

A Fast Path to Digital Design with Verilog HDL

Mohammad-Reza Movahedin

Email: `mov.courses@gmail.com`

Please report bugs, errors and your valued comments to this Email address.

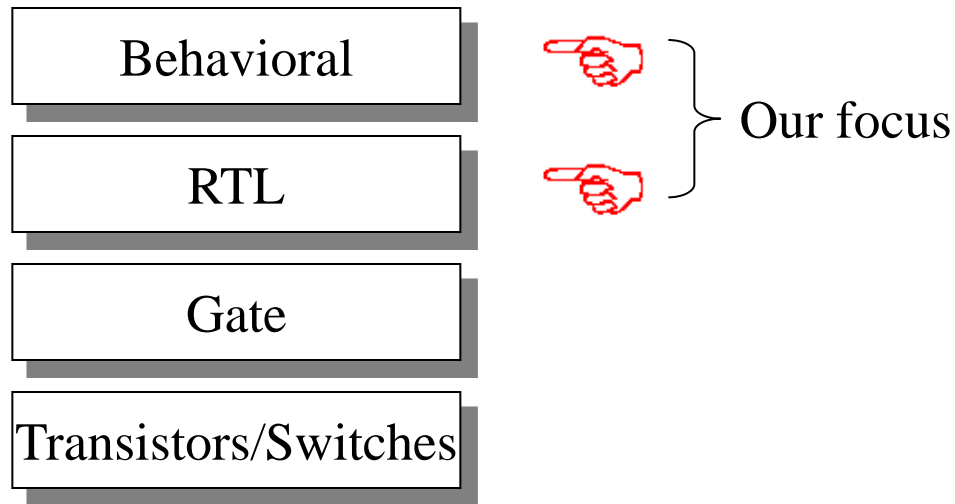
Sept. 2010

Introduction

What is Verilog?

- A Hardware Description/Modeling Language (HDL)
- Originally developed in 1984 as a simulation language
- IEEE Standards: 1364-1995 & 1364-2001
- Further enhancements and maintenance are left to IEEE-P1800 Standards Group (System-Verilog)
- Designs described/modeled by Verilog can be:
 - simulated for validation,
 - synthesized for implementation,
 - formally checked for verification,
 - and so on.

Abstraction Levels in Verilog



Our Strategy in Verilog Training

- Almost 90% of Verilog codes developed by design and verification teams use less than 30% of language capabilities,

Thus our focus is mainly on those 30%.

- Not only the syntax and semantics, but also coding styles are presented and emphasized,
- If there are N ways to do something,
Only 1-2 best, safest, and
most widely used methods are presented.

Main Language Capabilities

- Digital is the only focus, analog/mixed-signal is left to Verilog-AMS,
- Support of concurrency modeling,
- Support of hierarchical designs,
- Support of timing,
- Support of sequential coding for modeling of digital designs,
- Use of C language keywords and syntax

Verilog General Basics

User Identifiers

- Formed from {[A-Z], [a-z], [0-9], _, \$}, but can't begin with \$ or [0-9]
 - `myidentifier` ✓
 - `m_y_identifier` ✓
 - `3my_identifier` ✗
 - `$my_identifier` ✗
 - `_myidentifier$` ✓
- Case sensitivity: `myid` ≠ `Myid`
- Escaped identifiers: `\[any printable ASCII characters]+[]`
 - `\{c@n}_[u]_(c)_#this^??+hurrah=!!`

Comments

- `// The rest of the line is a comment`
- `/* Multiple line
comment */`
- `/* Nesting /* comments */ do NOT work */`

Verilog 4-Value Logic Set

- *0* represents low logic level or false condition
- *1* represents high logic level or true condition
- *x/X* represents unknown logic level
- *z/Z* represents high impedance logic level
but if read by a gate/module, will act as a *x/X*

Logical Operations

| & | 0 | 1 | X |
|--------------|----------|----------|----------|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | X |
| X | 0 | X | X |

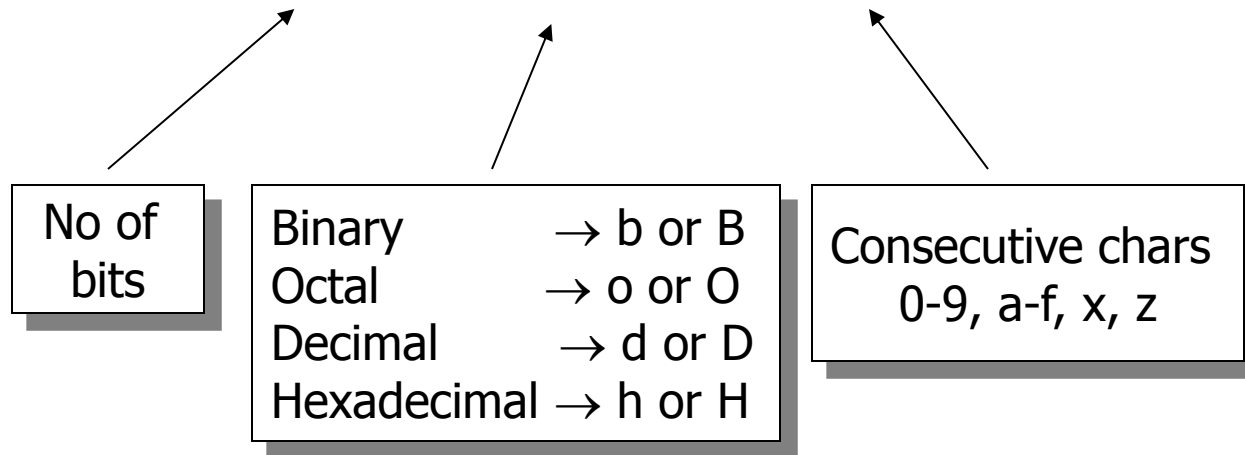
| | 0 | 1 | X |
|----------|----------|----------|----------|
| 0 | 0 | 1 | X |
| 1 | 1 | 1 | 1 |
| X | X | 1 | X |

| ^ | 0 | 1 | X |
|----------|----------|----------|----------|
| 0 | 0 | 1 | X |
| 1 | 1 | 0 | X |
| X | X | X | X |

| ~ | 0 | 1 | X |
|----------|----------|----------|----------|
| | 1 | 0 | X |

Numbers in Verilog (i)

