4 & w4;
endmodule
Memory models

- Memory is an array of registers

```vhdl
reg mem1bit [ 0:1023];       //array of bits
reg [7:0]   membyte [0:1023]  //array of bytes

mem1bit[511]  - refers to one bit of memory
membyte[511]   - refers to one byte of memory
```
Conditional statements

• **if-else** constructs
  • like C, except that instead of open and close brackets `{ ... }` use keywords `begin` ... `end` to group multiple assignments associated with a given condition
  • `begin` ... `end` are not needed for single assignments

• **case** constructs
  • similar to C switch statements, selecting one of multiple options based on values of a single selection signal

• **for** 
  • `for (i = 0; i < 10; i = i + 1)` statements

• **repeat**
  • `repeat (count)` statements //repeat statements “count” times

• **while**
  • `while (abc)` statements //repeat statements while abc “true” (non-0)
if-else example

```verilog
module  MUX4 (A,B,C,D,S0,S1,Z);
input  A,B,C,D,S0,S1;
output Z;
wire  Z1,Z2;
always
  begin
    if ((S1 == 1'b0) && (S0 == 1'b0)) Z = A;
    else if ((S1 == 1'b0) && (S0 == 1'b1)) Z = B;
    else if ((S1 == 1'b1) && (S0 == 1'b0)) Z = C;
    else
      Z = D;
  end
endmodule
```
module tri_asgn (source, ce, wrclk, selector, result);
input [3:0] source;
input ce, wrclk;
input [1:0] selector;
output result;
reg result;
reg [3:0] intreg;
// combine net declaration and assignment
wire [1:0] sel = selector;
tri result_int; //tri (tri-state) is same as wire
// continuous assignment statement
assign // (condition) ? true-result : false-result
  result_int = (sel == 2'b00) ? intreg[0] : 1'bZ,
  result_int = (sel == 2'b01) ? intreg[1] : 1'bZ,
  result_int = (sel == 2'b10) ? intreg[2] : 1'bZ,
  result_int = (sel == 2'b11) ? intreg[3] : 1'bZ;

//“if” statement
always @(posedge wrclk)
begin
  if (ce)
    begin
      intreg = source ;
      result = result_int ;
    end
end
endmodule

Example:
tri-state bus driver
FSM modeling styles

module FSM (CLK,X,Z);
input CLK,X;
output Z;
reg [1:0] CS;
parameter SA = 2'b00 // define state A with binary value 00
parameter SB = 2'b01 // define state B with binary value 01
parameter SC = 2'b10 // define state C with binary value 10
// if-else statement format
always @ (posedge CLK)
begin
if (CS == SA) // IF-ELSE form
begin
if (X == 0) CS = SC;
else CS = SB;
end
else if (CS == SB)
begin
if (X == 0) CS = SA;
else CS = SC;
end
else
begin
if (X == 0) CS = SB;
else CS = SA;
end
end
// case statement form
always @ (CS)
case (CS) // CASE (selector)
SA: begin
    Z = 1'b0;
end
SB: begin
    Z = 1'b1;
end
SC: begin
    Z = 1'b1;
end
endcase
end
endmodule
for loop – similar to C construct

// 32-bit full adder
always
begin
    for (n=0; n<32; n++)    // ripple carry form
    begin
        sum[n] = Ain[n] ^ Bin[n] ^ carry[n];
        carry[n+1] = (Ain[n] & Bin[n]) | (Ain[n] & carry[n]) | (Bin[n] & carry[n]);
    end
end
while loop – execute until while expression not true

reg [15:0] buffer [0:7];
integer k;
...
always @(posedge clock)
begin
    k = 8;
    while (k)  //store data at posedge of next 8 clocks
    begin
        @(posedge clock) buffer[k] = data;
        k = k - 1;
    end
end
repeat loop – repeat a fixed times

parameter cycles = 8;   // repeat loop counter for below
reg [15:0] buffer [0:7];
integer k;
...
always @(posedge clock)
begin
    k = 0;
    repeat (cycles)   //store data at posedge of next 8 clocks
        begin
            @(posedge clock) buffer[k] = data;
            k = k + 1;
        end
end
Signal strengths

- Verilog allows a signal strength to be associated with a logic state
  
  *(Similar to IEEE 1164 std_logic for VHDL)*

<table>
<thead>
<tr>
<th>Strength</th>
<th>Abbreviation</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply</td>
<td>Su</td>
<td>7</td>
<td>Power supply</td>
</tr>
<tr>
<td>Strong</td>
<td>St</td>
<td>6</td>
<td>Default gate drive (default)</td>
</tr>
<tr>
<td>Pull</td>
<td>Pu</td>
<td>5</td>
<td>Pull up/down</td>
</tr>
<tr>
<td>Large</td>
<td>La</td>
<td>4</td>
<td>Large capacitance</td>
</tr>
<tr>
<td>Weak</td>
<td>We</td>
<td>3</td>
<td>Weak drive</td>
</tr>
<tr>
<td>Medium</td>
<td>Me</td>
<td>2</td>
<td>Medium capacitance</td>
</tr>
<tr>
<td>Small</td>
<td>Sm</td>
<td>1</td>
<td>Small capacitance</td>
</tr>
<tr>
<td>High Z</td>
<td>Hi</td>
<td>0</td>
<td>High impedance</td>
</tr>
</tbody>
</table>

