Today’s VLSI: IMPOSSIBLE to Manage by Hand

1979 (3μm) 29K Trans.  →  2011 (32nm) 2.3B Trans.

8086 16 mm²  →  Sandy Bridge 435 mm²
Computer-Aided Design Concepts

- **Be creative** — More time on **chip features**, less time on building block realization (e.g. adders, multipliers, etc.)

- **Be prophetic** — Fast system simulation to **debug & design budget at early stages** before detailed implementation.

- **Be efficient** — Fast area/power/delay estimate to select suitable architectures based on design constraints.

**Need from CAD tools & high-level modeling**

---

Computer-Aided Design Tools

- Calibre
- Design Compiler
- RC Compiler
- IC Compiler
- PrimeTime
- SoC Encounter
- Cadence Virtuoso
- HSpice
- Modelsim
- Power Compiler

**Mentor Graphics**
Computer-Aided Design Methodology

1. Register-transfer-level (RTL) Coding

\[
\text{module ADD (O, A, B);}
\text{output [15:0] O;}
\text{input [15:0] A, B;}
\text{assign O = A+B;}
\text{endmodule}
\]

\[
\text{module TestAdd;}
\text{...}
\text{ADD TEST(O,A,B);}
\text{...}
\text{endmodule}
\]

Main Design + Testbench = Functional Verification

2. Synthesis

Design Compiler

3. Place & Route

SoC Encounter

4. DRC & LVS

Calibre

Two Pieces of Verilog

1. Design
   (synthesis and PnR)

2. Testbench
   (simulation testvectors)
Example of a Design and its Testbench

```verilog
// Design
module EE216A_ADD (DO, DI_A, DI_B);

parameter WL = 4;
output [WL:0] DO;
input [WL-1:0] DI_A, DI_B;

assign DO = DI_A + DI_B;

endmodule

// Testbench
module Test_EE216A_ADD;

parameter WL_Test = 4;
wire [WL_Test:0] DO;
reg [WL_Test-1:0] DI_A, DI_B;

EE216A_ADD #(WL_Test) TEST(DO, DI_A, DI_B);

initial begin
$monitor($time, "Out=%d, IA=%d, IB=%d", DO, DI_A, DI_B);
$display("@ 4b Adder Simulation Starts \n");
DI_A = 4'd0; DI_B = 4'd0;
#10 DI_A = 4'd3; DI_B = 4'd1;
#27 DI_A = 4'd5; DI_B = 4'd7;
#15 DI_A = 4'd0; DI_B = 4'd2;
$finish;
end
endmodule
```

Four Levels of Abstraction

- **Behavioral**
  - case(), always@(), ...  

- **Data Flow**
  - A+B, A*B, ...  

- **Gate Level**
  - AND, OR, NAND ...  

- **Switch Level**
  - NMOS, PMOS

Not useful in today’s design flow
Verilog Modeling

- Modeling Concepts
- Data Flow Model
- Behavioral Model

Modeling Concepts
**Modules**

The basic building block in Verilog
(similar to the functions in C language)

```verilog
`timescale 1ns/10ps
module EE216A_MUXADD (DO, DI_A, DI_B, DI_C, SEL);

parameter WL = 4;
output [WL:0] DO;
input [WL-1:0] DI_A, DI_B, DI_C;
input SEL;

assign DO = DI_A + (SEL ? DI_B : DI_C);
endmodule
```

**Parameters can be passed into instances**

```verilog
module EE216A_MUXADD (DO, DI_A, DI_B, DI_C, SEL);

parameter WL = 10;
output [WL:0] DO_1, DO_2;
input [WL-1:0] DI_A, DI_B, DI_C;
input SEL;

EE216A_MUXADD #(WL) U00 (.DO(DO_1), .DI_A(DI_A), .DI_B(DI_B), .DI_C(DI_C), .SEL(SEL));
EE216A_MUXADD #(WL) U01 (.DO(DO_2), .DI_A(DI_A), .DI_B(DI_B));
endmodule
```

**Modules and Module Instance**

```verilog
`timescale 1ns/10ps
module EE216A_MUXADD (DO, DI_A, DI_B, DI_C, SEL);

parameter WL = 4;
output [WL:0] DO;
input [WL-1:0] DI_A, DI_B, DI_C;
input SEL;

assign DO = DI_A + (SEL ? DI_B : DI_C);
endmodule
```