<size>'<radix> <value>



- **8'h ax = 1010xxxx**
- **12'o 3zx7 = 011zzzxxx111**

Numbers in Verilog (ii)

- You can insert “_” for readability
 - `12'b 000_111_010_100`
 - `12'b 000111010100`
 - `12'o 07_24`

} Represent the same number
- Bit extension
 - MS bit = 0, x or z \Rightarrow extend this
 - `4'b x1 = 4'b xx_x1`
 - MS bit = 1 \Rightarrow zero extension
 - `4'b 1x = 4'b 00_1x`

Numbers in Verilog (iii)

- If *size* is omitted it
 - is inferred from the *value* or
 - takes the simulation specific number of bits or
 - takes the machine specific number of bits
- If *radix* is omitted too, decimal is assumed
 - 15 = <size>'d 15

wire is a wire

- *wire* keyword is used to declare an identifier as a real hardware wire, single-line (scalar) or bus,
- When unconnected, it is in high impedance, i.e. has Z value,
- Can have one or multiple drivers (see next table),
- Other net types, i.e. *wor* (*trior*), *wand* (*triand*) and *triereg*, are rarely used,
- *tri* keyword is equal to *wire*.

Multiple Driver Resolution

| wire | 0 | 1 | X | Z |
|------|---|---|---|---|
| 0 | 0 | X | X | 0 |
| 1 | X | 1 | X | 1 |
| X | X | X | X | X |
| Z | 0 | 1 | X | Z |

Implicit *wires*

- An undeclared identifier is treated implicitly as a single bit *wire*,
- Never use this feature, and if possible turn it off in your compiler,
- WARNING: misspelled identifiers can confuse you and easily cause headaches.

Vectors

- Represent busses

```
wire [3:0] busA;  
wire [1:4] busB;  
wire [1:0] busC;
```

- Left number is most significant (MS) bit
- Slice management

```
busC = busA[2:1];    ⇔    busC[1] = busA[2];  
                           busC[0] = busA[1];
```

- Vector assignment (***by position!!***)

```
busB = busA;    ⇔    busB[1] = busA[3];  
                     busB[2] = busA[2];  
                     busB[3] = busA[1];  
                     busB[4] = busA[0];
```

- You should have enough reasons not to use [N:0] format for a bus definition.

Variables

- Several types of variables:
 - *reg*
 - *integer*
 - *real*
 - *time, realtime*
- Only *reg* has two formats of scalar (1-bit) or vectored, but not the rest,
- Should be assigned ONLY inside a sequential portion of a code (*always, initial, task, function*),
- Can be read anywhere in the design, i.e. can act as a driver for a *wire*.

Variables, *reg*

- *reg* keyword is used to declare an identifier as a single-bit or vectored 4-value logic variable, and initialized to all *X*,
- *reg* has nothing to do with registers, it is just a variable,
- Wished they have used *var* keywords instead of *reg*,
- A variable acts as a container that holds the value assigned to it,
- In contrary to a *wire* that needs to have a driver, a *reg/var* holds its value as long as no new value is assigned to it,
- Since it is only assigned in sequential portion of code, only a single value is assigned to it at any time,
thus no multiple driver concept exists for *reg*,
- Based on the context, it can be a clocked register, a latch, output of a combinational logic, a hardware wire, or just a temporary variable with no hardware correspondent,
- Declaration may contain an initial value assignment,
- Again: *reg* is not always a reg!

Variables, *integer* & *real*

- *integer* declares a 32-bit signed number,
- *real* declares a 64-bit double precision floating point number per IEEE-754 standard,
- Never use inside a design, only for test bench development,
- Declaration

```
integer i, k;  
real r;
```
- Integers are initialized to X
- Reals are initialized to *0.0*
- Useful Tasks: *\$realtobits*, *\$bitstoreal*

Variables, *time*

- Special data type for simulation time measuring
- Declaration

```
time my_time;
```

- Use inside procedure

```
my_time = $time; // get current sim time
```

Arrays

- Syntax

```
integer A[1:5];           // 5 integers
reg B[-15:16];            // 32 1-bit regs
reg [7:0] mem[0:1023];    // 1024 8-bit regs
```

- Accessing array elements:

- Entire element: `mem[10] = 8'b 10101010;`
- Element subfield (needs temp storage):

```
reg [7:0] temp;
..
temp = mem[10];
var[6] = temp[2];
```

Arrays, cont.

- Note: Cannot access array subfield or entire array at once, it is an array, not a bus !!

```
var[2:9] = ???;           // WRONG!!
```

```
var = ???;                // WRONG!!
```

- Multi-dimensional *wire* and *reg* arrays are added in 2001:

```
reg [7:0] var [1:10][1:100];
```

- Memory: one-dimensional array of (vectored) *reg*
- Useful tasks: *\$readmemb*, *\$readmemh*

Strings

- Implemented with regs:

```
reg [8*13:1] string_val; // can hold up to 13 chars
..
string_val = "Hello Verilog";
string_val = "hello"; // MS Bytes are filled with 0
string_val = "I am overflowed"; // "I " is truncated
```

- Escaped chars:

- \n newline
- \t tab
- %% %
- \\ \
- \" "

