Digital Design with the Verilog HDL

Chapter 7: parameters, Task, and Function in Verilog

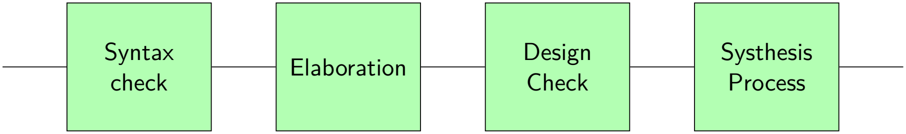
Binh Tran-Thanh

Department of Computer Engineering
Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology

May 26, 2023

Elaboration of Verilog Code

Synthesis Tool Flow



"Compile" in ModelSim	"Simulate" in ModelSim		
"Analyze" in Design Vision	"Ebolarate" in Design Vision	"Check Design" in Design Vision	"Compile" in Design Vision

Elaboration of Verilog Code

- Elaboration is a **pre-processing stage that takes place before code is synthesized.**
- It allows us to automatically alter our code before Synthesis based on **Compile-Time** information
- Uses of Elaboration
    - **Unrolling of FOR Loops**
    - **Parameterization**
    - **Code Generation**
    - **Constant Functions**
    - **Macros**
- Example ???

Overview

- Parameters
- Generated Instantiation
- Functions and Tasks

Parameters

- Compile-time constant parameters in Verilog
  - In Verilog: `parameter N=8'd100;`
  - **Values are substituted during Elaboration;**
  - **Parameters cannot change value after synthesis**
- Can be used for three main reasons
  - Make code more readable
  - Make it easier to update code
  - Improve (re)usability of modules

More Readable, Less Error-Prone

```
/* your BOSS's style  */




always @(*)begin
  case(mode)
    4'b0000: ...
    4'b0100: ...
    4'b0101: ...
    4'b1010: ...
    default: ...
  endcase
end
```

```
/* Team carrier's style */
parameter ADD = 4'b0000;
parameter SUB = 4'b0100;
parameter XOR = 4'b0101;
parameter AND = 4'b1010;

always @(*)begin
  case(mode)
    ADD: ...
    SUB: ...
    XOR: ...
    AND: ...
    default: ...
  endcase
end
```

Reusability/Extensibility of modules

```
module xor_array(y_out, a, b);
  parameter SIZE = 8, DELAY = 15;// parameter defaults
  output [SIZE-1:0] y_out;
  input [SIZE-1:0]a,b;
  wire #DELAY y_out= a ^ b;
endmodule

   // use defaults
xor_array G1 (y1, a1, b1);
   // override default parameters (SIZE = 4, DELAY = 5)
xor_array #(4, 5) G2(y2, a2, b2);
```

- module instantiations **cannot** specify delays without **parameters**
  - Where would delays go?
  - What type would they be?

Overriding parameters

- parameters can be overridden
  - Generally done to "resize" module or change its delay
- Implicitly: override in order of appearance
  - `xor_array #(4, 5) G2(y2, a2, b2);`
- Explicitly: name association (preferred)
  - `xor_array #(.SIZE(4), .DELAY(5)) G3(y2, a2, b2);`
- Explicitly: `defparam`
  - `defparam G4.SIZE = 4, G4.DELAY = 15;`
  - `xor_array G4(y2, a2, b2);`
- `localparam` parameters in a module **CAN NOT be overridden**
  - `localparam SIZE = 8, DELAY = 15;`

Parameters With Instance Arrays

```
module array_of_xor(y, a, b);
  parameter SIZE = 4;
  input [SIZE-1:0] a,b;
  output [SIZE-1:0] y;
  xor G3[SIZE-1:0] (y, a, b);// instantiates 4 xorgates
endmodule                      // (unless size overridden)

module variable_size_register(q, data_in, clk, set, rst);
  parameter BITWIDTH = 8;
  input [BITWIDTH-1:0] data_in;  // one per flip-flop
  input clk, set, rst;           // shared signals
  output [BITWIDTH-1:0] q;       // one per flip-flop
  // instantiate flip-flops to form a BITWIDTH-bit register
  flip_flop M [BITWIDTH-1:0] (q, data_in, clk, set, rst);
endmodule
```

