Working with Operators, Operands, Variables, and Literals

CMPE 415 Programmable Logic Devices

Prof. Ryan Robucci

UMBC

DEPARTMENT OF COMPUTER SCIENCE AND ELECTRICAL ENGINEERING

References

- Verilog IEEE 2001 Specification
- Deepak Kumar Tala, Asic-World
  - http://www.asic-world.com/verilog/operators1.html
  - http://www.asic-world.com/verilog/operators2.html

Data Types: Nets and Variables

Nets: Provide structural connectivity, its value is determined by connection or continuous assignment

Variables: Provide connectivity as well as ability to be assigned in a procedural block

- reg: stores logic value in simulation, may represent output of either combinatorial or sequential hardware
- integer: supports computation but avoid use as hardware a variable
- real: like integer but in simulation stores values as real numbers
- time: stores time as u64
- realtime: stores time as real

- Variables where originally called registers, but since they are not actually always implemented as registers in hardware the name was changed for Verilog 2001 to variable. Be aware when reading older texts.

reg

- A variable type which can be used to abstract the hardware storage element since it remember values set in procedural code or sequential primitives –
  but it may also represent a node in a hardware circuit who's value is continuously set by combinatorial logic

- It is assigned a value in Verilog by one of
  - procedural statement
  - user-defined seq. primitive
  - task or function

- May not connect to output (or inout) ports of instantiated modules or Verilog primitives
- May not be assigned value using a continuous assignment

reg vectors

- reg variables are commonly used as vectors.
  reg [msbindex:lsbindex] <varname>;
  indexes may be negative

Two vector declarations with different bit-significance ordering.
msb-first and lsb-first

```
reg [31:0] regA;
reg [0:31] regB;
```

The most-significant bits are implied to be regA[31] and regB[0]

```
regA = regB;
```

sets regA[31] to regB[0]
regA[30] to regB[1] ….
regA[0] to regB[31]

Bits may be <u>read</u> and <u>assigned</u> one at a time using the indexing operator []:

```
regA[31] = regA[0];
```

reg vectors

Multiple bits may be read and assigned using the indexing operator []:

```
regA[31:24] = regA[7:0];
```

Multiple bits may be read and assigned using the indexed part-select syntax:

```
regA[ 0 +: 8] // regA[ 7: 0]
regA[15 -: 8] // regA[15: 8]
regB[ 7 -: 8] // regB[ 7: 0]
regB[ 8 +: 8] // regB[ 8: 15]
```

```
regA[ 0 +: 8] = regB[ 7 -: 8]
```

Is the same as `regA[ 7 : 0] = regB[7 : 0];`

Signed reg vectors and wire vectors*

- reg variables and wire nets may be signed. These support signed operations like integers (integers are signed by default).
- signed reg [msb:lsb] <varname>;

  signed wire [msb:lsb] <varname>;

Ex:

```
signed reg [7:0] regA;
signed wire [7:0] wireA;
```

These support values -128 to 127 and the <u>signed modifier determines different behaviors for some operations</u> that support signed numbers.

Convention for the slides herein*

Slides in this class may include several examples where we won't want to see all declarations crowding the slides.

Assume prefixes ur1,sr8, ur8,ur16,ur32, etc. denote unsigned variables of obvious lengths 8,16,32, etc.
Assume prefixes sr1,sr4,sr8,sr16,sr32, etc. denote signed variables of obvious lengths 1,4,8,16,32, etc.
Examples:

```
    reg [7:0] ur8a;
    signed reg [15:0] sr16a;
```

Assume prefixes uw1,sw8, uw8,uw16,uw32, etc. denote unsigned nets of obvious lengths 8,16,32, etc.
Assume prefixes sw1,sw4,sw8,sw16,sw32, etc. denote signed nets of obvious lengths 1,4,8,16,32, etc.
Examples:

```
    wire [7:0] uw8a;
    signed reg [15:0] sr16a;
```

Integers*

- Generally act as a 32-bit signed value, but can be larger
- If an integer is used where a reg[31:0] is required, providing the integer can cause an implicit casting to unsigned – this interpretation of a signed storage as an unsigned can be a disaster.
- Example declaration:

```
  integer int1,int2;
  integer intX=100;
  integer intY=10;
```

Use Integers a throw away variable*

- In general you may use integers for named constants and as temporary thow-away variables in your code where computations assigned to integers are **computed strictly at synthesis**
- They are dangerous to use as register outputs (for which we assign using <= . Caution for other use is required.

```
integer intX=100;
integer intY=10;
integer intO=5;
integer intZ;
reg [31:0] u32q;

always @(posedge clk)
   intZ = intX/intY;    ← Can be precomputed
   u32q <= u32q + intZ; ← final result is reg and designer
end                       determined that its size will hold the
                          value in intZ this time
```

