Verilog for Behavioral Modeling

Nestoras Tzartzanis
E-mail: nestoras@isi.edu

Dept. of Electrical Engineering-Systems
University of Southern California
Objective of the Lecture

• To address those features of Verilog that are required for the class project
  ⇒ discussion will be limited to behavioral modeling
  ⇒ features will be briefly presented
  ⇒ emphasis will be given to the examples
Warning

- This lecture includes features supported by and tested with the Cadence Verilog-XL simulator
- The primary source of the presented material is the Cadence Verilog-XL Reference Manual
Outline

• Introduction
• Data types
• Expressions
• Assignments
• Behavioral modeling
• Hierarchical structures *(modules)*
• System (built-in) functions
• Example
Verilog Features

- Verilog can simulate models at the following levels:
  - algorithmic
  - RTL
  - gate
  - switch
- Verilog offers:
  - behavioral language
  - structural language
Verilog Behavioral Language

- Structures procedures for sequential or concurrent execution
- Explicit control of the time of procedure activation specified by both delay expressions and by value changes called event expressions
- Explicitly named events to trigger the enabling and disabling of actions in their procedures
- Procedural constructs for conditional, if-else, case, and looping operations
- Arithmetic, logical, bit-wise, and reduction operators for expressions
- Procedures called tasks that can have parameters and non-zero time durations (*not covered in today’s lecture*)
- Procedures called functions that allow the definition of new operators (*not covered in today’s lecture*)
Outline

• Introduction

=======>• Data types

• Expressions
• Assignments
• Behavioral modeling
• Hierarchical structures
• System (built-in) functions
• Example
Data Types

• Value sets:
  — 0: logic zero or false condition
  — 1: logic one or true condition
  — x: unknown logic value
  — z: high-impedance state

• Registers

• Nets

• Memories

• Integers and times
  — integer type declaration example: integer n, k[1:64];

• Reals
Data Types: Registers

- Abstractions of data storage elements
- They store a value from one assignment to the other
- Register type declaration examples:
  ```verilog
  reg a;       // a scalar register
  reg [3:0] b; // a 4-bit vector register
  reg [2:5] c;
  ```
Data Types: Nets

- Physical connections
- They do not store a value
- They must be driven by a driver (i.e., gate or continuous assignment)
- Their value is z, if not driven
- Net type declaration examples:
  
  ```verilog
  wire d; // a scalar wire
  wire [3:0] e, f; // 4-bit vector wires
  ```

- Warning:
  - There are more types of nets than wires, but they are not needed for behavioral modeling
Data Types: Memories

- Memories are modeled as arrays of registers
- Each register in the array (i.e., an element or word) is addressed by a single array index
- Memories are declared in register declaration statements:
  
  reg [31:0] Imem[0:255]; // a 256-word, 32-bit memory
Parameters

- Parameters are not variables
  \[\Rightarrow\] they are constants
- Parameter declaration examples
  
  ```verilog
  parameter size = 8; // defines size as a constant with value 16
  ```
Outline

• Introduction
• Data types

Expressions
• Assignments
• Behavioral modeling
• Hierarchical structures
• System (built-in) functions
• Example
Expressions

- The following operator types are supported:
  - binary arithmetic
  - relational
  - equality
  - logical
  - bit-wise
  - reduction
  - shift
  - concatenation
  - conditional
Number Representation

- Constant numbers can be: decimal, hexadecimal, octal, or binary
- Two forms of representation are available:
  - simple decimal number (e.g., 45, 507, 234, etc.)
  - `<size><base_format><number>`
    → `<size>`: number of bits that the constant contains
    → `<base_format>`: a letter specifying the number’s base (d, h, o, or b), followed by the single quote character (’)
    → `<number>`: digits that are legal for the specified `<base_format>`

Examples:
- `4'b1001` // a 4-bit binary number
- `16'ha04f` // a 16-bit hexadecimal number
- `4'b01x0` // a 4-bit binary number
- `12'hx` // a 12-bit unknown number
Arithmetic Operators