Examples: *strong1, pull1, supply0, Su1, Su0, etc.*
Timing control and delays**

`timescale 1ns/10ps //time units/precision (s,ns,ps,fs), multiplier=1,10,100`

Intra-assignment:

```plaintext
x = #5 y;        //Equivalent to the following
hold = y;        //capture y at t = 0
#5;              //delay until t = 5
x = hold;        //update x
```

Delayed assignment:

```plaintext
#5 x = y;        //Equivalent to the following
#5;              //delay from t = 0 until t = 5
x = y;           //change the value
```

** Delays are ignored by synthesis tools
Producing a clock signal

```
initial x = 0;       // set initial value
always begin         // block is repeated (assume t=0 initially)
    #25 x = 1;     // delay to t=25, then continue by assigning x=1
    #10 x = 0;    // delay to t=35, then continue by assigning x=0
    #5;          // delay to t=40, then continue
end
```
Specifying delays

• Net delays:
  assign #8 a = b & c; //a changes 8 time units after b/c change
  wire #8 a = b & c; //equivalent to the following statement pair
  wire a;
  assign #8 a = b & c;
  //above also equivalent to the following statement pair
  wire #8 a; //8 units of delay to any assignment to net a
  assign a = b & c;

Logic gate delays:
  nand #5 N1(out1,in1,in2); //delay of 5 for any change at gate output
  nand #(3,5) N2(out2,in1,in2); //output rising delay=3, falling delay=5
  nand #(3,5,7) N3(out3,in1,in2); //rising delay=3, falling delay=5, delay to hi-Z=7
Allow for process variations and loading

• Triplet of delay values: (minimum : typical : maximum)

  // triplet of net delays
  #(1.1 : 1.4 : 1.5) assign delay_a = a;

  // triplet of nand gate rise, fall times
  nand #(1:2:3, 2:3:4) N1(out1, in1, in2);

  // 3 triplets of buffer delays, for rise, fall and change to hi-Z state
  buf #(1:2:3, 2:3:4, 4:5:6) B1(out2, in3);
Example – D flip flop

module example
    reg Q, Clk;
    wire D;
    assign  D = 1; //D=1 for this example
    always @(posedge Clk)  Q = D; //normal flip flop clocking
    initial Clk = 0; //initial state of Clk reg
    always #10 Clk = ~Clk; //toggle clock for period of 20
    initial begin
        #50;
        $finish; //simulation control – end simulation
    end
    always begin
        $display("T=",%2g, $time," D=",D," Clk =",Clk," Q="Q); //generate output listing every 10 time units
        #10;
    end
endmodule
Specify block for pin-to-pin delays

module DFF (Q, clk, D, pre, clr);
    input clk,D,pre,clr; output Q;
    DFlipFlop(Q, clk, D); //previously-defined D flip flop module
specify specparam
    tPLH_clk_Q = 3, tPHL_clk_Q = 2.9;
    tPLH_set_Q = 1.2, tPHL_set_Q = 1.1;
    (clk => Q) = (tPLH_clk_Q,tPHL_clk_Q); //=> clk to Q (rise,fall)
    (pre,clr *> Q) = (tPLH_set_Q,tPHL_set_Q); //*> each input to each output
end specify
endmodule
Blocking vs non-blocking assignments

• Blocking statements (x = y;)
  • Executed in order listed, delaying execution of next statement as specified
  • Will not block execution of statements in parallel blocks

```plaintext
x = 0; // x changes at t = 0
a = 1; // a changes at t = 0
#10 c = 3; // delay until t=10, then c changes
#15 d = 4; // delay until t=25, then d changes
e = 5; // e changes at t = 25
```
Blocking vs non-blocking assignments

• Non-blocking statements (\( x <= y; \) )
  • Schedule assignments without blocking other statements (VHDL-like)
  • Do not wait for statement to execute

\[
\begin{align*}
x &= 0; & \text{//execute at } t = 0 \\
a &= 1; & \text{//execute at } t = 0 \\
c &= #15 \ 3; & \text{//process at } t = 0, \text{ schedule } c \text{ to change at } t = 15 \\
d &= #10 \ 4; & \text{//process at } t = 0, \text{ schedule } d \text{ to change at } t = 10 \\
c &= c + 1; & \text{//process at } t = 0, \text{ schedule } c \text{ to change at } t = 0
\end{align*}
\]
Example of blocking/non-blocking delays

initial begin
    a = 1; b = 0;  //change at t=0
    #1  b = 1;    //block until t=1; then b=1
    c = #1 1;    //block until t=2, c=val from t=1
    #1;          //delay to t=3
    d = 1;       //change at t=3
    e <= #1 1;   //non-blocking, e change at t=4
    #1  f <= 1;  //delay to t=4, non-blocking
    g <= 1;      //non-blocking
end

Results:

<table>
<thead>
<tr>
<th>t</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example