Unsigned Sized Literals*

- When specifying numerical literals, a negative sign can denote a **two's compliment operation** (one's compliment then add 1) **on an unsigned sized value** and still **result in an unsigned value** (a value that operands treat as unsigned)
  - Ex:-'d8 is a 32-bit unsigned value storing a signed representation of (-8)
    This means that operations don't know to use signed arithmetic on the result, even though we might like to think of the bits as representing an signed representation
    - (-'d8)/2 does not produce “-4”
      since (-'d8) is unsigned and thus unsigned division is performed which does not preserve sign
      (see next slide where the msb becomes 0)
    - -('d8)/2 does produce “-4”

Integer and Unsigned Sized Literals*

```
integer int1,int2,int3,int4,int5,int6,int7;
initial begin
  int1=  -8;      // 1111 1111 1111 1111 1111 1111 1111 1000
  int2=  -'d8;    // 1111 1111 1111 1111 1111 1111 1111 1000
  int3= -'d8/2;   // 0111 1111 1111 1111 1111 1111 1111 1100
  int4=  -8/2;    // 1111 1111 1111 1111 1111 1111 1111 1100
  int5= int1/2;   // 1111 1111 1111 1111 1111 1111 1111 1100
  int6= int2/2;   // 1111 1111 1111 1111 1111 1111 1111 1100
  int7= -('d8/2);// 1111 1111 1111 1111 1111 1111 1111 1100
  int8= (-'d8)/2;// 0111 1111 1111 1111 1111 1111 1111 1100
end
```

Signed Integer and Sized Literals*

- One solution is the new signed sized literals using s in front of the format specifier to provide a signed type

- Ex:-'sd8 is a 32-bit signed value storing a signed representation of (-8)
  This means that operations do know to use signed arithmetic
  - (-'sd8)/2 does produce “-4”
    since (-'sd8) is signed and thus signed division is performed which does preserve sign
  - -('sd8)/2 does produce “-4”
- Ex:

```
16'sh00A0; //16-bit signed +10
-16'sh00A0; //16-bit signed -10
16'shFFFF; //16-bit signed -1
```

Signed Integer and Sized Literals*

```
integer int1,int2,int3,int4,int5,int6,int7;
initial begin
  int1=  -8;       // 1111 1111 1111 1111 1111 1111 1111 1000
  int2=  -'sd8;    // 1111 1111 1111 1111 1111 1111 1111 1000
  int3= -'sd8/2;   // 1111 1111 1111 1111 1111 1111 1111 1100
  int4=  -8/2;     // 1111 1111 1111 1111 1111 1111 1111 1100
  int5= int1/2;    // 1111 1111 1111 1111 1111 1111 1111 1100
  int6= int2/2;    // 1111 1111 1111 1111 1111 1111 1111 1100
  int7= -('sd8/2);// 1111 1111 1111 1111 1111 1111 1111 1100
  int8= (-'sd8)/2;// 1111 1111 1111 1111 1111 1111 1111 1100
end
```

Reals (*avoid use)

- Reals generally treated as double, 64 bit elements
- Literals may be provides in exponential (e.g. 1.2e-3) or basic decimal format (.34)
- No bit access
- No direct passing through ports of primitives or modules
- Use $realtobits,$bitstoreal, $rtoi (truncate), and $itor to convert to 64-bit binary and integers
  - May be used to pass reals through 64-bit ports that are declared as follows:

```
module (output wire [63:0] realIn,
        input wire [63:0] realOut));
real tmp;
tmp = $bitstoreal(realIn);
realOut = $realtobits(tmp);
```

time, realtime, and `timescale *

- time is a u64
- realtime is a real
- Like integer and real, these may not be used in a module port or be an output or input of a primitive
- Time is defined according to the time_unit and base provided by

  `` `timescale time_unit base / precision base ``

  - time_unit is the time step represented by **#1.0**
  - base is `s`, `ms`, `us`, `ns`, `ps`, or `fs`
  - precision is the rounding precision of the simulation
  - Example:

    `` `timescale 1ns/1ps ``

    sets #1.0 to mean 1 ns delay

Checking `timescale

- If you want to print the timescale used in your simulation, you can print it using the following in your testbench or in other files.

```
initial  $printtimescale;
```

Prints the following when place in module DUT_tb:

```
Timescale of (DUT_tb) is
1ns/1ps.
```

- Checking other modules in the hierarchy can be done by providing the hierarchical name of the module:

```
initial  $printtimescale(DUT_tb.DUT.I4);
```

Time-related system tasks and functions*

- Related system functions:
    - **$time** is a system function which provides an integer
    - **$stime** provides a truncated 32-bit time
    - **$realtime** provides a real (64-bit) time according
- In a format string for a system task like **$display**, use "%0t" to print time:
    - **$display**("time is %0t",**$time**);

    %t follows a <u>global default format</u> that which can be redefined using the following system task:

    $timeformat(unit, precision, suffix, min_field_width);
    - unit is a value from 0 to -15 representing seconds to femtoseconds
    - precision represents the number of decimal points to display
    - suffix is a string to append to what is printed (e.g. "ns")
    - min_field_width represents the minimum number of characters to use to print time