- **Binary**: +, -, *, /, % (the modulus operator)
- **Unary**: +, -
- Integer division truncates any fractional part
- The result of a modulus operation takes the sign of the first operand
- If any operand bit value is the unknown value x, then the entire result value is x
- Register data types are used as unsigned values
  — negative numbers are stored in two’s complement form
Relational Operators

- a < b  
  a less than b

- a > b  
  a greater than b

- a <= b  
  a less than or equal to b

- a >= b  
  a greater than or equal to b

- The result is a scalar value:
  - 0 if the relation is false
  - 1 if the relation is true
  - x if any of the operands has unknown x bits

- Note: If a value is x or z, then the result of that test is false
### Equality Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a === b</code></td>
<td>a equal to b, including x and z</td>
</tr>
<tr>
<td><code>a !== b</code></td>
<td>a not equal to b, including x and z</td>
</tr>
<tr>
<td><code>a == b</code></td>
<td>a equal to b, result may be unknown</td>
</tr>
<tr>
<td><code>a != b</code></td>
<td>a not equal to b, result may be unknown</td>
</tr>
</tbody>
</table>

- Operands are compared bit by bit, with zero filling if the two operands do not have the same length.
- Result is 0 (false) or 1 (true).
- For the `==` and `!=` operators the result is x, if either operand contains an x or a z.
- For the `===` and `!==` operators:
  - bits with x and z are included in the comparison and must match for the result to be true.
  - the result is always 0 or 1.
Logical Operators

- ! logical negation
- && logical and
- || logical or

- Expressions connected by && and || are evaluated from left to right
- Evaluation stops as soon as the result is known
- The result is a scalar value:
  - 0 if the relation is false
  - 1 if the relation is true
  - x if any of the operands has unknown x bits
Bit-wise Operators

- \( \sim \) negation
- \& and
- \( | \) inclusive or
- ^ exclusive or
- ^\~ or \(~^ \) exclusive nor (equivalence)

- Computations include unknown bits, in the following way:
  \[ \sim x = x, \ 0\&x = 0, \ 1\&x = x\&x = x, \ 1|x = 1, \ 0|x = x|x = x \]
  \[ 0^x = 1^x = x^x = x, \ 0^\sim x = 1^\sim x = x^\sim x = x \]

- When operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions
Reduction Operators

- Reduction operators are unary
- They perform a bit-wise operation on a single operand to produce a single bit result
- Reduction unary NAND and NOR operators operate as AND and OR respectively, but with their outputs negated
- Unknown bits are treated as described before
Shift Operators

- The left operand is shifted by the number of bit positions given by the right operand
- The vacated bit positions are filled with zeroes

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;</td>
<td>left shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right shift</td>
</tr>
</tbody>
</table>
Concatenation Operator

- Concatenations are expressed using the brace characters { and }, with commas separating the expressions within.

- Examples

  \{a, b[3:0], c, 4'b1001\}  // if a and c are 8-bit numbers, the results has 24 bits

- Unsized constant numbers are not allowed in concatenations

- Repetition multipliers that must be constants can be used:

  \{3\{a\}\}  // this is equivalent to \{a, a, a\}

- Nested concatenations are possible:

  \{b, \{3\{c, d\}\}\}  // this is equivalent to \{b, c, d, c, d, c, d\}
Conditional Operator

- The conditional operator has the following C-like format:
  
  \[ cond_{expr} \ ? \ true_{expr} : false_{expr} \]

- The \( true_{expr} \) or the \( false_{expr} \) is evaluated and used as a result depending on whether \( cond_{expr} \) evaluates to true or false.
## Operator Precedence Rules