Signed Keyword

- Signed keyword is added in Verilog-2001,
- Signed constants can be defined with an extra s in the radix part of the number, e.g. 4'sb1000 | 8'shAC;
- Can be added in the declaration of a vectored wire or reg, showing that it is a 2's complement signed number,

```
reg signed [15:0]
```

```
wire signed [7:0]
```

- If a vector declared as signed, when an extension is required, sign extension is applied instead of regular zero filling,

```
wire signed [3:0] a = 4'b    1000;
```

```
wire signed [7:0] b = 8'b00000000;
```

```
wire          [7:0] c = 8'b00000000;
```

```
wire [7:0] x = b | (a << 2);           // 8'b11100000
```

```
wire [7:0] y = a | c;                   // 8'b00001000
```

- Will have a different result in *, /, %

Logical & Arithmetic Operators

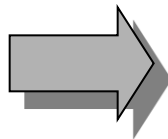
Logical Operators

- C-like behavior:
 - `&&` → logical AND
 - `||` → logical OR
 - `!` → logical NOT
- Operands evaluated to ONE bit value: *0*, *1* or *x*
 - 0 when all bits are 0
 - 1 when at least one bit is 1
 - x when all bits are 0, x or zi.e. all bits OR-ed
- Result is ONE bit value: *0*, *1* or *x*

`A = 6;`

`B = 0;`

`C = x;`



`A && B → 1 && 0 → 0`

`A || !B → 1 || 1 → 1`

`C || B → x || 0 → x`

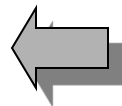
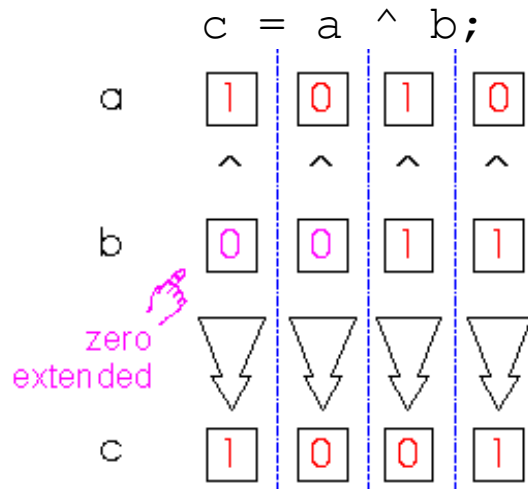
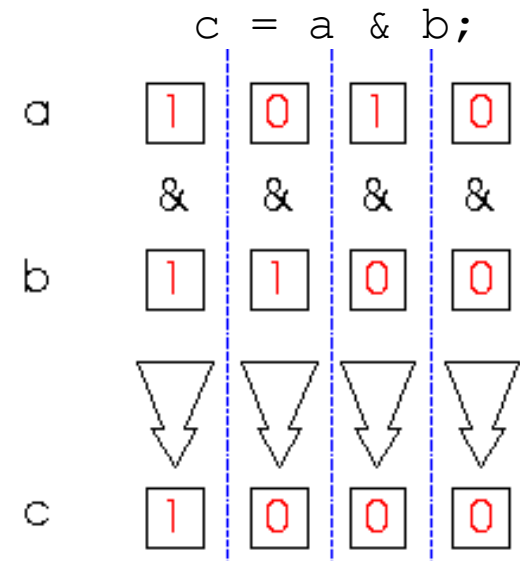
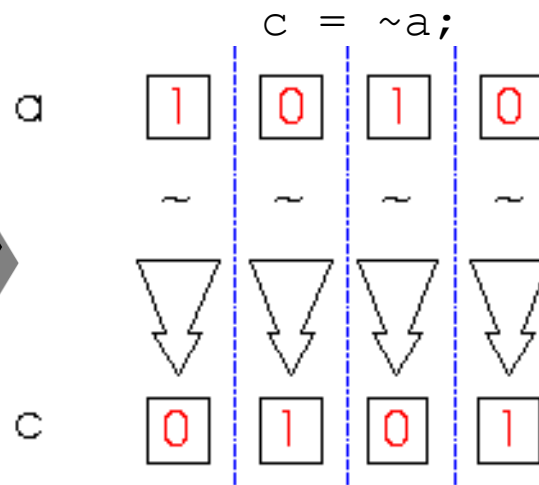
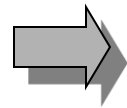
but `C&&B=0`

Bitwise Operators

- `&` → bitwise AND
- `|` → bitwise OR
- `~` → bitwise NOT
- `^` → bitwise XOR
- `~^` or `^^` → bitwise XNOR
- Operation on bit by bit basis
- Warning: `A=2'b10; if(!A) ≠ if(~A)`

Bitwise Operators, cont.

- $a = 4'b1010;$
 $b = 4'b1100;$



- $a = 4'b1010;$
 $b = 2'b11;$

Reduction Operators

- `&` → **AND**
- `|` → **OR**
- `^` → **XOR**
- `~&` → **NAND**
- `~|` → **NOR**
- `~^` **or** `^^` → **XNOR**
- One multi-bit (vector) operand → One single-bit result

```
a = 4'b1001;  
c = |a; // c = 1|0|0|1 = 1
```
- Not a good readable practice, avoid using it if possible,
 - Example: `A && B` \neq `A & & B`
 - Example: `wire A; wire[7:0] B; => A && B` \equiv `A & | B`
 - Bad Code: `if(A &&& B)`

Shift Operators

- `>>` → shift right
- `<<` → shift left
- `>>>` → signed shift right (for signed vectors)
- `<<<` → signed shift left (for signed vectors)
- Result is same size as first operand, sign extended if signed, zero filled otherwise,

```
a = 4'b1010;
```

```
...
```

```
d = a >> 2;    // d = 0010
```

```
c = a << 1;    // c = 0100
```