Parameterized Ripple Carry Adder

```
module RCA(sum, c_out, a, b, c_in);
  parameter BITS = 8;
  input [BITS-1:0] a, b;
  input c_in;
  output [BITS-1:0] sum;
  output c_out;
  wire [BITS-1:1] c;
  Add_full M[BITS-1:0](sum, {c_out, c[BITS-1:1]}, a, b,
                                     {c[BITS-1:1], c_in});
endmodule
```

- Instantiate a 16-bit ripple-carry adder:
  `RCA #(.BITS(16)) add_16(sum, carryout, a, b, carryin);`

Parameterized Shift Left Register [1]

```
module shift(out, in, clk, rst);
  parameter BITS=8;
  input in, clk, rst;
  output [BITS-1:0] out;
  dff shiftreg[BITS-1:0](out, {out[BITS-2:0],in} , clk, rst
    );
endmodule
```

- Instantiate a 5-bit shift register:
  `shift #(.BITS(5)) shift_5(shiftval, shiftin, clk, rst);`

Parameterized Shift Left Register [2]

```
module shift_bhv(outbit, out, in, clk, rst);
  parameter WIDTH= 8;
  output reg[WIDTH-1:0] out;
  output reg outbit;
  input in, clk, rst;

  always @(posedge clk) begin
    if (rst) {outbit, out} <= 0;
    else {outbit, out} <= {out[WIDTH-1:0], in};
  end
endmodule
```

- Instantiate a 16-bit shift register: `shift_bhv #(16) shift_16( shiftbit, shiftout, shiftin, clk, rst);`

Parameters + Generate Statements

- Problem: Certain types of logic structures are efficient only in certain scenarios
- For example when designing an adder:
    - Ripple-Carry Adders are better for small operands
    - Carry Look-ahead Adders are better for large operands
- If we change a parameter to use a larger or smaller adder size, we may also want to change the structure of the logic

Generated Instantiation

- Generate statements: control over the instantiation/creation of
  - modules, gate primitives, continuous assignments, `initial` blocks, `always` blocks, nets and regs
- Generate instantiations are resolved during Elaboration
  - Can alter or replace a piece of code based on compile-time information
  - **Before** the design is simulated or synthesized
  - *Think of it as having the code help write itself*

Special Generate Variables

- Index variables used in generate statements; declared using `genvar` (e.g., `genvar i`)
- Useful when developing **parameterized** modules
- Can override the parameters to create different-sized structures
- Easier than creating different structures for all different possible bitwidths

Generate-Loop

A generate-loop permits making one or more instantiations (**pre-synthesis**) using a **for-loop**.

```
module gray2bin1 (bin, gray);
  parameter SIZE = 8; // this module is parameter izable
  output [SIZE-1:0] bin;
  input [SIZE-1:0] gray;
  genvar i;
  generate
    for(i=0; i<SIZE; i=i+1) begin: bit
      assign bin[i] = ^gray[SIZE-1:i]; // reduction XOR
    end
  endgenerate
endmodule
```

How does this differ from a standard for loop?

Generate-Conditional

A generate-conditional allows conditional (**pre-synthesis**) instantiation using **if-else-if** construct.

```
module multiplier(a, b, product);
  parameter A_WIDTH = 8, B_WIDTH = 8;
  localparam PRODUCT_WIDTH = A_WIDTH+B_WIDTH;
  input [A_WIDTH-1:0] a; input [B_WIDTH-1:0] b;
  output [PRODUCT_WIDTH-1:0] product;
  generate
    if((A_WIDTH < 8) || (B_WIDTH < 8))
      CLA_multiplier #(A_WIDTH,B_WIDTH) u1(a, b, product);
    else
      WALLACE_multiplier #(A_WIDTH,B_WIDTH) u1(a, b,
   product);
  endgenerate
endmodule
```