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!, ~</code></td>
<td>highest precedence</td>
</tr>
<tr>
<td><code>*, /, %</code></td>
<td>highest precedence</td>
</tr>
<tr>
<td><code>+, -</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;&lt;, &gt;&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;, &lt;=, &gt;, &gt;=</code></td>
<td></td>
</tr>
<tr>
<td><code>==, !=, ===, !==</code></td>
<td>lowest precedence</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td></td>
</tr>
<tr>
<td><code>^, ^~</code></td>
<td></td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td></td>
</tr>
<tr>
<td>`</td>
<td></td>
</tr>
<tr>
<td><code>?:</code></td>
<td></td>
</tr>
</tbody>
</table>
Issues about Operands

• Parts or single bits of registers and nets can be used as operands:

  reg [7:0] a; // a is declared as an 8-bit register

  a[4] could be used as a single-bit operand in an expression

  a[6:3] could be used as a 4-bit operand in an expression

• There is no mechanism to express bit-selects or port-selects of memory elements directly (only entire memory words)

  reg [7:0] mema[0:255] // mema is declared as a 8-bit, 256 word register

  mema[15] is a valid operand which includes the 16th memory word
Outline

• Introduction
• Data types
• Expressions

Assignments
• Behavioral modeling
• Hierarchical structures
• System (built-in) functions
• Example
Assignments

- Continuous assignments, which assigns values to nets
- Procedural assignments, which assigns values to registers
Continuous Assignments

- Continuous assignments drive values onto nets (vector and scalar)
- The assignment occurs whenever simulation causes the value of the right-hand side to change
- They provide a way to model combinational logic specifying the logical expression that drives the net instead of an interconnection of gates
- The right-hand side expression is not restricted in any way
- Whenever an operand in the right hand side changes value during simulation, the whole right-hand side expression is evaluated and assigned to the left-hand side
- Modeling a 16-bit adder with a continuous assignment
  ```verilog
  wire [15:0] sum, a, b; // declaration of 16-bit vector nets
  wire cin, cout; // declaration of 1-bit (scalar) nets
  assign {cout, sum} = a + b + cin;
  ```
Procedural Assignments

- Procedural assignments drive values onto registers (vector and scalar)
- They do not have duration (the register data type holds the value of the assignment until the next procedural assignment to the register)
- They occur within procedures such as *always* and *initial* (described right after)
- They are triggered when the flow of execution in the simulation reaches them
Outline

- Introduction
- Data types
- Expressions
- Assignments

---

Behavioral modeling

- Hierarchical structures
- System (built-in) functions
- Example
Behavioral Modeling

- Introduction
- Procedural assignments
- Conditional statement
- Case statement
- Looping statements
- Timing controls
- Block statements
- Structured procedures
Introduction to Behavioral Modeling

- Activity starts at the control constructs *initial* and *always*
- Each *initial* and *always* statement starts a new concurrent activity flow
- Time units are used to represent simulation time
- A simple example:

```verilog
module behave;
    reg [1:0] a,b; // a and b are 2-bit register data types
initial // this statement is executed only once
    begin
        a = 2'b01; // a is initialized to 01
        b = 2'b00; // b is initialized to 00
    end
always // this statement is repetitively executed until
    begin // simulation is completed
        #50 a = ~a; // register a inverts every 50 time units
    end
always // this statement is repetitively executed
    begin // simulation is completed
        #100 b = ~b; // register b inverts every 100 time units
    end
endmodule
```

Nestoras Tzartzanis
More on Procedural Assignments

• Blocking procedural assignments: \(<lvalue>=<timing_control> <expression>\)
  — must be executed before the execution of the statements that follow them in a sequential block
  — does not prevent the execution of statements that follow them in a parallel block

• Non-blocking procedural assignments: \(<lvalue><= <timing_control> <expression>\)
  — allow to schedule assignments without blocking the procedural flow
  — used whenever you want to make several register assignments within the same time step without regard to order or dependance upon each other (e.g., register swap)
  — they are executed in two steps: (a) the simulator evaluates the right-hand side, (b) the assignment occurs at the end of the time step indicated by the \(<timing_control>\)

