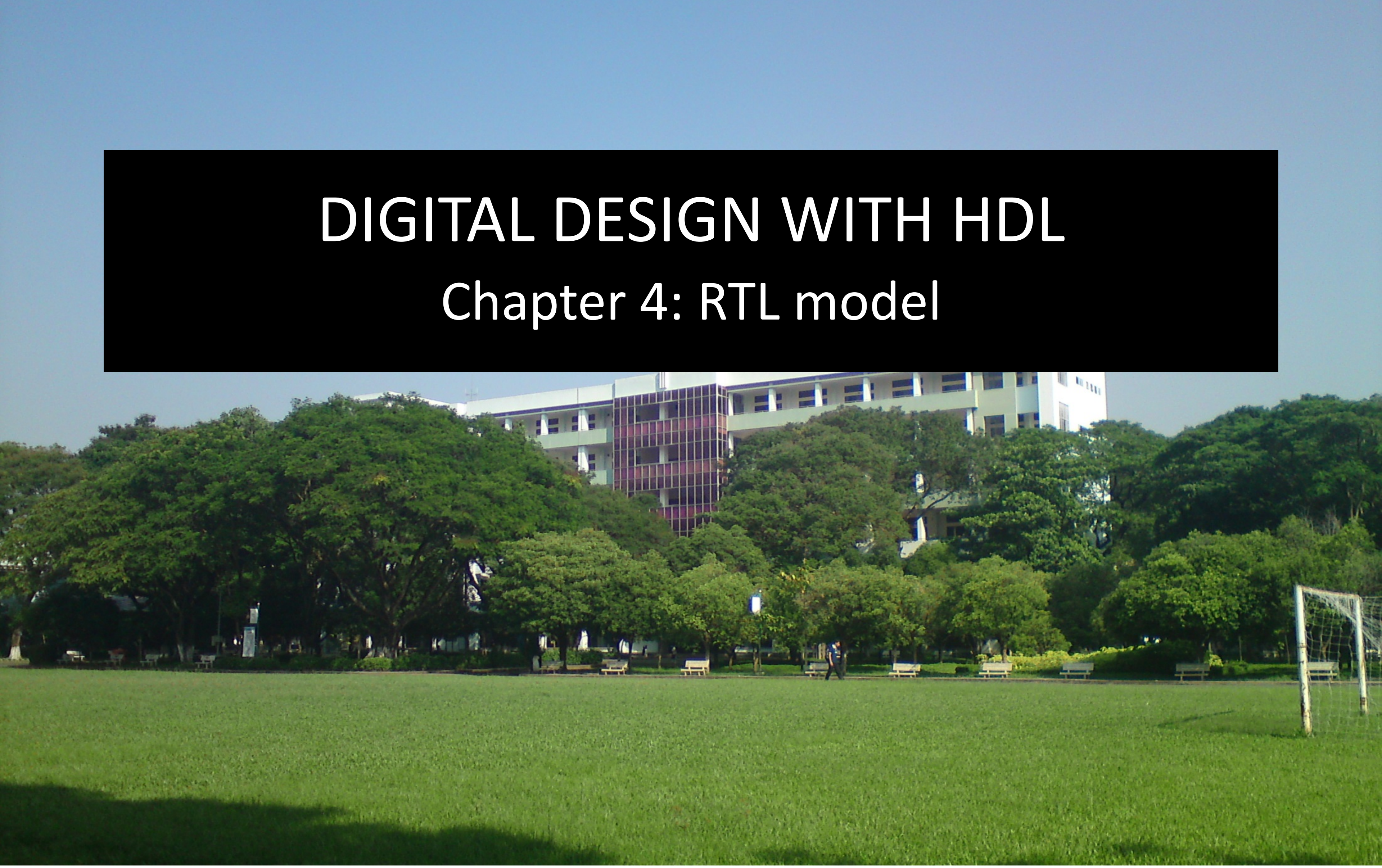


DIGITAL DESIGN WITH HDL

Chapter 4: RTL model

RTL Verilog

- Higher-level of description than the structural model
  - Don’t always need to specify each individual gate
  - Can take advantage of **operators**
  - **Used to design combinational logic only**
- More hardware-explicit than the behavioral model
  - Doesn’t look as much like software
  - Frequently easier to understand what’s happening
- Very **easy to synthesize**
  - Supported by even primitive synthesizers
  - **Note**: not all operators can be synthesized

Continuous assignments

- Syntax:

```
assign [#delay] lhs_net = rhs_expression;
```

  - Occur outside procedural blocks (initial or always)
- Operation:
  - Whenever the value of a variable in rhs changes, rhs is evaluated and assigned to lhs after the delay interval
- Example:

```
wire out, a, b;
assign out = a & b;
```

- Continuous: rhs is constantly operating

The lhs_net

- The left-hand-side can be:
  - Scalar nets
  - Vector nets
  - Single bits of vector nets
  - Part-selects of vector nets
  - Concatenation of any of the above
- Examples:

```
wire [7:0] out, val;
wire [7:0] a, b;
assign out[7:4] = a[3:0] | b[7:4]; //vector nets
assign val[3] = c & d; //single bits of vector nets
assign {a, b} = stimulus[15:0]; //concatenation of scalar nets
```