Arrays

- A 2D bit memory can be defined with reg array
  Example:
  ```
  reg [31:0] cache_memory [1023:0]
  ```
  Defines 1024 32-bit entries
- Integer(32-bit) array
  ```
  integer cache_memory [1023:0];
  ```
- Real array
  ```
  real sin_table[1023:0];
  ```
- An array of time can be created:
  ```
  time T[1:00];
  ```
- Multi-dimensional arrays are allowed
  - Example:
    - `reg [31:0] cache_memory [15:0][1023:0]`
      16 banks of 1024 words of length 32 bits each,
    - `cache_memory [15][1023][0]` is the lsb of the last word

Strings*

- no built in string/char type
  - just reserve 8 bits per char in reg declaration

*string is added in SystemVerilog

```
parameter numchars = 7;
reg [((8*numchars)-1):0] my_string;
my_string = “Hi”;
```

Result is padded with 0s on left and no trailing 0/null:

my_string = 0 0 0 0 0 'H' 'I'

Constants may be stored and recalled with parameters, localparam

Parameters can be defined using keyword localparam
These are good way to store named constants (magic numbers), but be mindful of the automatic type being assigned.

```
localparam a=31; //int
localparam a=32,b=31; //int,int
localparam byte_size=8, byte_max=bytesize-1; //int
localparam a =6.22; //real
localparam delay = (min_delay + max_delay) /2 //real
localparam initial_state = 8'b1001_0110; //reg
```

Arithmetic Operators

- Binary Arithmetic Operators:
  - **+,-,***
  - **/,%** (modulus)
    - Integer division is defined by fractional truncation, round towards zero
    - Divide by zero (/ or %) results in unknown (x)
    - Modulus result takes sign of the first operand
  - ** (power)
    - Unspecified if either:
      - First operand is zero and the second is nonpositive
      - First operand is negative and second is not an integral value (e.g. reg, int)
- Unary Arithmetic Operators:
  - **+,-** (e.g. -x)
- For all arithmetic Operators, if any bit of any operand is **x** or **z**, the entire output result is unknown (**x**)

Unary Reduction Operators*

- Unary Reduction operators
  - operate on one operand, produce single bit result
  - Operations with x and z bits may be **resolvable** (e.g x and 0 = 0)

Symbol	Operator
~&	and
|	or
~|	nor
^	xor
~^,^~	xnor

```
& 2'b1z      →  x
& 2'b1x      →  x
& 2'b0z      →  0
& 2'b0x      →  0
& 0          →  0
& 1          →  0
& -1         →  1
```

```
&(010101) → 0 Reduction And
|(010101) → 1 Reduction Or

&(010x10) → 0 Reduction And
|(010x10)  → 1 Reduction Or
```

Logical Operators *

Symbol	Operator
!	negation
&&,||	and,or
==,!=	Logical equality
===,!==	Case equality

•===,!== compare bit matching including x and z
•==,!= produce x if any x or z exists
•left zero padding performed as needed
•Signed and unsigned integers and reals are converted to binary and treated as unsigned
•Multibit Logical Arguments are True if they contain any bit that is a 1, since any 1 means the value is non-zero (e.g. 2’b1x is True)
•Logical Arguments that contain x/z bit without a high bit elsewhere are ambiguous (e.g. it is ambiguous if 2’b0x is non-zero)

- &&
  - A True result requires two non-zero arguments
  - A False result requires two zero arguments
  - A Undermined result (1’bx) occurs if one of the arguments contains x/z bit(s) without any high bit
- ||
  - A True result requires one non-zero argument
  - A False result requires two zero arguments
  - A Undermined result (1’bx) occurs if
    - one argument is 0 and the other contains x/z bit(s) without any high bit
    - both arguments are such that they contain x/z bit(s) without any high bit
- What about a non-valid bits?
  - (3'b110 && 3'b11x) → (True && True) → True=1
  - (3'b110 & 3'b11x) → (3'b110) → TRUE=1
  - (3'b100 && 3'b00x) → (Tue && Undetermined) → Undetermined=x
  - (3'b100 & 3'b00x) → (3'b000) → False=0

*Use case equality for testbenches but not RTL

Operator	Description
a === b	a equal to b, including x and z (Case equality)
a !== b	a not equal to b, including x and z (Case inequality)
a == b	a equal to b, result may be unknown (logical equality)
a != b	a not equal to b, result may be unknown (logical equality)

- 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

Note : The result is always 0 or 1 unless result is unknown and x results

Equality Examples

```
module equality;

initial begin
   if (2'bx0== 2'bx0) $display("True"); else $display("False");     False
   if (2'bx0===2'bx0) $display("True"); else $display("False");     True
   if (2'bz0== 2'bz0) $display("True"); else $display("False");     False
   if (2'bz0===2'bz0) $display("True"); else $display("False");     True
   if (2'bzx== 2'bzx) $display("True"); else $display("False");     False
   if (2'bzx===2'bzx) $display("True"); else $display("False");     True
   if (2'bxz===2'bzx) $display("True"); else $display("False");     False
end

endmodule // equality
```

Bit-Wise Operators*

- Bit-wise operators on pairs of operands
  - output same size as inputs
  - shorter binary operands are zero-padded on the left to the longer
  - these operators imply gates

