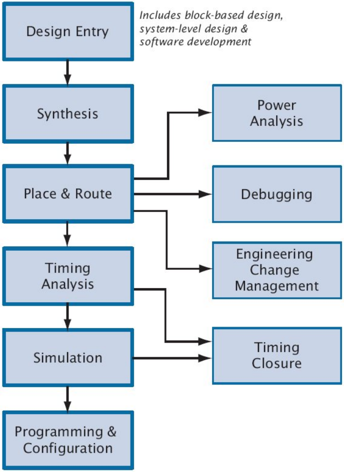
Verilog

Radek Pelánek and Šimon Řeřucha

Contents

Computer Aided Design

Figure 1. Quartus II Design Flow



Hardware Description Languages (HDL)

- Verilog – “C like”
- VHDL (VHSIC hardware description language) – “Pascal like”
- many others (less supported), e.g. AHDL (Altera HDL)

Specification, Simulation, Synthesis

Aims of hardware description languages (HDLs)

- Specification – original aim of HDLs
- Simulation – i.e. executable specifications
- Synthesis – added later

Note: not everything what can be specified can be synthetized

Structural vs Behavioral Modeling

- structural modeling
  - functionality and structure of the circuit
  - call out the specific hardware
- behavioral modeling
  - only the functionality, no structure
  - synthesis tool creates correct logic

Levels of Abstraction

Verilog can be used at four levels of abstraction:

algorithmic level	like C code with if, case, loop statements
register transfer level	registers connected by boolean equations
gate level	interconnected by and, nor, etc.
switch level	MOS trasistors inside gates

Syntax Example

```
/*
  toplevel module - intended to give a feeling of Verilog syntax.
*/

// module declaration
module toplevel(clock,reset,flop1);

 // port list
 input clock;
 input reset;
 output flop1;

 // data types
 reg flop1;
 reg flop2;

 // procedural block
 always @(posedge reset or posedge clock)
 if (reset)
   begin
     flop1 <= 0;
     flop2 <= 1;
   end
 else
   begin
     flop1 <= flop2;
     flop2 <= flop1;
   end
endmodule
```

Lexical Conventions

Similar to C programming language:

- white space — just a separator
- case sensitive; keywords are lower case
- identifiers start with alphabetic or underscore character, may contain numbers
- comments: `/* multiline */` `// singleline`

Numbers

- sized (default 32)
- radix: binary (b), octal (o), decimal (d), hexadecimal (h) (default decimal)
- syntax: `[size]'[radix][value]`
- underscore may be used for readability
- real numbers: decimal ([value].[value]) or scientific notation ([mantissa]E[exponent])
- negative numbers: minus sign before the size; internally represented in 2s complement

Numbers: Examples

syntax	stored as	description
1	00000000 00000000 00000000 00000001	unsized 32 bit
8'hAA	10101010	sized hex
32'hDEAD_BEEF	11011110 10101101 10111110 11101111	sized hex
6'b10_0011	100011	sized binary

Operators

Similar to C programming language, priorities usuall (see documentation):

- arithmetics: +, -, *, /, %, ...
- relational: $<, >, ==, \ldots$
- logical: $!, \&\&, ||$
- bit operators: $-, \&, |, \ldots$
- reduction operators (unary): $\&, |, \ldots$
- conditional: ? :
- concatenation ($\{,\}$), shift operators ($<<, >>$)

Four Valued Logic

- 0
- 1
- z – the high impedance output of tri-state gate; a real electrical effect
- x – unknown; not a real value, used in simulator as a debuging aid

Data Types

- nets – structural connections between components
  - wire – interconnecting wire, no special function
  - tri0, tri1, trireg, supply0, ...
- registers – represent variables used to store data; it does not represent a physical (hardware) register
  - reg – unsigned variable
  - integer – signed var 32 bits
  - real – double precision floating point

Bus Declarations

- Syntax: `data-type[MSB:LSB] name;`
- Examples:
  - `wire [15:0] in, out;`
  - `reg [7:0] tmp;`

Modules

- modules — building blocks
- design hierarchy — instantiating modules in other modules
- in Quartus you can mix verilog modules with other modules specified in other formalisms (e.g., schematic, VHDL)

Module Structure

```
module modulename(portlist);
  port declarations
  datatype declarations
  circuit functionality
  timing specification
endmodule
```

timing specification is for simulation

Ports

- communication between module and its environment
- all but top-level modules have ports
- ports can be associated by order or names
- port types: input, output, inout
- ports are by default wire