- **Cannot be used to assign a value to a register**

The rhs_expression

- Variables/signals in the right-hand-side can be
  - Registers: scalars and vectors
  - Nets: scalars and vectors
  - Function calls
- Example

```
wire x1, x2, x3, x;
assign x = (x1&x2)|x3;
```

```
reg [3:0] a, b
wire [3:0] sum;
assign sum = a + b;
```

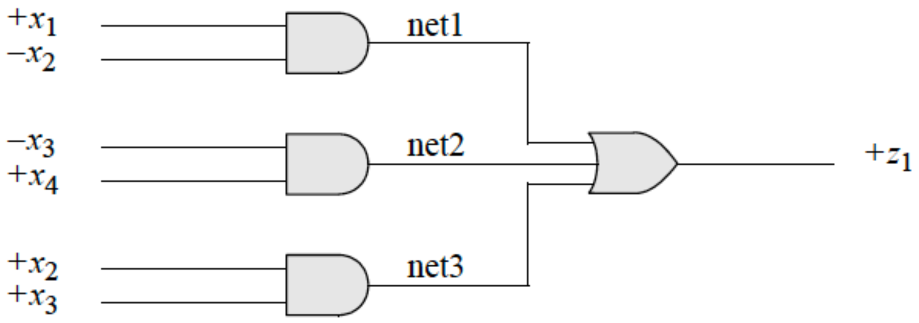
- Can also be a pass-through/alias

```
assign a = stimulus[16:9];
```

  - **note**: "aliasing" is only one direction

Example

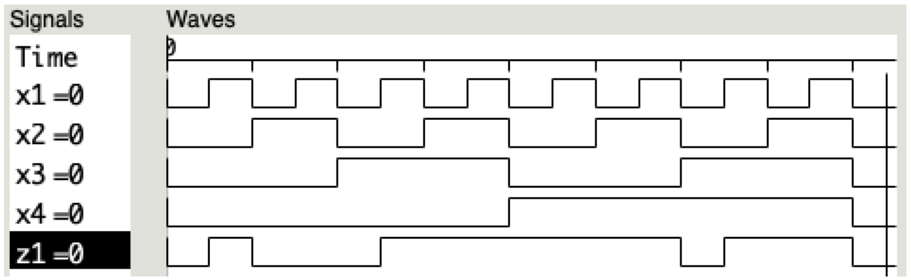
- Design and test a module of the following circuit



```
module log_eqn_sop (x1, x2, x3, x4, z1);
  input x1, x2, x3, x4;
  output z1;

  assign z1 = (x1&~x2) | (~x3&x4) | (x2&x3);
endmodule
```

```
module log_eqn_sop_tb;
  reg [5:1] x; //combine stimulus
  wire z1;
  initial for (x = 0; x < 16; x = x + 1) #5;
  log_eqn_sop inst1(.x1(x[1]),.x2(x[2]),
                    .x3(x[3]),.x4(x[4]),.z1(z1));
endmodule
```



Expression in Verilog

- Expressions consist of **operands** and **operators**
  - The value of an expression is determined from the combined operations on the operands
  - The value of an expression can be assigned to a left-hand-side net or register variable
    - Continuous assignments: net
      - *assign*
    - Procedural assignments: register
      - In *initial* or *always* blocks

Operands

Operands	Comments	Example
Constant	Signed or unsigned	4'b1010; -10
Parameter	Similar to a constant	parameter a = 4'b1010
Net	Scalar or vector	wire a; wire [3:0] b;
Register	Scalar or vector	reg a; reg [3:0] b;
Bit-select	One bit from a vector	a[0]; a[1]
Part-select	Contiguous bits of a vector	a[3:1]; a[2:0]
Memory element	One word of a memory	mem[0];
Function call	System or user function	func(a,b);