Generate-Case

A generate-case allows conditional (**pre-synthesis**) instantiation using **case** constructs.

```
module adder (output co, sum, input a, b, ci);
  parameter WIDTH = 8;
  generate
    case(WIDTH)
        // 1-bit adder implementation
      1: adder_1bit x1(co, sum, a, b, ci);
        // 2-bit adder implementation
      2: adder_2bit x1(co, sum, a, b, ci);
      default: adder_cla #(WIDTH) x1(co, sum, a, b, ci);
    endcase
  endgenerate
endmodule
```

Generate a Pipeline [Part 1]

```
module pipeline(output [BITS-1:0] out, input [BITS-1:0] in,
    input clk, rst);
  parameter BITS = 8;
  parameter STAGES = 4;
  wire[BITS-1:0] stagein [0:STAGES-1]; // value from
   previous stage
  reg[BITS-1:0] stage [0:STAGES-1];// pipeline registers
  assign stagein[0] = in;
  generate
    genvar s;
    for (s = 1; s < STAGES; s = s + 1) begin: stageinput
      assign stagein[s] = stage[s-1];
    end
  endgenerate
```

// continued on next slide

Generate a Pipeline [Part 2]

// continued from previous slide

```
  assign out = stage[STAGES-1];
  generate
    genvar j;
    for (j = 0; j < STAGES; j = j + 1) begin: pipe
      always @(posedge clk) begin
        if(rst) stage[j] <= 0;
        else stage[j] <= stagein[j];
      end
    end
  endgenerate
endmodule
```

What does this generate?

Functions and Tasks

- HDL constructs that look similar to calling a function or procedure in an HLL (High Level Language).
- Designed to allow for more code reuse
- There are 3 major uses for functions/tasks
  - To describe logic hardware in **synthesizable modules**
  - To describe functional behavior in **testbenches**
  - To compute values for parameters and other **constants** for synthesizable modules before they are synthesized
- When describing hardware, you must make sure the function or task can be synthesized!

Functions and Tasks in Logic Design

- It is critical to be aware of whether something you are designing is intended for a synthesized module
  - Hardware doesn't actually "call a function"
  - No instruction pointer or program counter
  - This is an abstraction for the designer
- In synthesized modules, they are used to describe the behavior we want the hardware to have
  - Help make HDL code shorter and easier to read
  - The synthesis tool will try to create hardware to match that description

Functions and Tasks in Testbenches

- Since testbenchesdo not need to synthesize, we do not have to worry about what hardware would be needed to implement a function
- Be careful: This doesn't mean that we can treat such functions & tasks as software
- Even testbench code must follow Verilog standards, including the timing of the Stratified Event Queue

Functions

- Declared and called within a module
- Used to implement combinational **behavior**
  - Contain no timing controls or tasks
  - **Can use behavioral constructs**
- inputs/outputs
  - At least one input , exactly one output
  - Return variable is the same as function name
    - Can specify type/range (default: 1-bit wire)
- Usage rules:
  - May be referenced in any expression (RHS)
  - May call other functions
  - Use automatickeyword to declare recursive functions

Constant Functions

- A special class of functions that can always be used in a synthesizable module
- **Constant functions** take only constant values (such as numbers or parameters) as their inputs.
    - All inputs are constant, so the output is also constant
    - The result can be computed at elaboration, so there is no reason to build hardware to do it
- Constant functions are useful when one constant value is dependent on another. It can simplify the calculation of values in parameterized modules.

Function Example

```
module word_aligner(output [7: 0]word_out, input [7: 0]
    word_in);

  assign word_out= aligned_word(word_in); //invoke function
  function [7: 0] aligned_word; // function declaration
    input [7: 0] word;
    begin
      aligned_word = word;
      if (aligned_word != 0)
         while(aligned_word[7] == 0)
            aligned_word = aligned_word<< 1;
    end
  endfunction
endmodule
```

What sort of hardware might this describe? Do you think this will synthesize?