Symbol	Operator
~	not/compliment
&	and
|	or
^	xor
~^	xnor

```
  ~4'bxz01
  →    xx10

   6'bxxzz0011
&  6'b01010101
 →   0x0x0001

   6'bxxzz0011
|  6'b01010101
 →   x1x10111

   6'bxxzz0011
^  6'b01010101
 →   xxxx0110
```

- Note x (z is treated same as x) is not always "contagious"
  if the operation makes the result unambiguous
  - x&1'b0 → 0 and x|1'b1 → 1

Logical Operators*

Operator	Description
!	Logical negation
&&	Logical and
||	Logical or

1'b1 && 1'b1 → 1
1'b1 && 1'b0 → 0
1'b1 && 1'bx → x
1'b0 && 1'bx → 0
1'b1 || 1'b0 → 1
1'b0 || 1'b0 → 0
1'b0 || 1'bx → x
1'b1 || 1'bx → 1

- Expressions connected by && and || are evaluated from left to right
  (con1 && cond2 || cond3)
- Evaluation stops as soon as the result is known <u>(short-circuit evaluation)</u>
- The result is a scalar value:
  - 0 if the relation is definitely false
  - 1 if the relation is definitely true
  - **<u>x one of the operands is x and the result is undetermined by the operator and the other operand</u>**

Logic Negation, !Op

- Creates inverse of logic result
- For binary vectors, the operand Op must be converted to a "boolean" value OpLogic=1/0/x
  The process can be described like a statemachine involving an intermediate result OpLogic and a scan position
  - Start by assuming OpLogic=0
  - Search Operand vector left to right
    - if 0 found, don't modify OpLogic, move inspection right
    - if 1 found, stop evaluation, operand is definitely True (OpLogic=1) and so result of logic 0
    - if x or z found, change OpLogic=x for now, continue inspection to the right
    - if end found,
      - if OpLogic is 1, result is 0
      - If OpLogic is x, result is x
- For integers,
  returns 1 for 0 and
  0 otherwise

```
! 1'b1      = 0
! 1'b0      = 1
! 1'bz      = x
! 1'bx      = x
! 2'b00     = 1
! 2'b01     = 0
! 2'b10     = 0
! 2'b11     = 0
! 2'b1z     = 0
! 2'b1x     = 0
! 2'b0z     = x
! 2'b0x     = x
! 0         = 1
! 1         = 0
! (-1)      = 0
```

Cast to Boolean

True if certainly non-zero, sure that a 1 exists

```
if (u2b) begin
  $display("True");
end else begin
  $display("Not True"); //though perhaps not false
end
```

u2b=2'b00	**Not True**
u2b=2'bxx	**Not True**
u2b=2'bx0	**Not True**
u2b=2'b0x	**Not True**
u2b=2'bz0	**Not True**
u2b=2'bz1	True
u2b=2'b1z	True

Relational Operators*

- Nets and registers treated as unsigned words
- If any bit is unknown, the relation is unknown and result is ambiguous x

Symbol	Operator
<	Less than
<=	Less than equal to
>	Greater than
>=	Greater than or equal to

When one or both operands of a relational expression are unsigned, the expression shall be interpreted as a comparison between unsigned values. If the operands are of unequal bit lengths, the **smaller operand shall be zero-extended** to the size of the larger operand. --IEEE Spec

Implicit Casting Surprises*

- Mix of unsigned and signed results in unsigned cast examples for W=32

Constant	Constant	Relation	Evaluation
u8a=0	$unsigned(0)	==	unsigned
s8a=-1	0	<	signed
s8a=-1	$unsigned(0)	>	unsigned
s8a=127	s8b = -127-1	>	signed
u8a=$unsigned(127)	s8b = -127-1	<	unsigned
s8a=-1	-2	>	signed
u8a=$unsigned(-1)	-2	>	unsigned
s8a=127	$unsigned(255)	<	unsigned
s8a=127	s8a=$unsigned(255)	>	signed

Logical Shift Operators

- Accepts two unsigned binary word operands
- integers are converted to two-complement binary equivalents and treated as unsigned (bit shift with 0-filling)

Symbol	Operator
<<	Left Shift
>>	Right Shift

```
module tb();


   reg [7:0] u8a;
   reg signed [7:0] s8b;

  initial begin
    #0
    u8a = -1;
    s8b = -1;
    $display("a = %b, b = %b",u8a,s8b);
    u8a = u8a >> 1;
    s8b = s8b >> 1;
    $display("a = %b, b = %b",u8a,s8b);
  end

endmodule // tb
```

```
a = 11111111, b = 11111111

a = 01111111, b = 01111111
```

Conditional ? Operator*

- Syntax:
  - Conditional_expression := selection_expression ? true_expression : false_expression
- Can use in continuous assign statements and in procedural blocks

```
Y=(sel)? A:B;
Y = (c>threshold)? A:B;
Y=(A>B)? A:B;  //max of A and B
```

Like **if**, **?** is conceptually a mux;
Note: z CAN be passed through <u>expressions</u> to the output if selection expression is unambiguous 1 or 0