Constant

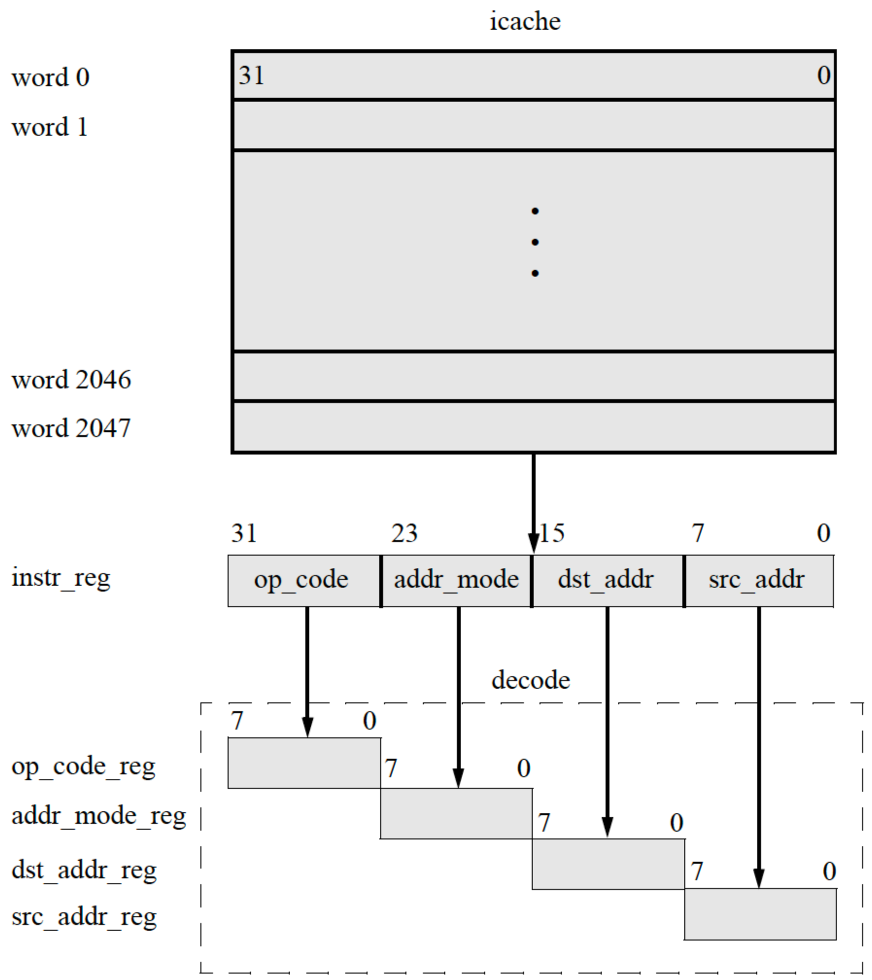
- Constant can be signed or unsigned
  - A **decimal integer**: a signed number
  - An integer **specified by a base**: an unsigned number

Constant	Comments
127	Signed decimal: Value = 8-bit binary vector: 8’b0111_1111
-1	Signed decimal: Value = 8-bit binary vector: 8’b1111_1111
-128	Signed decimal: Value = 8-bit binary vector: 8’b1000_0000
4’b1110	Binary base: Value = unsigned decimal 14
8’b0011_1010	Binary base: Value = unsigned decimal 58
16'h1A3C	Hexadecimal base: Value = unsigned decimal 6716
16'hBCDE	One word of a memory
9'O536	Octal base: Value = unsigned decimal 350
-22	Signed decimal: Value = 8-bit binary vector: 8’b1110_1010
- 9'o352	Octal base: Value = 8-bit binary vector: 1110_1010 = unsigned decimal 234

Memory element

- A memory element refers to a word in memory
  - One or more bits
  - **Referenced by words only** - no bit-/part-select

```
reg [31:0] icache [0:2047];//memory
reg [31:0] instr_reg;
instr_reg = icache [127];
op_code_reg = instr_reg [31:24];
```



Operators

Type	Symbol	Operation	# operands	Example
Arithmetic	+	Add	2 or 1	a + b; +5
	-	Subtract	2 or 1	a - b; -10
	*	Multiply	2	a * b;
	/	Divide	2	a / b;
	%	Modulus	2	a % b;
Logical	&&	Logical and	2	a && b;
	||	Logical or	2	a || b;
	!	Logical negation	1	!a;
Relational	>	Greater than	2	a > b;
	<	Less than	2	a < b;
	>=	Greater than or equal	2	a >= b;
	<=	Less than or equal	2	a <= b;

Operators (cont.)

Type	Symbol	Operation	# operands	Example
Equality	==	Logical equality	2	a == b;
	!=	Logical inequality	2	a != b;
	===	Case equality	2	a === b;
	!==	Case inequality	2	a !== b;
Bitwise	&	Bitwise and	2	a & b;
	|	Bitwise or	2	a | b;
	~	Bitwise negation	1	~a
	^	Bitwise Exclusive-or	2	a ^ b;
	~^ or ~^	Bitwise Exclusive-nor	2	a ~^ b; a ~^ b;
Reduction	&	AND	1	&a;
	^&	NAND	1	^&a;
	|	OR	1	|a;
	^|	NOR	1	^|a;
	^	Exclusive-OR	1	^a
	^~ or ~^	Exclusive-NOR	1	~^a; ^~a;

Operators (cont.)