• \(<timing_control>\) is optional in both cases

• It is possible to complete your project by using blocking only statements
**Procedural Assignment Example**

module block;
    reg a, b, c, d, e, f;
    initial begin
        a = #10 1; // a is assigned at simulation time 10
        b = #2 0; // b is assigned at simulation time 12
        c = #4 1; // c is assigned at simulation time 16
    end
    initial begin
        d <= #10 1; // d is assigned at simulation time 10
        e <= #2 0; // e is assigned at simulation time 2
        f <= #4 1; // f is assigned at simulation time 4
    end
endmodule
module block;
    reg a, b, c, d, e, f;
initial begin
    #10 a = 1;  // a is assigned at simulation time 10
    #2 b = 0;   // b is assigned at simulation time 12
    #4 c = 1;   // c is assigned at simulation time 16
end
initial begin
    #10 d <= 1;  // d is assigned at simulation time 10
    #2 e <= 0;   // e is assigned at simulation time 12
    $4 f <= 1;   // f is assigned at simulation time 16
end
endmodule
Conditional Statement

- $if (<expression>) <statement> else <statement>$
- Equivalent to: $if (<expression> != 0) <statement> else <statement>$
- $else$-clause is optional and is always associated with the closest previous $if$ that lacks an $else$
- Example
  
  ```
  if (index > 0)
    if (rega > regb)
      result = rega;
    else // else applies to preceding if
      result = regb;
  ```
**Case Statement**

- Case statement is a special multi-way decision statement:

```verilog
define reg [1:0] sel;
define reg [15:0] in0, in1, in2, in3, out;

\begin{verbatim}
case (sel)
  2'b00: out = in0;
  2'b01: out = in1;
  2'b10: out = in2;
  2'b11: out = in3;
endcase
\end{verbatim}

default out = 16'bx;
```

- The *default* statement is optional

- If all comparisons fail and the default is not given, none of the case statements is executed
Case Statement (cond)

- Case statement used to trap x and z values:

```verilog
case (sel)
    2'b00:   out = in0;
    2'b01:   out = in1;
    2'b0x,
    2'b0z:   out = flag ? 16'bx : 16'b0;
    2'b10:   out = in2;
    2'b11:   out = in3;
    2'bx0,
    2'bz0:   out = flag ? 16'bx : 16'b0;
    default out = 16'bx;
endcase
```
Case Statement with Don’t-Cares

- **casez**: high-impedance (z) values are treated as don’t-cares
- **casex**: high-impedance (z) and unknown (x) values are treated as don’t-cares

```verilog
reg [31:0] instruction;
reg [2:0] aluOp;

casez (instruction[31:26])
    6'b00????: aluOp = 3'b000; // ? indicates a don’t-care bit position
    6'b100???: aluOp = 3'b001;
    6'b101???: aluOp = 3'b010;
    6'b1100??: aluOp = 3'b001;
    6'b11010?: aluOp = 3'b100;
    6'b110110: aluOp = 3'b101;
endcase
```
Looping Statements

- `forever`
- `repeat`
- `while`
- `for`
Forever Statement

- Continuously executes a statement
- Should be used in conjunction with the timing controls (to be presented later)
Repeat Statement

- Executes a statement a fixed number of times

```verilog
class parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;
begin: mult // begin-end statements could optionally be named
  reg [longsize:1] shift_opa, shift_opb;
  shift_opa = opa;
  shift_opb = opb;
  result = 0;
  repeat (size)
    begin
      if (shift_opb[1])
        result = result + shift_opa;
      shift_opa = shift_opa << 1;
      shift_opb = shift_opb >> 1;
    end
  end
end
```

Nestoras Tzartzanis
**While Statement**

- Executes a statement until an expression becomes false

```verilog
begin: count1s
    reg [7:0] tempreg;
    count = 0;
    tempreg = rega;
    while (tempreg)
        begin
            if (tempreg[0]) count = count + 1;
            tempreg = tempreg >> 1;
        end
    end
end
```
For Statement