• Blocking: (a-b end up with same value – race condition)
  always @(posedge clock)
    a = b; //change a NOW
  always @(posedge clock)
    b = a; //change b to new a value

• Non-blocking: (a-b swap values)
  always @(posedge clock)
    a <= b; //read b at t=0, schedule a to change
  always @(posedge clock)
    b <= a; //read a at t=0, schedule b to change
Verilog simulation commands

$finish       //stop simulation
$time         //current simulation time
$display("Some text to be printed to listing window");
$monitor("T=", $time, ", A=", A, ", B=", B);   //print text & signals to list window
  T=0 A=1 B=x       //similar to C printf statement
  T=5 A=0 B=1
...
$dumpvars     //
User-defined primitive gates (combinational)

• Define own primitive “gates” via a truth table

```plaintext
primitive Adder (Sum, InA, InB);
  output Sum; input InA, InB;
  table
    // inputs : output
    00 : 0;
    01:  1;
    10:  1;
    11 : 0;
  endtable
endprimitive
```
User-defined primitive gates (sequential)

• Include a “state” in the primitive truth table between input & output

primitive Dlatch (Q, Clock, Data);
  output Q; reg Q; input Clock, Data;
  table
    // inputs : present state : output (next state)
    1 0 : ? : 0 ; // ? Represents 0, 1, or x
    1 1 : b : 1 ; // b Represents 0 or 1 input
    1 1 : x : 1 ; // can combine with previous line
    0 1 : ? : - ; // - represents no change in output
  endtable
endprimitive
/* Lab 2: Binary to Seven-Segment Display Driver */
module bin_2_7seg (seg7, hexval);
    input [2:0] hexval;
    output [6:0] seg7;
    reg [6:0] seg7;

    always @(hexval) begin
        case (hexval)
            3'b000: seg7 = 7'b1000000; //0
            3'b001: seg7 = 7'b1111001;  //1
            3'b010: seg7 = 7'b0100100;  //2
            3'b011: seg7 = 7'b0110000;  //3
            3'b100: seg7 = 7'b0011001;  //4
            3'b101: seg7 = 7'b0010010;  //5
            3'b110: seg7 = 7'b0000010;  //6
            3'b111: seg7 = 7'b1111000;  //7
        endcase
    end
endmodule
/* Lab 4 – Moore model finite state machine */
module MooreFSM (RST, EN, Clock, OUT, C1, C0);
  input RST, EN, Clock;  output C1, C0;
  output [3:0] OUT;  reg [3:0] OUT;
  parameter S0 = 4'b0001;   parameter S1 = 4'b0010;
  parameter S2 = 4'b0100;   parameter S3 = 4'b1000;
  always @(posedge Clock) begin
    if (RST == 1) begin
      OUT = S0; //reset to S0
    end
    else if (EN == 1) begin //state changes
      case (OUT)
        S0: OUT = S1;
        S1: OUT = S2;
        S2: OUT = S3;
        S3: OUT = S0;
      endcase
    end
    assign C1 = OUT[3] | OUT[2];  //Encode outputs
    assign C0 = OUT[1] | OUT[3];
  end
endmodule
/ * Lab 5 – Universal 8-bit register/counter */
module Counter(CLK, RST, CE, M, Din, Q);
    input CLK, RST, CE;
    input [1:0] M;
    input [7:0] Din;
    output [7:0] Q;
    reg [7:0] Q;

    always @(posedge CLK) begin
        if (RST) begin
            Q = 8'h00; //reset
        end
        else if (CE) begin //clock enable
            if (M == 2'b01) Q = Q << 1; //shift
            else if (M == 2'b10) Q = Q + 1; //count
            else if (M == 2'b11) Q = Din; //load
        end
    end
endmodule
/* Lab 6 - Register file */
`timescale 1ns / 1ns

module RegFile (ReadAddress, WriteAddress, WE, DataIn, DataOut);
    input [3:0] ReadAddress, WriteAddress;
    input [7:0] DataIn;
    input WE;
    output [7:0] DataOut;

    reg [7:0] RF [0:15]; //16 8-bit registers

    assign DataOut = RF[ReadAddress]; //continuous read

    always @(WE) begin
        if (WE)
            RF[WriteAddress] = DataIn; //write register
    end
endmodule
module RegFile_tb;

//Internal signals declarations:
reg [3:0] ra;
reg [3:0] wa;
reg we;
reg [7:0] din;
wire [7:0] dout;

// Unit Under Test port map
RegFile UUT (
  .ReadAddress(ra),
  .WriteAddress(wa),
  .WE(we),
  .DataIn(din),
  .DataOut(dout));
//testbench continued – stimulus for inputs

initial begin
    we = 0;
    ra = 4'b0000;
    din = 8'd0;
    for (wa = 4'h0; wa != 4'hf; wa = wa + 1) begin  //16 write operations
        din = din + 5;
        #5 we = 1; //we pulse = 5ns
        #5 we = 0; //we period = 10ns
    end
    for (ra = 4'h0; ra != 4'hf; ra = ra + 1) begin  //read the 16 registers
        #10; //read time 10ns
    end
    $finish;
end

endmodule