*instatiate sub-module by stating its name*
## Net

**Connect physical hardware elements**

Keyword: **wire**  
(default: 1 bit, unless specified as a multi-bit net)

1-bit net

```plaintext
wire a;
```

3-bit vector net

```plaintext
wire [2:0] a;
```

(same as `a[2]`, `a[1]`, `a[0]`)

## Register

**Represent a data storage element**

Keyword: **reg**

**Note 1:** Unlike a net, a register does **NOT** need a driver to continuously assign it a value.

**Note 2:** Depends on the `always@` expression, a register can be an edge-triggered flip-flop, or a combinational logic (discussed later)

```plaintext
reg Q;
always@(D)
Q = D + 1;
```

```plaintext
wire D; reg Q;
always@(posedge CLK)
Q <= D + 1;
```
### Vector

**The way to manage multi-bit data**

Format: `[high:low]`

**Note 1:** both wire and reg can be defined as vector

**Note 2:** Use `[x:y]` to specify range of a vector

<table>
<thead>
<tr>
<th>8-bit vector net</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wire [7:0] a ;</code></td>
</tr>
<tr>
<td><code>a[7]: MSB of a</code></td>
</tr>
<tr>
<td><code>a[2:0]: Three LSBs of a</code></td>
</tr>
</tbody>
</table>

---

### Array

**The way to manage multiple homogeneous data**

(mostly when modeling a memory or a reg. file)

Format: `[idx_low:idx_high]`

**Note 1:** wire **CANNOT** be defined as array

**Note 2:** multidimensional array **NOT** supported

<table>
<thead>
<tr>
<th>8-word, 16-bit memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reg [15:0] a[0:7] ;</code></td>
</tr>
<tr>
<td><code>a[7]: 7^{th} element of a</code></td>
</tr>
<tr>
<td><code>(not the 7^{th} bit of a!!)</code></td>
</tr>
</tbody>
</table>
Ports (1) – General Knowledge

Interface for a module to communicate with its environment (other modules, testbench, etc...)

3 types: **input**, **output**, **inout**
(declared, by default, as **wire** in Verilog)

```verbatim
`timescale 1ns/10ps
module ModuleName (DO, DI_A, DI_B, DI_C);
    Specify port list right after the module name
    output [15:0] DO;
    input  [15:0] DI_A, DI_B, DI_C;
    ...... 
endmodule
```

Ports (2) – How Interface Works

Help to partition a design into multiple modules

```verbatim
module Module_A (AO, AI);
    output [1:0] AO; // 2-bit
    input   AI;
    ...... 
endmodule

module Module_B (BO, BI);
    output BO; input BI;
    ...... 
endmodule

module Top (TO, TI);
    output [1:0] TO;
    input   TI;
    Module_A U_00(.AO(TO), .AI(IntNode));
    Module_B U_01(.BO(IntNode), .BI(TI));
endmodule
```

Port Connection Rules (Next Page)
### Ports (3.1) – Connection Rules

Start point (**driver**) can be **wire** or **reg**.
End point (**load**) can ONLY be **wire**.

![Connection Diagram](image)

### Ports (3.2) – Connection Rules

**Two** methods to make connection b/w external signal and module port:

1. By same order defined in module

   ```
   ADD AdderInst(S_w, A_w, B_w, C_w) ;
   ```

2. By name (recommended, fewer mistakes)

   ```
   module ADD (SUM, A, B, CIN) ;
   .......
   endmodule
   ```

   ```
   ADD AdderInst(.SUM(S_w), .CIN(C_w),
   .A(A_w), .B(B_w)) ;
   ```
Lexical Conventions (1)

**Comments**
- Single-line – //
- Multiple-line – /* ...... */

**Operators**
- **Unary**: Stand before a single operand
- **Binary**: Stand between two operands
- **Ternary**: Two operators that separate three operands

**Unary Example**
- `b = ~a ;`
- `b = |a ;`

**Binary Example**
- `c = a + b ;`
- `c = a * b ;`

**Ternary Example**
- `d = c ? a : b ;`

---

Lexical Conventions (2)

**Numbers**

<size><base><value>

- `8'b01010011` 8-bit binary, value = 83
- `16'h00fb` 16-bit hexadecimal, value = 251
- `4'd10` 4-bit decimal, value = 10