Concatenation Operator

- $\{op1, op2, \dots\} \rightarrow$ concatenates $op1, op2, \dots$ to a single number
- Operands must be explicitly sized !!

```
reg a;  
reg [2:0] b, c;  
..  
a = 1'b 1;  
b = 3'b 010;  
c = 3'b 101;  
catx = {a, b, c};           // catx = 1_010_101  
caty = {b, 2'b11, a};       // caty = 010_11_1  
catz = {b, 1};              // WRONG !!
```

- Replication ..

```
catr = {{4{a}}, b, {2{c}}}; // catr = 1111_010_101101
```

Relational Operators

- $>$ \rightarrow greater than
- $<$ \rightarrow less than
- $>=$ \rightarrow greater or equal than
- $<=$ \rightarrow less or equal than
- Result is one bit value: 0 , 1 or x

$1 > 0 \quad \rightarrow \mathbf{1}$

$'b1x1 >= 0 \quad \rightarrow \mathbf{x}$

$10 > z \quad \rightarrow \mathbf{x}$

Equality Operators

- `==` → logical equality
 - `!=` → logical inequality
 - `===` → case equality
 - `!==` → case inequality
- } Return *0*, *1* or *x*
- } Return *0* or *1*

– `4'b 1z0x == 4'b 1z0x` → ***x***

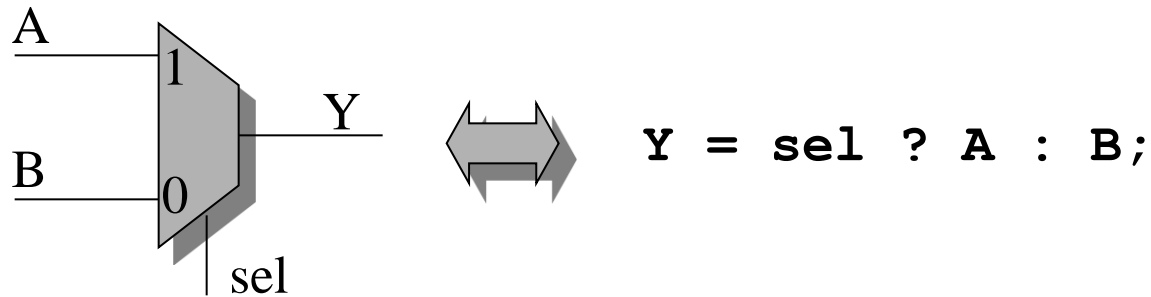
– `4'b 1z0x != 4'b 1z0x` → ***x***

– `4'b 1z0x === 4'b 1z0x` → ***1***

– `4'b 1z0x !== 4'b 1z0x` → ***0***

Conditional Operator

- `cond_expr ? true_expr : false_expr`
- Like a 2-to-1 mux ..




- Not to nest too deep, it becomes hard to read.

```
a = 'bx ? 8'b10101010 : 8'b10100101; // 8'b1010xxxx
```

Arithmetic Operators

- $+$, $-$, $*$, $/$, $\%$, $**$
- If arguments are n and m bits, bit-width of the result:
 - $+/-$: $\max(n, m) + 1$
 - $*$: $n + m$ (signed?: $n + m - 1$,)
- If any operand is x , then the result becomes x

Operator Precedence

| | |
|--------------------------------------|--|
| <code>+ - ! ~ unary</code> | highest precedence |
| <code>* / %</code> |  |
| <code>+ - (binary)</code> | |
| <code>< < > ></code> | |
| <code>< <= == > ></code> | |
| <code>== != === !==</code> | |
| <code>& ~ &</code> | |
| <code>^ ^~ ~^</code> | |
| <code> ~ </code> | |
| <code>& &</code> | |
| <code> </code> | |
| <code>?: conditional</code> | lowest precedence |

Use parentheses to
enforce your
priority

Module Design

Module Structure

```
`timescale 1ns/1ns
```

```
module module_name
```

```
  #(
```

```
    parameters
```

```
  )
```

```
  (
```

```
    ports declaration
```

```
  )
```

```
;
```

```
  wires and variables
```

```
  declaration
```

```
  continuous assignments
```

```
  module instantiations
```

```
  always @(*) blocks
```

```
  always @(clk edge) blocks
```

```
  initial blocks
```

```
  [ONLY in test-bench]
```

```
endmodule
```


Module Parameter

- Parameters are used to tune module properties, such as:
 - Port bus width,
 - Memory depth and width,
- They can have a default value at module level, and can be altered by top module via `#()` or `defparam` keywords,

Ports Declaration

```
`timescale 1ns/1ns

module adder_subtractor
#(
    parameter nb = 32
)
(
    input sub,
    input [nb-1:0] a,
    input [nb-1:0] b,
    output [nb-1:0] s,
    output c,
    output z,
    output n,
    output reg v
);
```

- Valid port directions are:
 - input,
 - output,
 - inout
- They are `wire` by default,
- Only output ports can be declared as a `reg`, if they are assigned inside an `always` block,
- Can be scalar or vector,

Ports Declaration, cont.

```
`timescale 1ns/1ns

module adder_subtractor
#(
    parameter nb = 32
)
(
    input sub,
    input [nb-1:0] a,
    input [nb-1:0] b,
    output [nb-1:0] s,
    output c,
    output z,
    output n,
    output reg v
);
```

```
`timescale 1ns/1ns

module adder_subtractor
#(
    parameter nb = 32
)
(
    input sub,
    input [nb-1:0] a, b,
    output [nb-1:0] s,
    output c, z, n,
    output reg v
);
```

*Not recommended,
single port per line, PLEASE.*

Ports Declaration, Old Syntax

```
`timescale 1ns/1ns

module adder_subtractor(sub, a, b, s, c, z, n, v);

    parameter nb = 32;

    input sub;
    input [nb-1:0] a;
    input [nb-1:0] b;
    output [nb-1:0] s;
    output c;
    output z;
    output n;
    output v;
    reg v;                                // or: output reg v;
```

Continuous Assignment

- It defines a permanent driver for a wire,
- The **assign** keyword is used for this purpose,
- Can be any arbitrary arithmetic or logical expression,
- They are executed in parallel, thus order does not matter,
- Wire declaration and assignment can be combined in a single statement,
- Better to order them in a more readable way,
- Logical loops are allowed by syntax, but are meaningless in hardware and can abnormally terminate simulation,
- Can define multiple drivers for a wire that are resolved in the background.
- A better and more readable way of gate level designs,
- Can incorporate a delay in timescale unit.