Parameterized Modules

```
module shift_n(it, out);
  input [7:0] it; output[7:0] out;
  parameter n = 2; // default value
  assign out = (it << n);
endmodule

//instantiations
wire [7:0] in1, out1, out2;
shift_n shft2(in1, ou1);
shift_n #(3) shft3(in1, ou2);
```

Gate Primitives

- and, nand, or, nor, xor, xnor – *n*-input gates
- examples:

```
and u1(out, in1, in2);
xor (out, in1, in2, in3);
```

- names are optional

Gate Primitives: Example

```
module myxor(input a, b, output x)
 and g1(p1, a, ~b),
     g2(p2, ~a, b);
 or (x, p1, p2);
endmodule
```

Continuous Assignment: Example

- abstractly models combinatorial hardware driving values onto nets
- syntax: `assign lhs = rhs;`
- always active, if any input changes, the assign statement is reevaluated

Continuous Assignment: Example

```
module fullAdder(input a, b, cin, output s, cout)
 assign
   s = a ^ b ^ cin,
   cout = (a & b) | (b & cin) | (a & cin);
endmodule
```

Procedural Blocks

initial block
- evaluated at the beginning of simulation
- variable initialization
- not supported for synthesis

always block
- continuously evaluated
- all always block in a module execute simultaneously

compound statements (multiple st.'s in block): begin, end keywords

Waiting

- delay: `#delay`
- event control: `@(list of events)`; statement is executed only after one specified events occur; events are: change of variable, `posedge`, `negedge`
- wait: `wait (condition)`; waits until the condition evaluates to true

Module Example

```
// module declaration

module up_counter(out, enable, clk, reset)

 // ports and datatype  declaration
 output [7:0] out;
 input enable, clk, reset;

 reg [7:0] out;         // auto-magically matches to ''out'' port

 // functionality
 always @(posedge clk)
  if (reset)
    out <= 8'b0 ;
  else if (enable)
    out <= out + 1;

endmodule
```

Procedural Assignments

- blocking (=)
  - sequential evaluation
  - use for combinatorial procedures
- non-blocking (<=)
  - parallel evaluation
  - assignment done after *all* right hand sides evaluated
  - use for `always @(posedge clk) ...` type procedures

Do not mix = and <= in one block.

Conditional Statement, Loops

Very similar to C programming language:

- if ... else
- for
- while
- forever
- repeat

If Statement

- if: 2-way branch, logical condition

```
always @(sela or selb or a or b or c)
        begin
                if (sela)
                        q = a;
                else
                if (selb)
                        q = b;
                else
                        q = c;
        end
```

Case Statement

- `case`: mutlipath branch, comparison to constant
- `default` block
- `casex`, `casez` variants: may include `z`, `x`, `?` in constant (don't cares)

Case: Example

```
always @(a or b or c or d or sel)
  begin
    case (sel)
      2'b00: mux_out = a;
      2'b01: mux_out = b;
      2'b10: mux_out = c;
      2'b11: mux_out = d;
    endcase
```

Casez: Example

```
casez (d)
  3'b1??: b = 2'b11;
  3'b01?: b = 2'b10;
  default: b = 2'b00;
endcase
```

Loops

- `repeat(`*n*`)`: loops *n*-times
- `while(`*cond*`)`: loops while condition *cond* holds
- `for(init,end,inc)`: does *init* assignment, loops while *end* condition holds, on every iteration does the *inc* assignment

```
always @(sela or selb or a or b or c)
for (i = 4; i <= 7; i = i + 1)
        begin
            result[i] = result[i-4];
        end
```

Unsupported for Synthesis

- `initial` blocks are ignored
- `delays` are ignored; e.g.: $a <= \#10b$ is synthetized as $a <= b$
- X and Z in four-valued logic

Coding Style - DO's

- Use meaningful names for signals and variables
- Use parentheses to optimize logic structure
- Use continuous assign statements for simple combo logic (not registered / direct)
- Use nonblocking for sequential and blocking for combo logic (registered)
- Define if-else or case statements explicitly

Coding Style – DON'T's

- Don't mix level and edge sensitive elements in the same always block
- Avoid mixing positive and negative edge-triggered flip-flops
- Don't mix blocking and nonblocking assignments in the same always block
- Be careful with multiple assignments to the same variable

Remember

- There's a difference in handling of direct and registered outputs
- Avoid driving single wire/register from multiple always blocks
- Top-level entity
- Set up your project properly (simulation mode, pin handling, ...)

Verilog Resources

Many tutorials available on-line, see e.g.,

- Introduction to Verilog (Peter M. Nyasulu)
- Verilog Tutorial (Deepak Kumar Tala)
- `http://l202.fi.muni.cz/vyuka/pv170/`