- Similar to C for-statement

  ```verilog
  for (initial_assignment; condition; step_assignment)
  statement
  ```

- Using a for-loop to initialize a memory:

  ```verilog
  begin :init_mem
    reg [7:0]   tempi;
    for (tempi = 0; tempi < memsize; tempi = tempi + 1)
      memory[tempi] = 0;
  end
  ```
Procedural Timing Controls

- *delay* control, which is introduced by the number symbol (#)
- *event* control, which is introduced by the at symbol (@)
- *wait* statement, which operates like a combination of the event control and the while loop
Delay Control

- Examples

  #10 rega = regb;
  
  #d rega = regb; // d is defined as a parameter
  
  #((d+e)/2) rega = regb;
  
  #regr regr = regr + 1; // delay is the value in regr
Event Control

- Synchronizes execution of a procedural statement with a value change on a net or register, or the occurrence of a declared event
- Examples:

  ```verilog
target
  @r rega = regb; // controlled by any value changes in the register r
  @(posedge clock) rega = regb; // controlled by positive edge on clock
  forever @(negedge clock) rega = regb; // controlled by negative edge
  @(trig or enable) rega = regb;       // controlled by trig or enable
  @(posedge clock_a or posedge clock_b) rega = regb;
  ```
Wait Statement

- The execution of a statement can be delayed until a condition becomes true
- The wait statement checks a condition: if it is false, causes the procedure to pause until that condition becomes true before continuing

   wait (condition_expression) statement

- Example:

```verilog
global reg enable, a, b;
initial begin
    wait (!enable) #10 a = 1; // a becomes 1 at #110
    #10 b = 1; // b becomes 1 at #120
end
initial
    #100 enable = 0;
```
Intra-Assignment Timing Controls

- Can be delay or event controls
- The right-hand side expression is evaluate before the delay or the event, but the assignment happens when the delay expires or the event happens:

  \[
  a = \#5 b;
  \]
  equivalent to:

  ```
  begin
  temp = b;
  #5 a = temp;
  end
  ```

  \[
  a = @(posedge clk) b;
  \]
  equivalent to:

  ```
  begin
  temp = b;
  @(posedge clk) a = temp;
  end
  ```
Block Statements

- Sequential blocks delimited by the keywords \textit{begin} and \textit{end}
  - statements execute in sequence, one after another
  - delays are cumulative; each statement executes after all the delays preceding it have elapsed
  - control passes out of the block after the last statement executes

- Parallel blocks delimited by the keywords \textit{fork} and \textit{join}
  - statements execute concurrently
  - delay values for each statement are relative to the simulation time when control enters the block
  - delay control is used to provide time-ordering for assignments
  - control passes out of the block when the last time-ordered statement executes or a \textit{disable} statement executes
Sequential Block Examples

begin
    #10 a = 1; // a becomes 1 at #10
    #20 a = 0; // a becomes 0 at #30
end

begin
    @trig r = 1; // r becomes 1 when trig changes value
    #250 r = 0; // r becomes 0 250 time units after r changes value
end

begin
    @(posedge clock) q = 0; // q becomes 0 at the posedge of clock
    @(posedge clock) q = 1; // q becomes 1 at the next posedge of clock
end
Parallel Block Examples

fork
  #30 a = 0; // a becomes 0 at #30
  #10 a = 1; // a becomes 1 at #10
join
fork
  @(posedge clock) q = 0; // q becomes 0 at the posedge of clock
  @(posedge clock) w = 1; // w becomes 1 at the same posedge of clock
join
begin
  fork
    @(Aevent);
    @(Bevent);
  join
  areg = breg; // this statement is executed after both Aevent and
  // Bevent have occurred regardless their order
end
Structured Procedures