Type	Symbol	Operation	# operands	Example
Shift	<<	Shift left	2	a << 5;
	>>	Shift right	2	a >> 4;
Conditional	?:	Conditional	2	a ? b + c : d + e;
Concatenation	{}	Concatenation	2 or more	{a,b,c}
Replication	{{}}	Replication	2 or more	{3{a}}

Arithmetic operators

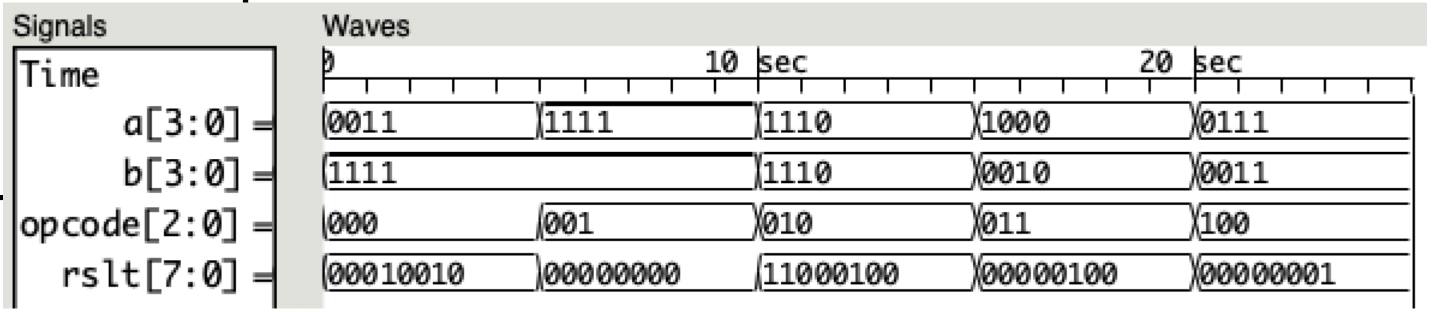
- Perform on one (unary) or two (binary) operands
  - Radices: binary, octal, decimal, or hexadecimal
  - Integer or real operands: signed (2'complement)
    - Integer register: signed
  - Register and net operands: unsigned
- Size of results
  - Multiply operator: sizeof(op1) + sizeof(op2)
  - Others: sizeof(the largest operand)

	Addition		Subtraction		Multiplication		Division
	Augend		Minuend		Multiplicand		Dividend
+)	Addend	–)	Subtrahend	×)	Multiplier	÷)	Divisor
	Sum		Difference		Product		Quotient, Remainder

Example

```
module arith(a, b, op, result);
  input [3:0] a, b;
  input [2:0] op;
  output reg [7:0] result;
  always @ (a or b or op) begin
    case (op)
      3'b000: result = a + b;
      3'b001: result = a - b;
      3'b010: result = a * b;
      3'b011: result = a / b;
      3'b100: result = a % b;
    endcase
  end
endmodule
```

```
a = 0011, b = 1111, opcode = 000, rslt = 00010010
a = 1111, b = 1111, opcode = 001, rslt = 00000000
a = 1110, b = 1110, opcode = 010, rslt = 11000100
a = 1000, b = 0010, opcode = 011, rslt = 00000100
a = 0111, b = 0011, opcode = 100, rslt = 00000001
```



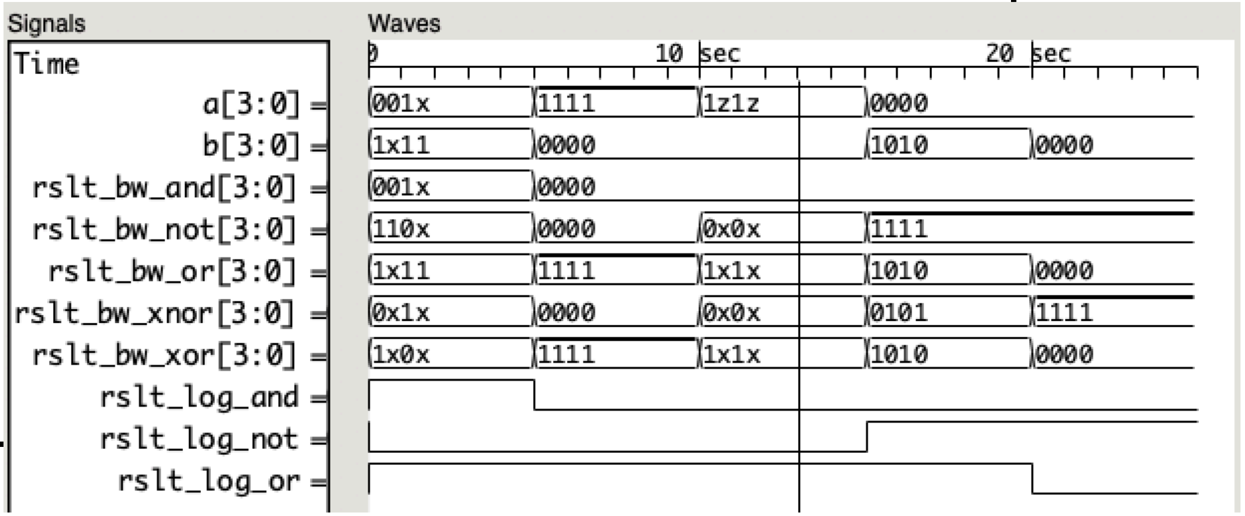
Logical vs. Bitwise

- Consider operands as logical values (true/false)
  - Nonzero operands: **true** (without any x or z bit)
  - Zero operands or ambiguous bit (x/z): **false**
- Results are always 1 bit
  - True: 1
  - False: 0