**Tip**: put underscores for better readability

`(8'b0101_0011, 16'h00_fb ... etc)`

**Verilog also uses...**
- `x` to specify unknown value
- `z` to specify high-impedance value
Lexical Conventions (3)

**Key words:** reserved words for data types  
**Identifiers:** names given to objects

```plaintext
wire x1_w;
reg x1_r;
output x1_o;
input x1_i;
```

(Just like we write “int i;” to create an integer named i in C language)

---

Data Flow Model
Overview: Expression, Operand, Operator

**Operand**
Can be time, integer, net, register, memory, etc.

**Operator**
Act on operands to produce desired results

\[ c = a + b \]

**Expression**
Combine operand & operator to produce result

---

**assign Statement (1)**

Describe a circuit at higher levels of abstraction

Keyword: **assign**

- an AND gate
  ```
  assign c = a & b ;
  ```

- a 4-bit adder
  ```
  ```
(remember to declare a,b,c as 4-bit operands first)
**assign** Statement (2)

Four parallel XOR gates

```vhdl
```

(Eqvl. to implement four XORs separately, but this coding style is much simpler and more readable)

**Properties of **assign** statement:**

- Left-hand side (e.g. `c[3:0]`) must **ALWAYS** be a scalar or vector net. Can’t be a register.
- Right-hand side can be nets or registers.
- Assignments are always active: The output is evaluated whenever any one of inputs changes.

---

**One More Trick: Implicit Assignment**

Declare a net and place a continuous assignment **AT THE SAME TIME**

```vhdl
wire c ;
assign c = a & b ;
```

Regular Explicit Assign

```vhdl
wire c = a & b ;
```

Implicit Assign

---

6.27

6.28
Delay (1)

- A useful feature in Verilog is to model the gate delay at the very beginning design stages.
- Specify the value of delay after keyword, with format #DelayAmount (e.g. 10 time units).

```verilog
wire #10 c = a & b ;
```

(Implicit assignment delay)

```verilog
wire c ;
assign #10 c = a & b ;
```

(Regular assignment delay)

Delay (2)

- **Inertial delay**: Any input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

```verilog
wire #10 c = a & b ;
```
Important Concepts About Delay

- **Modeled delay is only for simulation purpose**
  - After synthesis the verilog into gate-level netlist, the realistic circuit delay will take place

- **Synthesis tool does NOT synthesize modeled delay. It just ignores it.**
  - Therefore, even if you put modeled delay in your verilog code, it won’t be any compilation error.

- **Recommended to put modeled delay in the testbench. Keep your synthesizable design as clean as possible.**

---

Arithmetic Operators

**Binary-type**
- Multiply – *
- Add – +
- Subtract – –
- Modulus – %
- Power – **
- Divide – /

+ and – can be binary or unary, but the unary “+” and “−” have higher precedence than the binary “+” and “−”

**Unary-type**
- Positive – +
- Negative – –

**Note:** If any operand bit has a value x, then the result of the entire expression is x
Logical and Rational Operators

Output a Boolean number (1 = true, 0 = false)

- Logical-and – &&
- Logical-or – ||
- Logical-not – ! (unary op.)
- Rational greater than – >
- Rational less than – <
- Rational greater or equal to – >=
- Rational less or equal to – <=

If A = 3, and B = 0, then
A && B evaluates to 0;
A || B = 1;
!A = 0;
!B = 1;
A > B = 1;
A < B = 0;
(A == 3) && (B == 0) = 1

**Note:** If any operand bit has a value x, then the result of the entire expression is x

---

Equality Operators (1)

Compare two operands **bit by bit**, with zero filling if the operands are of unequal length

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
<th>Possible Value</th>
<th>Synthesizable</th>
</tr>
</thead>
<tbody>
<tr>
<td>a == b</td>
<td>a equal to b, result unknown if x or z in a or b</td>
<td>0, 1, x</td>
<td>YES</td>
</tr>
<tr>
<td>a != b</td>
<td>a not equal to b, result unknown if x or z in a or b</td>
<td>0, 1, x</td>
<td>YES</td>
</tr>
<tr>
<td>a === b</td>
<td>a equal to b, <strong>INCLUDING</strong> x and z</td>
<td>0, 1</td>
<td>NO</td>
</tr>
<tr>
<td>a !== b</td>
<td>a not equal to b, <strong>INCLUDING</strong> x and z</td>
<td>0, 1</td>
<td>NO</td>
</tr>
</tbody>
</table>
Equality Operators (2)

Let $A = 4, B = 3, X = 4'b1010$ ; $Y = 4'b1101$ ; $Z = 4'b1xxz$, $M = 4'b1xxz, N = 4'b1xxx$

- $A == B$ 0
- $X != Y$ 1
- $X == Z$ $x$
- $Z === M$ 1
- $Z === N$ 0
- $M !== N$ 1

Bitwise Operator

Bitwisely compute logics in parallel. Multi-bit result. AND(\&), OR(\|), INV(\^), XOR(\^), XNOR(\^\^), ...