Function Example

```
module arithmetic_unit(result1, result2, operand1, operand2
   );
  output [4: 0] result1;
  output [3: 0] result2;
  input [3: 0] operand1, operand2;

  assign result1 = sum_of_operands(operand1, operand2);
  assign result2 = larger_operand(operand1, operand2);

  function[4: 0] sum_of_operands(input [3: 0] operand1,
   operand2);
    sum_of_operands = operand1 + operand2;
  endfunction

  function[3: 0] larger_operand(input [3: 0] operand1,
   operand2);
    larger_operand = (operand1 >= operand2) ? operand1 :
   operand2;
  endfunction

endmodule
```

Constant Function Example

```
module register_file(...);
  parameter NUM_ENTRIES = 64;

  localparam NUM_ADDR_BITS = ceil_log2(NUM_ENTRIES);

  function[31: 0] ceil_log2(input [31: 0] in_val);
    reg sticky;
    reg[31:0] temp;
    begin
       sticky = 1'b0;
       for (temp = 32'd0; value>32'd1; temp = temp+1) begin
         if((value[0]) & (|value[31:1]))
            sticky = 1'b1;
         value = value>>1;
       end
       clogb2 = temp + sticky;
    end
endfunction
```

Tasks

- Declared within a module
  - Only used within a behavior
- Tasks provide the ability to
  - Describe common behavior in multiple places
  - Divide large procedures into smaller ones
- Tasks are not limited to combinational logic
  - Can have time-controlling statements (`@`, `#`, `wait`)
- Some of this better for testbenches
  - Use `automatic` keyword to declare ”reentrant” tasks
- Can have multiple outputs, inout ports
- Local variables can be declared & used

Task Example [Part 1]

```
module adder_task (c_out, sum, clk, rst, c_in, data_a,
    data_b);
  output reg[3: 0] sum;
  output regc_out;
  input [3: 0] data_a, data_b;
  input clk, rst, c_in;

  always @(posedge clk or posedge rst) begin
    if (rst) {c_out, sum} <= 0;
    else add_values (c_out, sum, data_a, data_b, c_in); //
    invoke task
  end

  task add_values;// task declaration
    output regc_out;
    output reg[3: 0] sum
    input [3: 0] data_a, data_b;
    input c_in;
    {c_out, sum} <= data_a+ (data_b+ c_in);
  endtask
```

Task Example [Part 2]

- Could have instead specified inputs/outputs using a port list.

```
task add_values(output regc_out, output reg[3: 0] sum,
    input [3:0] data_a, data_b, input c_in);
```

- Could we have implemented this as a function?

Function Example

```
module adder_func(c_out, sum, clk, rst, c_in, data_a,
    data_b);
  output reg[3: 0] sum;
  output reg c_out;
  input [3: 0] data_a, data_b;
  input clk, rst, c_in;

  always @(posedge clk or posedge rst) begin
    if (rst) {c_out, sum} <= 0;
    else {cout, sum} <= add_values(data_a, data_b, c_in);
   // invoke function
  end

  function[4:0] add_values;// function declaration
    input [3: 0] data_a, data_b;
    input c_in;
    add_values = data_a + (data_b + c_in);
  endfunction
endmodule
```

How does this differ from the task-based version?

Task Example

```
task leading_1(output reg[2:0] position, input [7:0]
    data_word);
  reg[7:0] temp; //internal task variable
  begin
    temp = data_word;
    position = 7;
    while(!temp[7]) begin
      temp = temp << 1;
      position = position -1;
    end
  end
endtask
```

- NOTE: “while” loops usually not synthesizable!
- What does this task assume for it to work correctly?
- How do tasks differ from modules?
- How do tasks differ from functions?

Distinctions between tasks and functions

The following rules distinguish tasks from functions:

- A function shall execute in one simulation time unit; a task can contain time-controlling statements.
- A function cannot enable a task; a task can enable other tasks and functions.
- A function shall have at least one input type argument and shall not have an output or inout type argument; a task can have zero or more arguments of any type.
- A function shall return a single value; a task shall not return a value.

The purpose of a function is to respond to an input value by returning a single value. A task can support multiple goals and can calculate multiple result values.