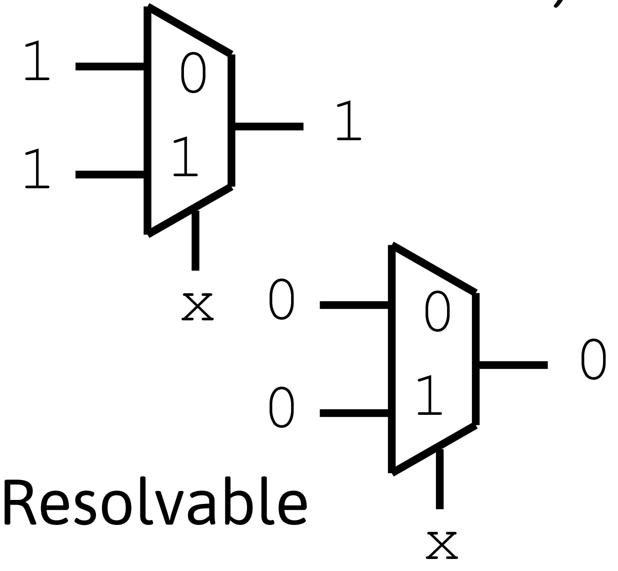
```
Y=(en)?data:16'bz;
```

en=x gives unknown but
en=1 will pass data and
en=0 will pass multiple z bits

This is sometimes the recommended coding for synthesizing a **tristate buffer**.

Conditional ? Operator

- Z not allowed in conditional_expression
- Zeros are automatically filled for different length operands
- If conditional_expression is ambiguous, both true_ex and false_ex are evaluated and result is calculated on a bitwise basis according to truth table below (same as collision of wires, exists since section doesn't matter if both bits are same)



Resolvable

1'bx?a:b	a=0	a=1	a=x
b=0	**0**	x	x
b=1	x	**1**	x
b=x	x	x	x

Concatenation and Repetition Operator

- Concatenation operator can create a single word from two or more operands
- Useful for forming logic buses
- Concatenation follows order given
- Nesting is allowed

```
{2b'01,3'b010} → 5'b01010
{4{2'b01}} → {2'b01,2'b01,2'b01,2'b01} → 8'b01010101
{4'b0001,{{2'b01},{2'b10}}} → 8'b00010110
```

No operand may be an unsigned constant since compiler would not be able to size the result

<u>Repetition operator</u> should be used with a constant repetitions: <repetitions>{<data>}

```
4{2b'01} → 8'b01010101
```

Operator Precedence

Precedence	Operator	Symbols
First	Unary	+ - ! ~
	Multiplication, Division, Modulus	* / %
	Add, Subtract	+ -
	Shift	<< >> (<<< >>>)
	Relational	< <= > >=
		== != === !==
Last	Conditional	? :

Literals (Unsigned)

Number	#Bits	Base	Dec. Equiv.	Stored
2'bl0	2	Binary	2	10
3'd5	3	Decimal	5	101
3'o5	3	Octal	5	101
8'o5	s	Octal	5	00000101
8'ha	8	Hex	10	00001010
3'b5	Not Valid!			
3'b0lx	3	Binary	-	01 x
12'hx	12	Hex	-	xxxxxxxxxxxxx
8'hz	S	Hex	-	zzzzzzzz
8'b0000_0001	8	Binary	1	00000001
8'b00l	8	Binary	1	00000001
8'bx0l	8	Binary	-	xxxxxx0l
'bz	unsized	Binary	-	z ... z (32 bits)
8'HAD	8	Hex	173	10101101

- Left-most bit is extended as required to fill bits

Working with signed and unsiged reg and wire*

Verilog 2001 provides signed reg and wire vectors

Casting to and from signed may be
implicit
or may be
explicit by using

```
$unsigned()     reg_s=$unsigned(reg_u);
$signed()       reg_u=$unsigned(reg_s);
```

Implicit and Explicit Casting is always a **dumb conversion** (same as C), they never change bits, just the interpretation of the bits (e.g. -1 is not round to 0 upon conversion to unsigned, it is just reinterpreted as the largest unsigned value) for subsequent operations

Working with signed and unsiged reg and wire*

- If a mix of signed and unsigned operands are provided to an operator, both operands are first cast to be unsigned (like C)
- Signed/Unsigned Casting may be followed with length adjustment
  - assignment to a shorter type is always just bit truncation (no smart rounding such as unsigned(-1) → 0)
    - Error Checking:
    - For unsigned, truncation is no problem as long as all the truncated bits are 0 (ex: 0000101 (5) can truncate up to 3 bits)
    - For signed, truncation is no problem as long as all the bits truncated are the same AND they match the surviving msb (ex: 1111100 (-3) can truncate up to 4 bits)
- Assignment to a longer type is done with either zero or sign extension depending on the type:
  - Unsigned types use zero extension
  - Signed types use sign extension

Rules for expression bit lengths*

A self-determined expression is one in which the length of the result is determined by the length of the operands or in some cases the expression has a predetermined length for the result.

Example: as self-determined expression the result of addition has a length that is the maximum length of the two operands.