- Consider operands as vector/scalar values
  - Apply operators bit by bit
- Size of results = size of the largest operand
  - Insert 0 bits to smaller operands
- Results store: 0, 1, or x

Example

```
module bitwise;
  reg [3:0] a, b;
  wire [3:0] rslt_bw_and, rslt_bw_or, rslt_bw_not, rslt_bw_xor, rslt_bw_xnor;
  wire rslt_log_and, rslt_log_or, rslt_log_not;
  assign rslt_bw_and = a&b;
  assign rslt_bw_or = a|b;
  assign rslt_bw_not = ~a;
  assign rslt_bw_xor = a^b;
  assign rslt_bw_xnor = a~^b;
  assign rslt_log_and = a&&b;
  assign rslt_log_or = a||b;
  assign rslt_log_not = !a;
endmodule
```



Relational

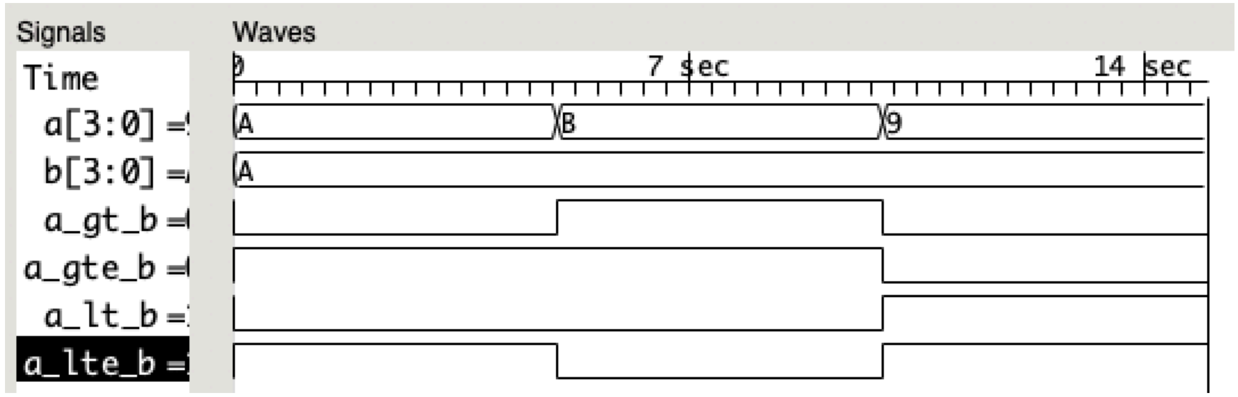
- Compare and return a logical value:
  - 1: true
  - 0: false
- Rules of operands
  - Net or register operands are unsigned
  - Real or integer operands are signed values
  - Any x or z bit in an operand return a result of x
  - Zero-extended for the smaller operand

Example

```
module comparator_assign(a, b, a_lt_b, a_lte_b, a_gt_b, a_gte_b);
  input [3:0] a, b;
  output      a_lt_b, a_gt_b, a_lte_b, a_gte_b;

  assign a_lt_b = a < b;
  assign a_lte_b = a <= b;
  assign a_gt_b = a > b;
  assign a_gte_b = a >= b;

endmodule
```



Equality

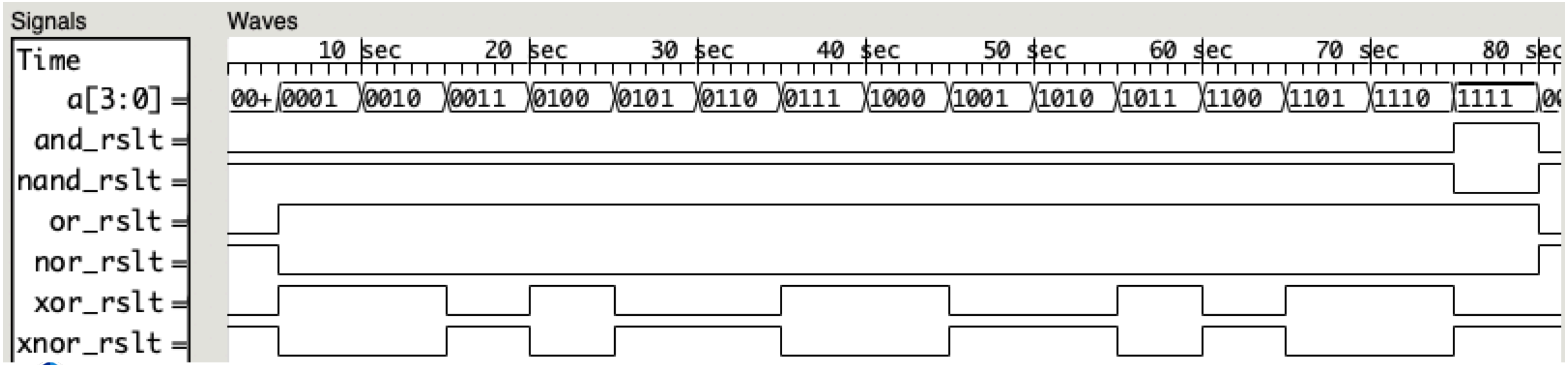
- Logical equality (== and !=)
  - Used in expressions to compare two values
  - Return 0 or 1 when there is no any x or z bit in the operands
    - Return a value of x when an x or z in either operand
  - Net or register operands are treated as unsigned values
    - Integer operands are signed values
- Case equality (=== and !==) — **Not synthesizable, simulation only**
  - Compare both operands bit-by-bit, including x and z
    - x === x and z === z
  - Return 1 if all bits are identical; return 0 if there exists an difference