Continuous Assignment, Example

```
wire [nb-1:0] bb = sub ? ~b : b;  
// wire [nb-1:0] bb = b ^ {nb{sub}};  
  
assign {c, s} = a + bb + sub;  
  
assign z = s == 0;  
  
assign n = s[nb-1];  
  
// wire version, remove reg in declaration  
assign v = sub ?  
    ( a[nb-1] != b[nb-1] && a[nb-1] != s[nb-1] )  
    : ( a[nb-1] == b[nb-1] && a[nb-1] != s[nb-1] );
```

Continuous Assignment, Example

```
// `timescale 1ns/1ns
  `timescale 1ns/100ps

wire A, B, enA, enB, Y;

assign #5.7 Y = enA ? A : 1'bz;
assign #4.3 Y = enB ? B : 1'bz;

. . . . .

wire A, B, C, D, Y1, Y2, Y;

assign
    Y1 = A & B,
    Y2 = C & D,
    Y = Y1 | Y2;
```

Concurrency Implementation

- Each simulation time is divided into several ***simulation iteration/time steps*** or ***delta times***,
- Each wire or variable has two associated values:
1- Current, and 2- New,
- New values are evaluated if there is any change in their governing wires and/or variables, but current values are kept unchanged,
- In case of wires, multiple drivers are also taken into account and new values are defined based on the appropriate resolution function,
- In case of variables, if there are more than one new value, only the latest one is saved and all others are overwritten,
- New values are assigned as current values at the beginning of next step (delta time). This happens for all wires and variables altogether,
- Simulation moves to next time when all new values are equal to current values, thus nothing to be done for that simulation time.

Delta Time, Live Example

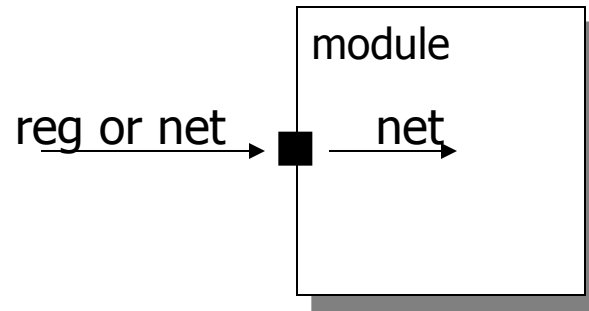
Module Instantiation

```
module_name
#(
    .parameter_name( value ),
    . . . . .
    .parameter_name( value )
)
    instance_name
(
    .port_name( connection ),
    . . . . .
    .port_name( connection )
);

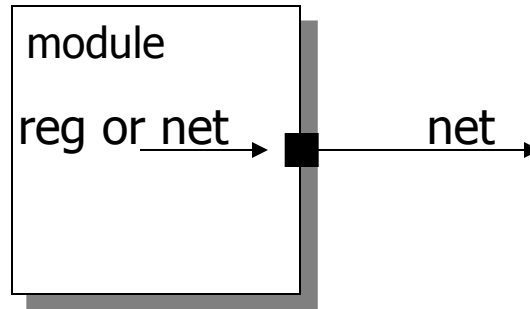
adder_subtractor
#(
    .nb( 24 )
)
    uut
(
    .sub( mode ),
    .a( a_in ),
    .b( b_in ),
    .s( s_out ),
    .c( c_out ),
    .z( z_out ),
    .n( ), // not connected
    .v( Overflow )
);
```

Port Binding

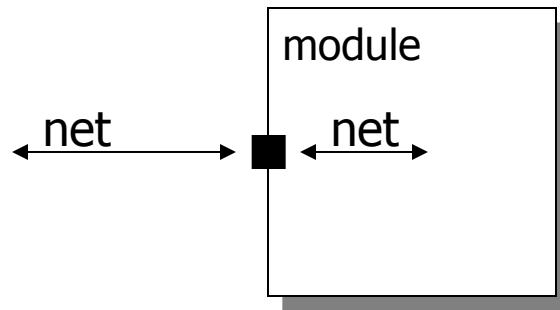
- Inputs



- Outputs



- Inouts



Port Binding, by Name, by Position

```
module adder_subtractor
#(
    parameter nb = 32
)
(
    input sub,
    input [nb-1:0] a,
    input [nb-1:0] b,
    output [nb-1:0] s,
    output c,
    output z,
    output n,
    output reg v
);
```

```
adder_subtractor
#(
    .nb( 24 )
)
    uut
(
    .sub( mode ),
    .a( a_in ),
    .b( b_in ),
    .s( s_out ),
    .c( c_out ),
    .z( z_out ),
    .n( ),
    .v( Overflow )
);
```

```
adder_subtractor
#(
    24
)
    uut
(
    mode,
    a_in,
    b_in,
    s_out,
    c_out,
    z_out,
    ,
    Overflow )
);
```

NOT Recommended

Built-In Primitives

- Built-in gate primitives:

`and, nand, nor, or, xor, xnor, buf, not, bufif0, bufif1, notif0, notif1, ...`

- Examples:

`nand (out, in1, in2);` 2-input NAND without delay

`and #2 (out, in1, in2, in3);` 3-input AND with 2 t.u. delay

`not #1 N1(out, in);` NOT with 1 t.u. delay and instance name

`xor X1(out, in1, in2);` 2-input XOR with instance name

- User defined primitives (UDP) can be defined and used. This is very useful for technology library definition.
- Continuous assignment is preferred for modeling of simple gates, use primitives only when necessary.