- initial statement
- always statement
- task (not discussed)
- function (not discussed)
**Initial Statement**

- Activated at the beginning of simulation
- Executed once

```verilog
initial
begin
  areg = 0; // initialize a register
  for (index = 0; index < size; index = index + 1)
    memory[index] = 0; // initialize a memory
end

initial
begin
  inputs = 6'b000000;
  #10 inputs = 6'b001011;
  #10 inputs = 6'b110001;
end
```
Always Statement

- Activated at the beginning of simulation
- Repeats continuously throughout the whole simulation run

always
  #100 clock = ~clock // creates a clock signal with #200 period time
always
  @(posedge clock) // the block below starts execution at posedge of clock
  begin
    #10 a = 0; // #10 after the posedge of clock, a becomes 0
    #20 b = 1; // #30 after the posedge of clock, b becomes 1
    #40 b = 0; // #70 after the posedge of clock, b becomes 1
  end
Outline

• Introduction
• Data types
• Expressions
• Assignments
• Behavioral modeling

Hierarchical structures
• System (built-in) functions
• Example
Hierarchical Structures

- Hierarchy consists of modules
- High-level modules create instances of lower-level modules
- Modules communicate through input, output, and bidirectional ports
- Each module definition stands alone; the definitions are not nested
Syntax of Module Definitions

module mname (port1, port2, port3, port4, ...); // keywords are in italics
output port1;
input port2, port3;
inout port4;
reg port1;       // output ports are usually defined as registers
reg variable1, variable2; // definitions of local variables
wire variable3;
initial
    statements  // any of the behavioral modeling statements
always
    statements  // any of the behavioral modeling statements
assign
    continuous-assign-statements // required for nets
mod1 mod1Instance (variable1, variable2, ...)  // instantiating lower-level modules
endmodule
Module Ports

- Used to connect modules
- Can be thought as procedure parameters
- Could be input, output, or inout
- Input and inout ports can be only net data type
- My advice is:
  - define all the output ports as registers
    - works fine with input ports because, in module connections, input ports are driven by output ports that are registers
  - define all the inout ports as wires and use assign statements for them
Example

module m;
    reg clk;
    wire [1:10] out_a, in_a;
    wire [1:5] out_b, in_b;
    // create an instance and set parameters
    vddf #(10, 15) mod_a (out_a, in_a, clk);
    // create an instance leaving default values
    vddf mod_b (out_b, in_b, clk);
endmodule

module vddf (out, in, clk);
    parameter size = 1, delay = 1;
    input [0:size-1] in;
    input clk;
    output [0:size-1] out;
    reg [0:size-1] out;
    always @(posedge clk)
        #delay out = in;
endmodule
Outline

• Introduction
• Data types
• Expressions
• Assignments
• Behavioral modeling
• Hierarchical structures

======> System (built-in) functions
• Example
$display and $write

- They display information
- They are identical, except that $display automatically adds a newline character to the end of the output
- Similar to the C printf statement

\[
\begin{align*}
\text{$display} & ("rval = \%h\ \text{hex} \ \%d\ \text{decimal}", \ \text{rval, rval}) \\
\text{$display} & ("rval = \%o\ \text{octal} \ \%b\ \text{binary}", \ \text{rval, rval}) \\
\text{$write} & ("\text{simulation time is $t \ \text{\textbackslash n}}", \ \text{$time}); \ //\text{$time} \ \text{is a function that returns the current simulation time}
\end{align*}
\]
Format Specifications

- `%h` or `%H` display in hexadecimal format
- `%d` or `%D` display in decimal format
- `%o` or `%O` display in octal format
- `%b` or `%B` display in binary format
- `%c` or `%C` display in ASCII character format
- `%m` or `%M` display hierarchical name (doesn’t need a parameter)
- `%s` or `%S` display as a string
- `%t` or `%T` display in current time format
Continuous Monitoring