Example: a comparison is always a self-determined expression with a 1-bit result

However, addition and other operation expressions may act as a context-determined expression in which the bit length is determined by the context of the expression that contains it.

Addition and Overflow Detection*

- Two's compliment addition of two numbers where the longest is N-bit can require up to N+1 bits to store a full result
- Two's compliment addition can only overflow if the signs of the operands are the same
- Overflow check: input sign bits are same and do not match result sign bit
  - Examples No Overflow:

```
  10000000 (-128)          01000000   (64)
+ 01111111  (127)        + 00111111   (63)
= 11111111   (-1)        = 01111111  (127)
```

  - Ex Overflow:

```
  10000000 (-128)
+ 10000000 (-128)
= 00000000 (-128)
```

Subtraction and Overflow Detection*

- Two's compliment subtraction can only under/overflow if the signs of the operands are different, otherwise the magnitude of the result must be smaller than the maximum magnitude of the two operands.
- Overflow check: resulting sign bit does not match the first operand and the sign bits of the operands are different
  - i.e. sign bit of the second operand and the result are the same and not equal to that of the first

$$\begin{array}{rl} & 11111111 \\ - & 01111111 \\ = & \end{array} \longrightarrow \begin{array}{rl} & 11111111 \\ + & 10000001 \\ = & 00000000 \end{array} \qquad \text{ok}$$

$$\begin{array}{rl} & 01111111 \\ - & 11111111 \\ = & \end{array} \longrightarrow \begin{array}{rl} & 01111111 \\ + & 00000001 \\ = & 10000000 \end{array} \qquad \text{Not ok}$$

Addition*

In this example we see that addition obeys modular arithmetic with a result u8y=0

```
    ur8a = 128;
    ur8b = 128;
    ur8y= ur8a+ur8b;
```

In this example we see that the addition is an expression paired with an assignment, so the length of the assigned variable sets the context-determined expression operand length of the addition to take on the length of the largest operand, 9-bits. Using zero-extension in this case, the addition operands are each extended to 9-bits before addition.

The result is ur9y=256

```
 ur8a = 128;
 ur8b = 128;
 ur9y= ur8a+ur8b; //9-bit addition,
                  // arguments to addition are
                  // zero-extended to 9 bits
```

Self-Determined Expression and Self-Determined Operands*

- Some operators always represent a self-determined expression, they have a well-defined bit-length that is independent of the context in which they are used and must be derived directly from the input operand(s) (the result may still be extended or truncated as needed). These may also force the operands to obey their self-determined expression bit length.
- The concatenation operator is one such example of a self-determined expression with a bit-length that is well-defined as the sum of length of its operands, and in turn its operands are forced to use their self-determined expression length.
  Single-operand : e.g. **{a}** for which the result is the length of **a**
  Multiple operands: e.g. **{2'b00,b,a}** for which the result length is 2+length(**b**)+length(**a**)
- The use of a single operand {} can force a self-determination for expressions like addition:
  - In this next example, the self determined length of the addition is 8-bits which results in 0 for the summation. The 8-bit result from the concatenation operator is always unsigned and thus is zero extended.

```
ur8a = 128;
ur8b = 128;
ur16y= {ur8a+ur8b}; //8 bit addition
ur16z= ur8a+ur8b;   //16 bit addition
```

Rules for expression bit lengths from IEEE Standards Doc.

5.4.1 Rules for expression bit lengths
The rules governing the expression bit lengths have been formulated so that most practical situations have a natural solution.

The number of bits of an expression (known as the size of the expression) shall be determined by the operands involved in the expression and the context in which the expression is given.

A self-determined expression is one where the bit length of the expression is solely determined by the expression itself—for example, an expression representing a delay value.

A context-determined expression is one where the bit length of the expression is determined by the bit length of the expression and by the fact that it is part of another expression. For example, the bit size of the right-hand expression of an assignment depends on itself and the size of the left-hand side.

Table 5-22 shows how the form of an expression shall determine the bit lengths of the results of the expression. In Table 5-22, i, j, and k represent expressions of an operand, and L(i) represents the bit length of the operand represented by i.

Multiplication may be performed without losing any overflow bits by assigning the result to something wide enough to hold it.