Reduction

- Unary operators
  - Operating on a single vector and producing a single-bit result
  - If any bit in the operand is x or z, the result is x
- Reduction AND (&)
  - If any bit in the operand is 0, the result is 0; otherwise 1
    - e.g., &4'b1010 => 0; &4'b1111 => 1
- Reduction NAND (~&)
  - Invert of reduction AND
- Reduction OR (|)
  - If any bit in the operand is 1, the result is 1; otherwise 0
    - e.g., |4'b1010 => 1; |4'b0000 => 0
- Reduction NOR (~|)
  - Invert of reduction NOR
- Reduction XOR (^)
  - If there is even number of 1 bits in the operand, the result is 0; otherwise 1
    - e.g., ^4'b1010 => 0; ^4'b1110 => 1
- Reduction XNOR (~^ or ^~)
  - Invert of reduction XOR

Example

```
module reduction (a, and_rslt, nand_rslt, or_rslt, nor_rslt, xor_rslt, xnor_rslt);
  input [3:0] a;
  output     and_rslt, nand_rslt, or_rslt, nor_rslt, xor_rslt, xnor_rslt;
  assign and_rslt = &a; //reduction AND
  assign nand_rslt = ~&a; //reduction NAND
  assign or_rslt = |a; //reduction OR
  assign nor_rslt = ~|a; //reduction NOR
  assign xor_rslt = ^a; //reduction exclusive-OR
  assign xnor_rslt = ^~a; //reduction exclusive-NOR
endmodule
```



Shift

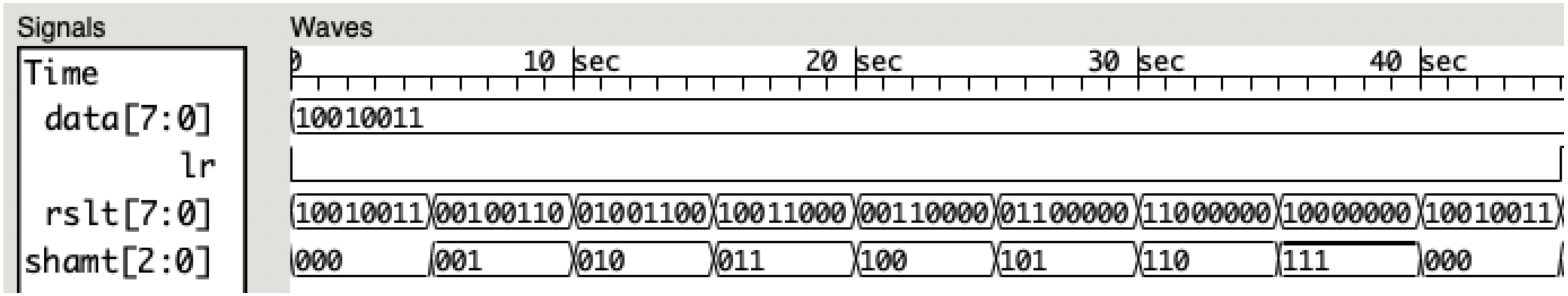
<Left operand> << <shift amount>

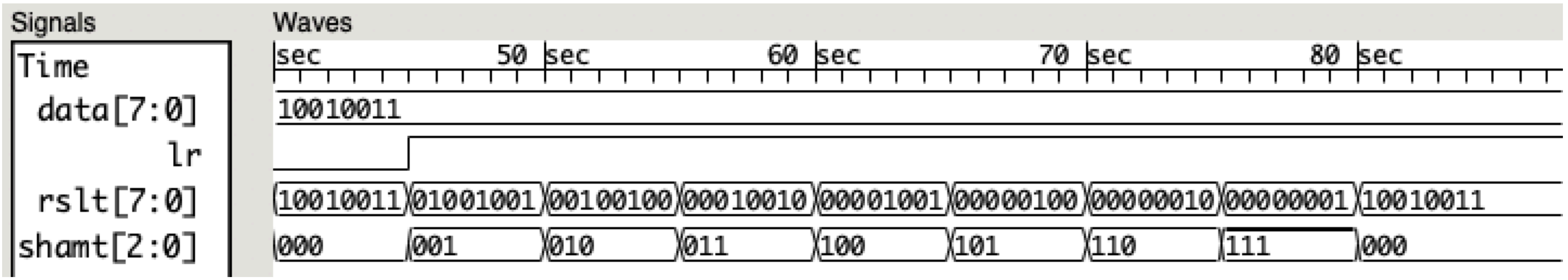
<Left operand> >> <shift amount>

- Logical shift operators
  - Bits in the <left operand> are shifted to left or right with 0s fillings
  - High-significant or low-significant bits are lost
- <Shift amount>: the number of positions shifted
  - Can be an expression
  - Always treated as unsigned numbers
  - If there exists any x or z bit, the result will be unknown