- $monitor, $monitoron, $monitoroff
- $monitor displays a list of variables any time one of them changes
- $monitoron and $monitoroff enable and disable the monitor statement respectively

$monitor($time, "rxb=%b txb=%b", rxb, txb);
Strobed Monitoring

- The "$strobe" system task displays the specified information at the end of the time unit
- Same format as "$display"

```verilog
forever @(negedge clock)
    $strobe("At time %d, data is %h", $time, data);
```
File Output

- **$fopen** opens a file:
  
  ```verilog
  integer out_file; // out_file is a file descriptor (declared as integer)
  out_file = $fopen("cpu.data");  // cpu.data is the file opened
  ```

- **$fdisplay, $fwrite, $fstrobe, $fmonitor** are used to write data into the file:
  
  ```verilog
  $fdisplay(out_file, "rval = %h hex %d decimal", rval, rval);
  ```

- **$fclose** closes a file:
  
  ```verilog
  $fclose(out_file);
  ```
$\textit{finish}$

- $\textit{finish}$ causes the simulator to exit and pass control back to the host operating system.
- Example: if the simulation time is #800, then use the following statement at the beginning of the top-level module:

  ```verilog
  initial
      #800 $finish;
  ```
Loading Memories from Text Files

- \$readmemb and \$readmemh read and load (binary and hexadecimal respectively) data from a specified text file into a specified memory

  \[
  \text{reg [7:0] mem[1:256]} \quad \text{// a 8-bit, 256-word memory is declared}
  \]

- To load the memory at simulation time 0 starting at address 1 from file mem.data:

  \[
  \text{initial \$readmemh("mem.data", mem)}
  \]

- To load the memory at simulation time 0 starting at address 16 and continuing on towards address 256 from file mem.data:

  \[
  \text{initial \$readmemh("mem.data", mem, 16)}
  \]

- To load the memory at simulation time 0 starting at address 128 and continuing down towards address 1 from file mem.data:

  \[
  \text{initial \$readmemh("mem.data", mem, 128, 1)}
  \]
Displaying Signals as Graphical Waveforms

- Used with *cwaves* graphical interface
- `$shm_open` opens a database
- `$shm_probe` specifies signals whose waveform can be displayed with *cwaves*
- `$shm_close` terminates the simulation’s connection to the database

```verilog
initial
    $shm_open("signals.shm"); // default database name is waves.shm
    $shm_probe(clk, opA, opB);
    // $shm_probe("AS"); would probe all signals
    #800  $shm_close();
    $finish;
```
Outline

• Introduction
• Data types
• Expressions
• Assignments
• Behavioral modeling
• Hierarchical structures
• System (built-in) functions

Example
Simulating a Simple Instruction Set Architecture

- A baby-instruction set:
  
  \[
  \begin{array}{c|c|c}
  0 & \text{Imm} & \text{R}_d \\
  \hline
  1 & \text{R}_{sr} & \text{R}_d \\
  \end{array}
  \]

  MV: \( \text{R}_d \leftarrow \text{sign\_ext(Imm)} \)

  \[
  \begin{array}{c|c|c}
  1 & \text{R}_{sr} & \text{R}_d \\
  \hline
  1 & \text{R}_{sr} & \text{R}_d \\
  \end{array}
  \]

  ADD: \( \text{R}_d \leftarrow \text{R}_d + \text{R}_{sr} \)

  XOR: \( \text{R}_d \leftarrow \text{R}_d \oplus \text{R}_{sr} \)

- No memory instructions
- No control transfer instructions
- If the operand of an instruction is written by the previous one, the old value is read

Nestoras Tzartzanis

February 3, 1998
Block Diagram — Data Path
A Data-Memory Example

- An example where a module port should be declared as *inout* (i.e., the data bus)
- Assuming a single phase clocking scheme, the data bus is actively driven by the test module during the first half cycle and by the memory during the second half cycle
- Temporary register data type variables are required from both sides
- The *assign*-statement should be used