Hierarchical Names

- At top module, i.e. usually test bench, wires and variables of sub-modules can be accessed via:

```
instance_name.[instance_name.]signal_name
```

- This can save time declaring extra wires for reading or interconnecting sub-modules,
- An easy way to put monitors on too deep sub-modules at the top level.
- If an initialization is required, e.g. for a free running counter, then this should be done only on top module using hierarchical names.

Initial Blocks

- Execution of each block starts at simulation beginning (sim-time = 0) and finishes when the last statement is executed,
- Execution inside each initial block is in a sequential order, like a C code, and takes zero simulation time,
- A module can have several initial blocks (alongside other elements) all of which are alive in parallel. However, execution of them are in an unknown order.
- Exclusively used in test-benches for:
 - Pattern generation,
 - Input feeding,
 - Output verification,
 - File access,
 - . . .
- Does not have any real hardware correspondent, thus should not be used in a design,
- Can assign values to only variables, but not nets,
- Use = (blocking) for variable assignments inside it,

```

`timescale 1ns/1ns

module add_sub_tb;

    parameter num_tests = 20;

    reg s;
    integer i;
    wire signed [7:0] z;
    reg signed [7:0] x, y;

    initial
        for(i = 0; i < num_tests; i = i + 1) begin
            x = $random;
            y = $random;
            s = $random;

            #1;

            if(s)
                $display("0x%x (%d) - 0x%x (%d) = 0x%x (%d), %0s",
                    x, x, y, y, z, z, !uut.v ? "OK" : "Overflown");
            else
                $display("0x%x (%d) + 0x%x (%d) = 0x%x (%d), %0s",
                    x, x, y, y, z, z, !uut.v ? "OK" : "Overflown");

            #9;
        end

    adder_subtractor #(8) uut (
        .sub( s ), .a( x ), .b( y ), .s( z ), .c( ), .z( ), .n( ), .v( ) );

endmodule

```



```

`timescale 1ns/1ns

module add_sub_tb;

    parameter num_tests = 20;

    reg s;
    integer i;
    wire signed [7:0] z;
    reg signed [7:0] x, y;

    initial
        for(i=0; i<num_tests; i=i+1) begin
            x = $random; y = $random; s = $random;
            #10;
        end

    initial
        for(i=0; i<num_tests; i=i+1) begin
            #1;

            if(s)
                $display("0x%x (%d) - 0x%x (%d) = 0x%x (%d), %0s",
                    x, x, y, y, z, z, !uut.v ? "OK" : "Overflown");
            else
                $display("0x%x (%d) + 0x%x (%d) = 0x%x (%d), %0s",
                    x, x, y, y, z, z, !uut.v ? "OK" : "Overflown");

            #9;
        end

    adder_subtractor #(.nb( 8 )) uut (
        .sub( s ), .a( x ), .b( y ), .s( z ), .c( ), .z( ), .n( ), .v( ));

endmodule

```

```

`timescale 1ns/1ns

module add_sub_tb;

    parameter num_tests = 20;

    reg s;
    integer i;
    wire signed [7:0] z;
    reg signed [7:0] x, y;

    initial
        for(i=0; i<num_tests; i=i+1) begin
            x = $random; y = $random; s = $random;
            #10;
        end

    always @( z ) begin
        #1;
        if(s)
            $display("0x%x (%d) - 0x%x (%d) = 0x%x (%d), %0s",
                x, x, y, y, z, z, !uut.v ? "OK" : "Overflown");
        else
            $display("0x%x (%d) + 0x%x (%d) = 0x%x (%d), %0s",
                x, x, y, y, z, z, !uut.v ? "OK" : "Overflown");
    end

    adder_subtractor #(.nb( 8 )) uut (
        .sub( s ), .a( x ), .b( y ), .s( z ), .c( ), .z( ), .n( ), .v( ));

endmodule

```

Initial Blocks, Revisited

- A module can have several initial blocks (alongside other elements) all of which are alive in parallel. However, execution of them are in an unknown (simulator dependent) order.

```
module test;
```

```
    reg a;
```

```
    initial  
        $display(a);
```

```
    initial  
        a = 0;
```

```
    initial  
        a = 1;
```

```
endmodule
```

```
module test;
```

```
    reg a = 1'bz;
```

```
    initial  
        $display(a);
```

```
    initial  
        a = 0;
```

```
    initial  
        a = 1;
```

```
endmodule
```

```
module test;
```

```
    reg a = 1'bz;
```

```
    initial  
        $display(a);
```

```
endmodule
```

Blocking (=) & Non-Blocking (<=) Assignment

- An assignment to a variable inside a sequential block can be done in two forms:
 - Blocking by "="
 - Non-blocking by "<="
- Consider zero-intra-assignment delay case:
- "=" changes current value of the variable immediately.
- "<=" changes the new value of the variable and does not touch the current value. This is copied to the current value at the end of current iteration step.
- Reminder: in both cases, when there are more than one assignment to a single variable, the last one takes place and the rest are ignored (overwritten)