Reference from IEEE Standards Doc.*

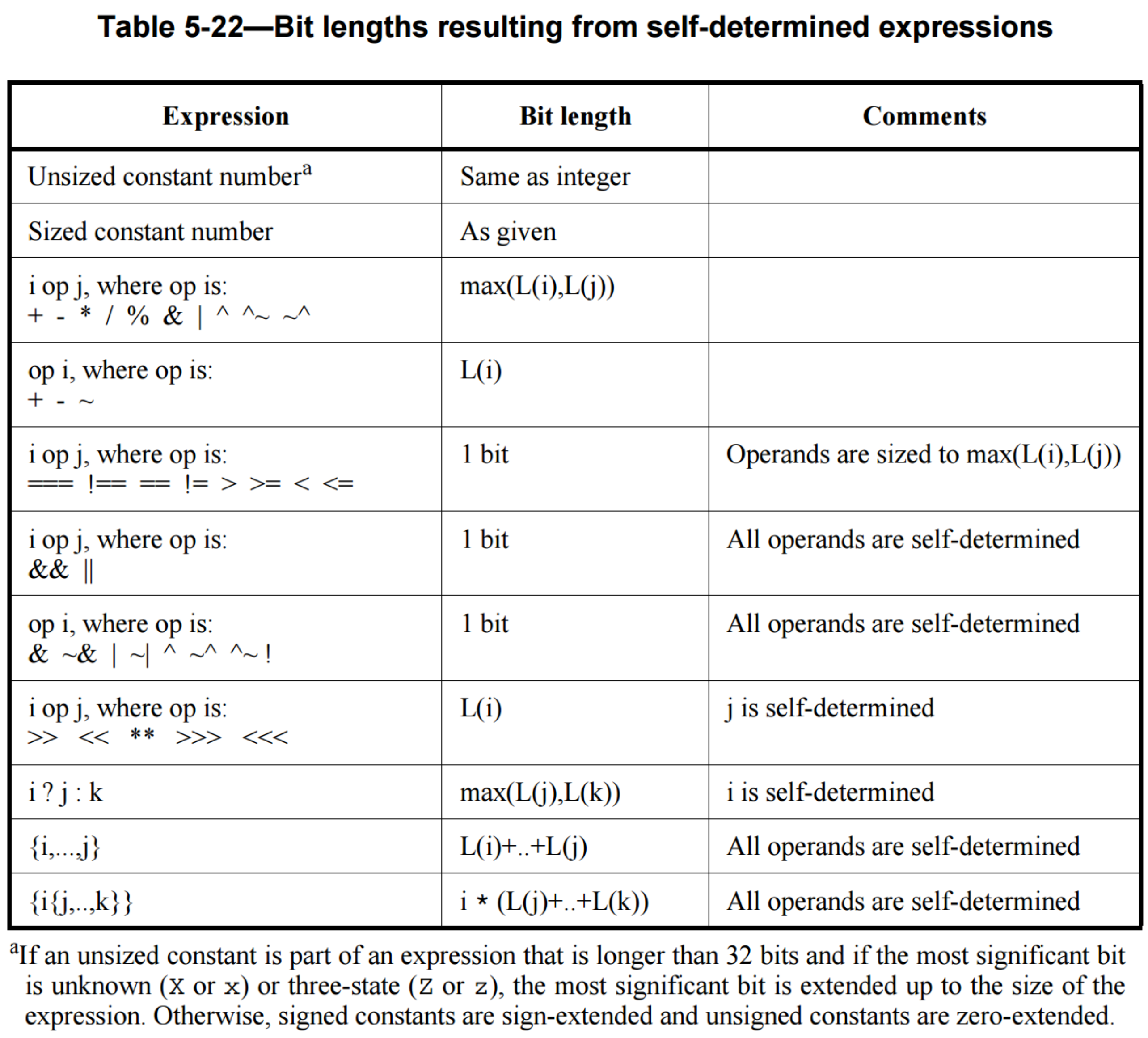


Table 5-22—Bit lengths resulting from self-determined expressions

Expression	Bit length	Comments
Unsized constant number[a]	Same as integer	
Sized constant number	As given	
i op j, where op is: + - * / % & | ^ ^~ ~^	max(L(i),L(j))	
op i, where op is: + - ~	L(i)	
i op j, where op is: === !== == != > >= < <=	1 bit	Operands are sized to max(L(i),L(j))
i op j, where op is: && ||	1 bit	All operands are self-determined
op i, where op is: & ~& | ~| ^ ~^ ^~ !	1 bit	All operands are self-determined
i op j, where op is: >> << ** >>> <<<	L(i)	j is self-determined
i ? j : k	max(L(j),L(k))	i is self-determined
{i,...,j}	L(i)+..+L(j)	All operands are self-determined
{i{j,..,k}}	i * (L(j)+..+L(k))	All operands are self-determined

[a]If an unsized constant is part of an expression that is longer than 32 bits and if the most significant bit is unknown (X or x) or three-state (Z or z), the most significant bit is extended up to the size of the expression. Otherwise, signed constants are sign-extended and unsigned constants are zero-extended.

Getting the Spec

At UMBC, goto http://ieeexplore.ieee.org/

Search for
    IEEE Std 1364-2005

You'll find

IEEE Standard for Verilog Hardware Description Language IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)

Along with the older standard 1995 standard and the SystemVerilog standard

Addition and Mixed Sign

Addition in the context with of assignment will cause extension of operands to the size of the result (the synthesizer may later remove useless hardware). However, the extension is performed according to the type of addition, which is determined by the operands. Therefore, signed extension is only performed if BOTH operands are signed.

```
module test();
   reg signed [7:0]   s;
   reg        [7:0]   u;
   reg signed [7:0]   neg_two;
   reg        [15:0] x1,x2;
   reg signed [15:0] y1,y2;
   initial begin
     neg_two = -2;
     s =  1; u =  1;
     x1 = u + neg_two; x2 = s + neg_two;
     y1 = u + neg_two; y2 = s + neg_two;
     $display("%b",x1); $display("%b",x2);
     $display("%b",y1); $display("%b",y2);
   end
endmodule
```

```
0000000011111111
1111111111111111
0000000011111111
1111111111111111
```