Let $X = 4'b1010$ ; $Y = 4'b1101$ ; $Z = 4'b10x1$

- $\sim X$ 4'b0101
- $X \& Y$ 4'b1000
- $X \| Y$ 4'b1111
- $X \^ Z$ 4'b00x0

Note: If one operand is shorter than the other, compiler automatically appends zeros
**Reduction Operator**

Bitwise op. on a single operand. One-bit result. 
AND (&), NAND (~&), OR (|), XOR (^), ...

Let \( X = 4'b1010 \); \( Y = 4'b1101 \); \( Z = 4'b10x1 \)

\[
\begin{align*}
& \& X & 1'b0 \\
| X & 1'b1 \\
^ Y & 1'b1 \\
^ Z & 1'bx
\end{align*}
\]

Note: If one operand is shorter than the other, compiler automatically appends zeros

---

**Shift Operator**

Shift to the right or left by a specified # of bits 
Right-shift (>>, Left-shift (<<)

Let \( X = 4'b1010 \)

\[
\begin{align*}
X >> 1 & 4'b0101 \\
X << 2 & 4'b1000 \\
X >> 3 & 4'b0001 \\
X << 4 & 4'b0000
\end{align*}
\]

Note: During shift operation, the vacant bit positions are filled with zeros, not wrap around.
Group multiple vector operands
Format: \{operand, ... , operand\} or \{(#)\{operand\}\}

Let A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110
{B,C}       4'b0010
{A,B,3'b001}  6'b100001
{A,B[0],C[1]}  3'b101
{(3){D[2]}}   3'b111

Note: Operands must be sized. Can be scalar or part-select vector wire/reg, or sized constants.

Important: How to Realize Sign Extension

• Verilog treats numbers as unsigned
• Sign-extension: very common operation in 2’s complement arithmetic

Let X[4:0] = 5'b11000 = −8 (signed number) and we want to divide X by 4 = 2^2:

wire [4:0] Y = X >> 2 ;
(2-bit right-shift is wrong. Y = 5'b00110 = +6)

wire [4:0] Y = {{{2}{X[4]}},X} ;
(Concatenate MSB is correct. Y = 5'b11110 = −2)
Conditional Operator (1)

Equivalent to “if-else” statement

<condition_expr> ? <true_expr> : <false_expr>

`condition_expr` evaluated first. If the result is

True : evaluate `true_expr`
False : evaluate `false_expr`

```
assign OUT = MODE ? (A+B) : (A-B) ;
(MODE=1, do addition; MODE=0, do subtraction)
```

Conditional Operator (2)

Nested conditional operations supported

```
assign OUT =
MODE[1] ? (MODE[0] ? (A+B) : (A–B)) :
(MODE[0] ? (A&B): (A|B)) ;
```

MODE = 2'b00: Bitwise OR
MODE = 2'b01: Bitwise AND
MODE = 2'b10: Subtraction
MODE = 2'b11: Addition
Operators: Precedence

<table>
<thead>
<tr>
<th>Operators</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>+ - ! ~</td>
</tr>
<tr>
<td>Multiply, Divide, Modulus</td>
<td>* / %</td>
</tr>
<tr>
<td>Add, Subtract</td>
<td>+ -</td>
</tr>
<tr>
<td>Shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>Relational</td>
<td>&lt; &lt;= &gt; &gt;=</td>
</tr>
<tr>
<td>Equality</td>
<td>== != === !==</td>
</tr>
<tr>
<td>Reduction</td>
<td>&amp;, <del>&amp;, ^, ^</del>,</td>
</tr>
<tr>
<td>Logical</td>
<td>&amp;&amp;,</td>
</tr>
<tr>
<td>Conditional</td>
<td>?:</td>
</tr>
</tbody>
</table>

Designer’s Experience: 16b Adder/Subtractor

Adder when MODE=0; Subtractor when MODE=1

Wrong

\[ \text{OUT} = A + \{(16)\{\text{MODE}\}\}^{\text{B}} + \text{MODE} \]