Example

```
module shift(data, shamt, rslt, lr);
   input [7:0] data;//left operand
   input [2:0] shamt;//shift amount
   output [7:0] rslt;
   input       lr; //0: shift left; 1: shift right
   assign rslt = (lr == 1'b0)? (data << shamt):(data >> shamt);
endmodule
```





Conditional

- Syntax:

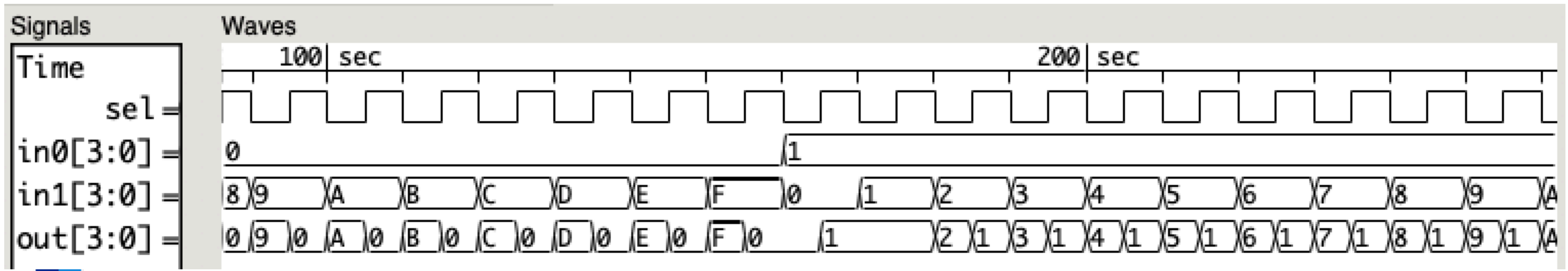
  <conditional expression>?<true expression>:<false expression>

- Conditional operators can be nested
- Operation:
  - <conditional expression> is evaluated to choose the result of <true expression> or of <false expression>
  - If <conditional expression> is evaluated to x, the results of both <true expression> and <false expression> will be applied to a special bitwise operator
    - $1 \diamond 1 = 1$
    - $0 \diamond 0 = 0$
    - Otherwise $\Rightarrow$ x

Example

```
module mux4bit(in0, in1, sel, out);
   input [3:0] in0, in1;
   input       sel;
   output [3:0] out;
   assign out = sel?in0:in1;
endmodule
```

```
   mux4bit M(.in0(in0[3:0]), .in1(in1[3:0]),
                        .sel(sel[0]), .out(out));
   initial begin
      for (in0 = 0; in0 < 16; in0 = in0 + 1)
            for (in1 = 0; in1 < 16; in1 = in1 + 1)
                  for (sel = 0; sel < 2; sel = sel + 1)
                        #5;
      #5 $finish;
   end
```



Signals
Waves
100 sec
200 sec
Time
sel
in0[3:0]
in1[3:0]
out[3:0]
0 1
8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A
0 9 0 A 0 B 0 C 0 D 0 E 0 F 0 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 A

Concatenation

- Form a result from multiple operands
- Syntax:

{operand 1, operand 2, ….}

- The sizes of operands must be known before making concatenation
  - E.g., {3'b000, 4'hA} ⇒ 7'b0001010
  - E.g., {3'b000, 10} ⇒ error
- The size of the result is sum of all sizes of the operands
- Operands are appended from left to right

Replication

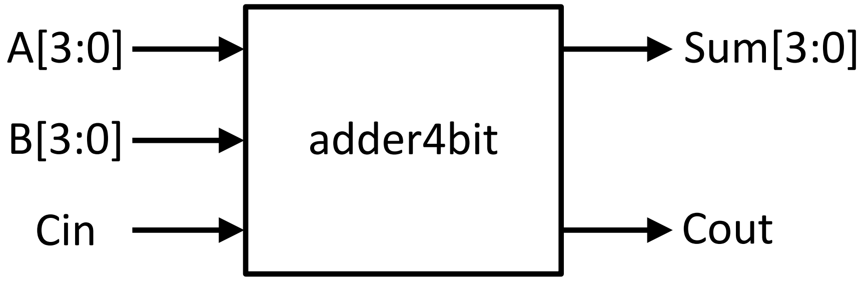
- Replicate the pattern a number of times
- Syntax:

{rep_number{expr1, expr2,… , exprN}}

- Similar to the concatenation operator with all operands are the same
  - The sizes of expressions must be known
- Ex:
  - {3{2'b01,2'b00}} ⇒ 12'b0100_0100_0100
  - {4{1}} ⇒ error

Design examples

- **Example 1**: design and implement an 4-bit full adder
- **Answer**:



```
module  adder4b (Sum, Cout, A, B, Cin);
  input    [3:0] A, B;
  input    Cin;
  output  [3:0] Sum;
  output  Cout;
  assign {Cout, Sum} = A + B + Cin;
endmodule
```

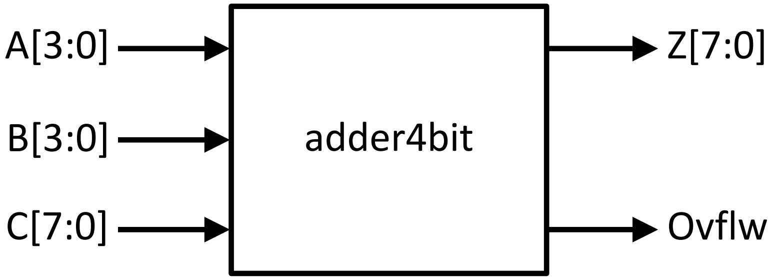
Design examples

- **Example 2**: Design a multiply-accumulate (MAC) unit that computes

$$Z[7:0] = A[3:0] \times B[3:0] + C[7:0]$$

It sets overflow to one, if the result cannot be represented using 8 bits.

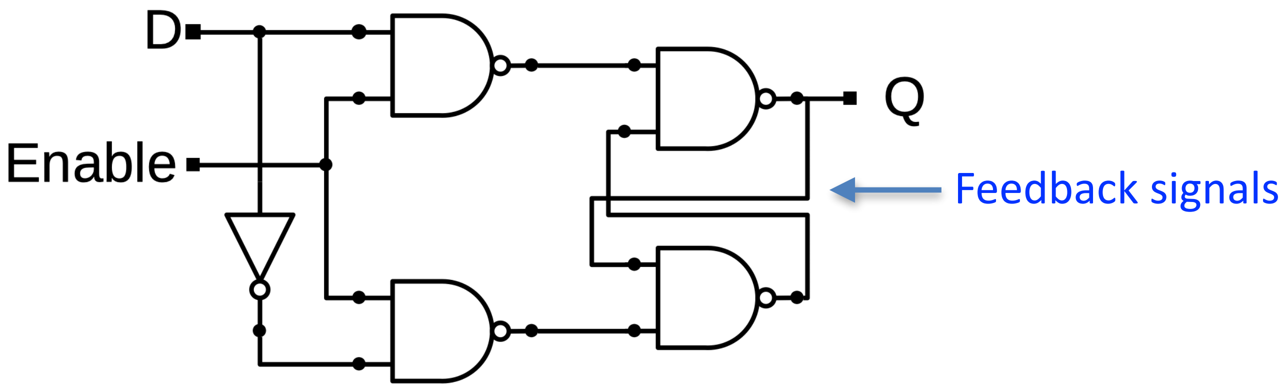
- **Answer**:



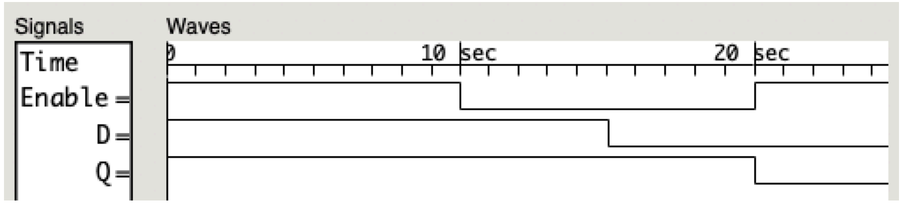
```
module mac(output [7:0] Z, output Ovflw,
                    input [3:0] A, B, input [7:0] C);
    wire [8:0] P;
    assign P = A*B + C;
    assign Z = P[7:0];
    assign Ovflw = P[8];
endmodule
```

Latch

- Output Q is keep unchanged when Enable is not active; otherwise input D is transfer to Q



```
module latch(output Q, input D, Enable);
   assign Q = Enable ? D : Q;
 endmodule
```



Exercise: Rock - Paper - Scissors

- In Vietnamese: oẳn - tù - tì

```
module rps(win, player, p0guess, p1guess);
```

- Assumptions:
  - Input: p0guess, p1guess = {0 for rock, 1 for paper, 2 for scissors}
  - Output: player is 0 if p0 wins, 1 if p1 wins, and don’t care if there is a tie
  - Output: win is 0 if there is a tie and 1 if a player wins
- Reminders
  - Paper beats rock, scissors beats paper, rock beats scissors
  - Same values tie

Summary

- RTL model
  - Use continuous assignment: assign
  - Combinational circuits only
  - Operators
  - Operands

The end