Gotchas*

```
. . .
reg [3:0] bottleStock = 10; //unsigned

always @ (posedige clk, negedge rst_)
   if (rst_==0)
     bottleStock<=10;
   else if (bottleStock >= 0) //always TRUE!!!
      bottleStock <= bottleStock-1;
```

Gotchas*

```
. . .
input wire [2:0] removeReq;
signed reg [3:0] remainingStock = 10; //signed

always @ (posedige clk, negedge rst_)
 if (rst_==0)
   remainingStock<=10;
 else if ((remainingStock-removeReq) >= 0)//always TRUE!!!
      remainingStock <= remainingStock-removeReq;
```

Scaling by Powers of Two

- Multiplying by a positive integer power of two may be performed with the logical left-shift shift operator -- Multiplying by 2^k is left shift by k:

`a<<k` Typically synthesizers require k to be a constant;

- k bits on left are discarded
- Detecting Overflow: no overflow if all discarded bits are the same and the new msb matches the truncated bits
- Logical Shifting is usually inexpensive – just “rewiring”.

Multiplication

In general to hold the result you need M+N bits where M and N are the length of the operands

```
wire [N-1:0] a;
wire [M-1:0] b;
wire [M+N-1:0] y;
```

```
y=a*b;
```

The multiplication in the context with the assignment will cause extension of operands to the size of the result (the synthesizer may later remove useless hardware). However, the extension is performed according to the type of operands (signed extension is only performed if BOTH operands are signed)

Multiplication

- Multiplication by two variables can be expensive, with a size on the order of MxN (full 1-bit adders and AND gates as 1-bit mult)

```
1011  x  10010  =
M bits   N bits
```

```
                      M bits
         (      1011  x  0  )
         (     10110  x  1  )
 N bits  (    101100  x  0  )
         (   1011000  x  0  )
       + (  10110000  x  1  )
         ---------------------
            11110110
```

- FPGA may have banks of “hard” mutipliers (e.g. 8x8 multipliers, sometime in what is called DSP slices) so as to avoid using large portions of the programmable fabric.
  Ex. 8-bit multipliers can be used for 16-bit mult, mathematically shown: **Y = {AH,AL} x {BH,BL} = AH*BH<<16+AH*BL<<8+AL*BH<<8+AL*BL**
  The synthesizer will recognize the size of the multiplication block construct the mapping to available multipliers for you.

Multiplication by Constants

- Multiplication by constants with only few non-zero bits can be inexpensive:

$$u*24 = u * (16+8) = u<<4 + u<<3$$

This concept is important for computer engineer to have in their tool belt.
Using example from previous slide: if the second operand is a constant, the synthesizer reduces the multiplication to shift and one adder:

$$\begin{array}{cccccr} 1011 & \times & \underline{10010} & = & & (\quad\ \mathbf{10110}) \\ \text{Variable} & & \text{Const} & & + & \underline{(\mathbf{10110000})} \\ \text{M bits} & & \text{N bits} & & & 11110110 \end{array}$$

Rounded Integer Division

- Rounded-result division of integers A,B may be accomplished by adding an offset(bias) to A that is half the magnitude of the divisor B (truncated to an integer) and matches the sign of the result

```
|round(float(A)/float(B))| = (|A|+|B/2|)/|B|
```

- Why: Because integer division is round towards 0, but if A%B is at least half of B then we need to round away from 0, which can accomplished by adding effectively adding 0.5 to the magnitude of the result before rounding

```
A+(B*sign(A))/2  where sign(A) is 1 or -1
                 according to the sign of A
```

- Try to make code to divide a integer (signed), **S**, by 256 with a rounded result

```
result = (S>=0) ? ((S+128)/256) : ((S-128)/256);
```

- Try to make code to divide a integer (signed), **S**, by 5 with a rounded result

```
result = (S>=0) ? ((S+2)/5) : ((S-1)/5);
```

- A synthesizer may only support division by powers of two and possibly only constants

Divide by (positive integer) power of two

- Divide x by 2^k is almost the same as an <u>arithmetic</u> right shift
- discard k bits on the right and replicate sign bit k times on the left. Must use the arithmetic shift to perform sign extension;

  `x>>>k;` same as `{k{x[msbindex]},x[msbindex:k]}`

- However, "integer division" is defined by truncation of the fractional bits of the result, also known as "round towards zero" To mimic this behavior more is needed:
  This is
  For a positive v corresponds to floor(x/n) ex: 5/2= 2
  For a negative v it corresponds to ciel(x/n) ex: -5/2= -2
- If x is <u>positive</u> you can just use logical shifting: `x>>k;`
- If x is <u>negative</u>, we want ciel(x/n) which may be computed by applying an **bias that is half the divisor** :

  $$\mathrm{floor}((x+2^{k-1})/(2^k))$$

  In verilog:

```
(x +(1<<(k-1)) >>> k
```