"=" & "<=" Assignment, Example

```
integer a, b;

initial begin
    a = 2;
    b = 3;
    $display("%0d, %0d", a, b);           // 2 3

    a <= b;
    b <= a;
    $display("%0d, %0d", a, b);           // 2 3

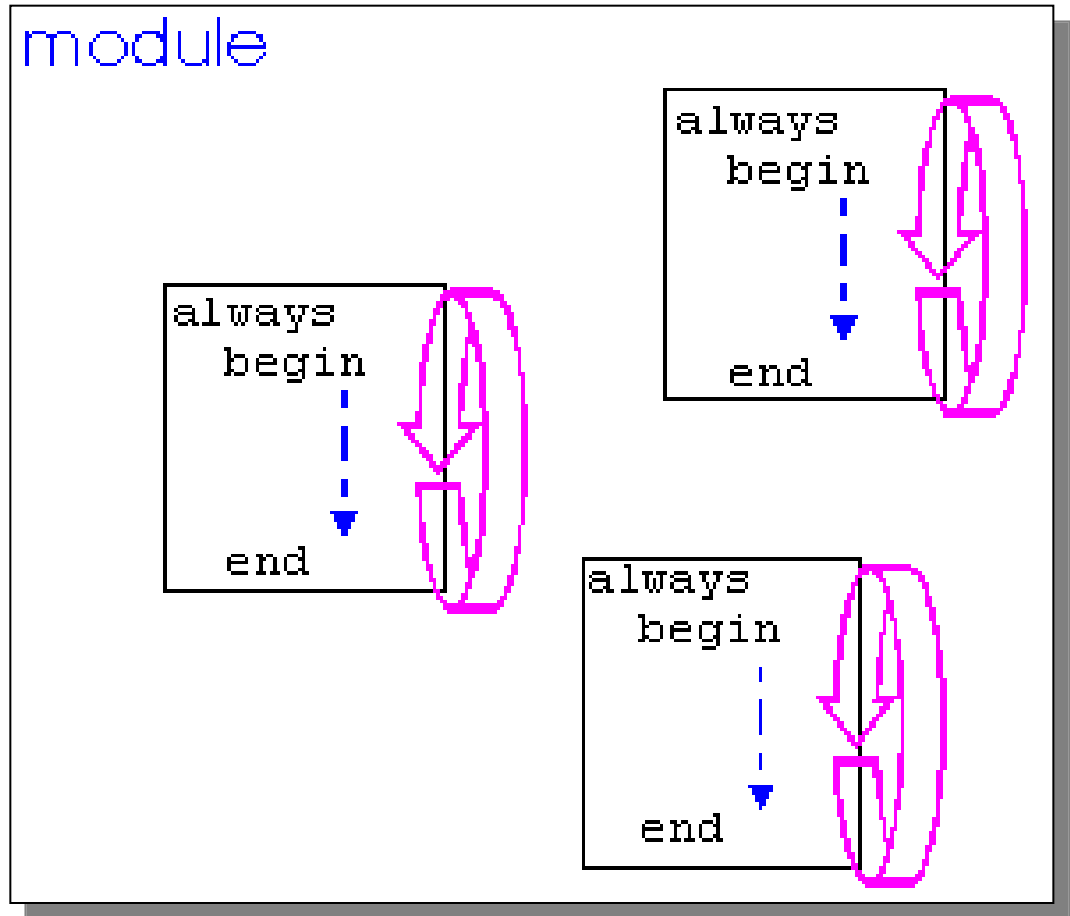
    #1;
    $display("%0d, %0d", a, b);           // 3 2

    b <= a;
    b <= a * a;                          // 2nd one overwrites the 1st one
    a = b;                                // = assignment is done immediately
    $display("%0d, %0d", a, b);           // 2 2

    #1;
    $display("%0d, %0d", a, b);           // 2 9
end
```

Always Blocks

- Start execution at simulation time zero and continue until simulation finishes.
- All always blocks are alive in parallel, alongside other concurrent elements, such as continuous assignments.
- Even though syntax allows to have zero delay blocks, such blocks can easily cause simulation crash. Thus, each block should contain delay, or be controlled by an event.



Events

- @

```
always @( signal1 or signal2 or .. ) begin
    ..
end
```

execution triggers every time
any of the signals listed in
sensitivity list changes

```
always @( posedge / negedge clk ) begin
    ..
end
```

execution triggers every time
clk changes from *0/x/z* to *1*
or *1/x/z* to *0*

```
always @( * ) begin
    ..
end
```

execution triggers when any
of the read signals inside the
block changes

Clock Generation

```
reg clk = 1'b1;  
always @( clk ) clk <= #5 ~clk;
```

- It is a clock with a period of 10 time-scale (e.g. ns). All positive edges are aligned at multiple of 10's.
- It is used in a test-bench, in a real design, you need a real clock generator, not this !!

Always @(*)

- Always sequential body is executed when any of the wires or variables that are read inside block statements is changed.
- Read signals/variables may appear in:
 - Right hand side (RHS) of an assignment,
 - Condition of an if-statement,
 - Argument of a case-statement,
 - etc. etc.

Always @(*), Example

```
// assign v = sub ?  
//      ( a[nb-1] != b[nb-1] && a[nb-1] != s[nb-1] )  
//      : ( a[nb-1] == b[nb-1] && a[nb-1] != s[nb-1] );  
  
always @ ( * ) begin  
    v = 0;  
    if( sub == 0 ) begin                // this is an add  
        if( a[nb-1] == b[nb-1] )  
            if( s[nb-1] != a[nb-1] )  
                v = 1;                // overflow occurred  
    end  
    else                                // this is a sub  
        if( a[nb-1] != b[nb-1] )  
            if( s[nb-1] != a[nb-1] )  
                v = 1;                // overflow occurred  
end
```

Combinational Logic Modeling

- In order to model a combinational logic by means of sequential codes, rules are:
 - Use “always @(*)” syntax,
 - Use “=” (blocking) assignment without any delay,
 - Make sure output (LHS variable) is assigned in all conditional branches (if- and case-statements)
 - To ensure above, and better readability, assign a default value at the always block beginning.
 - Do not use the output in any assignments. That results in an invalid logical loop.
- Reminder: *reg* is not always a reg!