Correct

\[ \text{OUT} = A + \left(\{(16)\{\text{MODE}\}\}^{\text{B}}\right) + \text{MODE} \]
## Operators: Summary

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>*</td>
<td>multiply</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>divide</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>add</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>subtract</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>modulus</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>unary plus</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>unary minus</td>
</tr>
<tr>
<td>Logical</td>
<td>!</td>
<td>logical negation</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td>logical and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Operators: Summary

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit-wise</td>
<td>~</td>
<td>bitwise negation</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>bitwise and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>bitwise ex-or</td>
</tr>
<tr>
<td></td>
<td>^~ or ^^</td>
<td>bitwise ex-nor</td>
</tr>
<tr>
<td>Reduction</td>
<td>&amp;</td>
<td>reduction and</td>
</tr>
<tr>
<td></td>
<td>~&amp;</td>
<td>reduction nand</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>~</td>
<td></td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>reduction ex-or</td>
</tr>
<tr>
<td></td>
<td>^~ or ^^</td>
<td>reduction ex-nor</td>
</tr>
</tbody>
</table>
Operators: Summary

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational</td>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
<td>equality</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>inequality</td>
</tr>
<tr>
<td>Concatenation</td>
<td>{}</td>
<td>concatenation</td>
</tr>
<tr>
<td>Conditional</td>
<td>?:</td>
<td>conditional</td>
</tr>
<tr>
<td>Shift</td>
<td>&gt;&gt;</td>
<td>right shift</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;</td>
<td>left shift</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;&gt;</td>
<td>arithmetic right shift</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;&lt;</td>
<td>arithmetic left shift</td>
</tr>
</tbody>
</table>

Combinational Circuits: Verilog Examples
**Example 6.1: How to use inout Port**

Remember: *inout* can only be of type *wire*

```vhdl
`timescale 1ns/10ps
module ModuleName (IO, OUT_Enab);

inout IO; // Bi-directional port
input Out_Enab; // Input control
wire Din, Dout; // Internal circuit nodes
assign IO = Out_Enab ? Dout : 'bz; // Send data out
assign Din = Out_Enab ? 'b0 : IO; // Get data in
endmodule
```

**Example 6.2: 4-bit Adder**

The use of concatenation

```vhdl
`timescale 1ns/10ps
module EE216A_ADD4b (SUM, COUT, A, B, CIN);

output [3:0] SUM; // I/O Declaration
output COUT;
input [3:0] A, B;
input CIN;
assign {COUT, SUM} = A + B + CIN; // Func. Detail
endmodule
```
Example 6.3: 4-bit Unsigned Multiplier

```verilog
`timescale 1ns/10ps
module EE216A_UMUL4b (OUT, A, B);
output [7:0] OUT; // I/O Declaration
input [3:0] A, B;
assign OUT = A * B;
// Compiler appends zeros on A & B, by default
endmodule
```

Example 6.4: 4-bit Signed Multiplier

```verilog
`timescale 1ns/10ps
module EE216A_SMUL4b (OUT, A, B);
output [7:0] OUT; // I/O Declaration
input [3:0] A, B;
assign OUT = {{(4){A[3]}}, A} * {{(4){B[3]}}, B};
// Manually do sign extension on A & B
endmodule
```
Example 6.5: 2:1 Multiplexer

By Logical Operator

```
`timescale 1ns/10ps
module EE216A_Mux2
(O, A,B,SEL);
output O;
input A,B,SEL;
assign O = (~SEL & A) | (SEL & B);
endmodule
```

By Conditional Operator

```
`timescale 1ns/10ps
module EE216A_Mux2
(O, A,B,SEL);
output O;
input A,B,SEL;
assign O = SEL ? B : A;
endmodule
```

Behavioral Model
Preview: Dataflow vs. Behavioral Model

**Dataflow (All synthesizable)**
- Module, instance, port connection
- Net, register, number, assignment
- Vector, array, operator

**Behavioral (Not all can be synthesized)**
- Timing control basics (timescale, initial, always)
- Blocking/Non-blocking statement
- Activation list, edge/level-trigger logic
- Conditional statement, for loop
- Other syntax useful in testbench

**Remember**: Testbench NEVER gets synthesized

Next Week

**Lecture 7**: Latches & FFs

**Lecture 8**: Verilog 2
- Behavioral Model
- State Machines