Procedural Flow Control: if

```
if (expr1)
    true_stmt1;
...
else if (expr2)
    true_stmt2;
...
else
    false_stmt;

always @( * )
    out = in[ sel ];
```

```
module mux4_1(
    output reg out,
    input [3:0] in,
    input [1:0] sel
);

always @( * )
    if (sel == 0)
        out = in[0];
    else if (sel == 1)
        out = in[1];
    else if (sel == 2)
        out = in[2];
    else
        out = in[3];

endmodule
```

Procedural Flow Control: case

case (expr)

item_1, .., item_n: stmt1;

item_n+1, .., item_m: stmt2;

...

default: def_stmt;

endcase

```
module mux4_1(  
    output reg out,  
    input [3:0] in,  
    input [1:0] sel  
);  
  
always @( * )  
    case (sel)  
        0: out = in[0];  
        1: out = in[1];  
        2: out = in[2];  
        3: out = in[3];  
    endcase  
endmodule
```

Procedural Loops

for (init_assignment; condition; step_assignment) // ++ and -- do not exist
 loop_statement;

while (condition)
 loop_statement;

repeat (no_of_times)
 loop_statement;

forever
 loop_statement;

loop_statement: a single line statement, or several lines grouped w/ begin-end

Unwanted Latch Inference

- When a variable is not assigned at least in one of the branch conditions, it means that the model is silent with regards to that variable in that specific circumstance.
- Since variables hold their values as long as they are not changed, above scenario means that variable value should not be changed in that specific situation.
- This models a latch, doesn't it?

Unwanted Latch Inference, Example

```
always @( * )
  case( op )
    2'b000: X = A + B;
    2'b001: X = A - B;
    2'b010: X = A & B;
    2'b100: X = A | B;
    2'b101: X = ~ A;
  endcase
```

When op is 2'b011, 2'b110 or 2'b111, X will hold its previous value, which is a latch behavior, not a combinational logic, even though op equal to those values do not happen at all.

```
always @( * )
  case( op )
    2'b000: X = A + B;
    2'b001: X = A - B;
    2'b010: X = A & B;
    2'b100: X = A | B;
    2'b101: X = ~ A;
    default: X = 'bx;
  endcase
```

```
always @( * ) begin
  X = 'bx;
  case( op )
    2'b000: X = A + B;
    2'b001: X = A - B;
    2'b010: X = A & B;
    2'b100: X = A | B;
    2'b101: X = ~ A;
  endcase
end
```


Always @(clk edge)

- How does a D-type flip-flop work? It waits for the rising edge of the clock, then the output (q) gets the value of input (d) after a delay.
- Here is the scenario in Verilog:

```
always @(posedge clk)
    q <= #1 d;
```

- It can be a large 64-bit register, a counter, shift register, etc. etc.

Always @(clk edge), Fully Synchronous Example

```
`timescale 1ns/1ns

module multi_func_reg(
    input clk,
    input reset,
    input up,
    input down,
    input x5,
    input x7,
    input load,
    input [31:0] data,
    output reg [31:0] q
);

    always @(posedge clk)
        if(reset)
            q <= #1 32'h00000000;
        else if(load)
            q <= #1 data;
        else if(up && !down)
            q <= #1 q + 1'b1;
        else if(down && !up)
            q <= #1 q - 1'b1;
        else if(x5)
            q <= #1 (q << 2) + q;
        else if(x7)
            q <= #1 (q << 3) - q;

endmodule
```

Always @(clk edge), Async. Set & Reset

- A clock edge triggered flip-flop or register can only asynchronously be set and/or reset, but nothing else, i.e. don't expect any async. functionality from them.
- The scenario would be: wait for clock edge to copy d to q, or set/reset to change output accordingly. Here is the code:

```
always @(posedge clk, posedge set, posedge reset)
    if(reset)                // highest priority
        q <= 1'b0;
    else if(set)
        q <= 1'b1;
    else
        q <= #1 d;
```


Module Structure

```
`timescale 1ns/1ns
```

```
module module_name
```

```
  #(
```

```
    parameters
```

```
  )
```

```
  (
```

```
    ports declaration
```

```
  )
```

```
;
```

```
  wires and variables
```

```
  declaration
```

```
    continuous assignments
```

```
    module instantiations
```

```
    always @(*) blocks
```

```
    always @(clk edge) blocks
```

```
    initial blocks
```

```
      [ONLY in test-bench]
```

```
endmodule
```


System Tasks

Always written inside sequential part of the test-bench

- `$display("..", arg2, arg3, ..);` → much like `printf()`, displays formatted string in std output when encountered
- `$monitor("..", arg2, arg3, ..);` → like `$display()`, but `..` displays string each time any of `arg2, arg3, ..` Changes
- `$stop;` → suspends simulation when encountered
- `$finish;` → finishes simulation when encountered
- `$fopen("filename");` → returns file descriptor (integer); then, you can use `$fdisplay(fd, "..", arg2, arg3, ..);` or `$fmonitor(fd, "..", arg2, arg3, ..);` to write to file
- `$fclose(fd);` → closes file
- `$random(seed);` → returns random integer; give her an integer as a seed

\$display & \$monitor string format

| Format | Display |
|----------|---|
| %d or %D | Display variable in decimal |
| %b or %B | Display variable in binary |
| %s or %S | Display string |
| %h or %H | Display variable in hex |
| %c or %C | Display ASCII character |
| %m or %M | Display hierarchical name |
| %v or %V | Display strength |
| %o or %O | Display variable in octal |
| %t or %T | Display in current time format |
| %e or %E | Display real number in scientific format |
| %f or %F | Display real number in decimal format |
| %g or %G | Display scientific or decimal, whichever is shorter |

Acknowledgement

- Several slides are taken from:
www.csd.uoc.gr/~hy225/verowell/verilog_basics.ppt ,
By: Thanasis Oikonomou, Oct. 1998.