Lecture 15: Register Transfer Language and Verilog

COS / ELE 375

Computer Architecture and Organization

Princeton University
Fall 2015

Bochao Wang
(Based on slides by David Penry and Neil Vachharajani)
(Prof. David August)

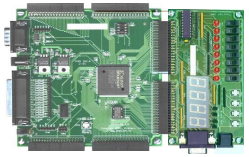
RTL (Register Transfer Language)

- Designing processors at the gate level is difficult
- Use a higher-level language

RTL: a language for describing the behavior of computers in terms of step-wise register contents

RTL Tools

RTL, just like (most) programming languages:

- Precise and unambiguous
- Programmer must debug
- Code can be checked automatically for certain properties
  - Type checking
  - Check RTL model against more abstract state machine
- Tools can process the code
  - RTL → simulator
  - RTL → processor
  - RTL → 

Hardware Description Languages

- Used to describe hardware for simulation and synthesis
- Why? Designs are too big to just draw schematics

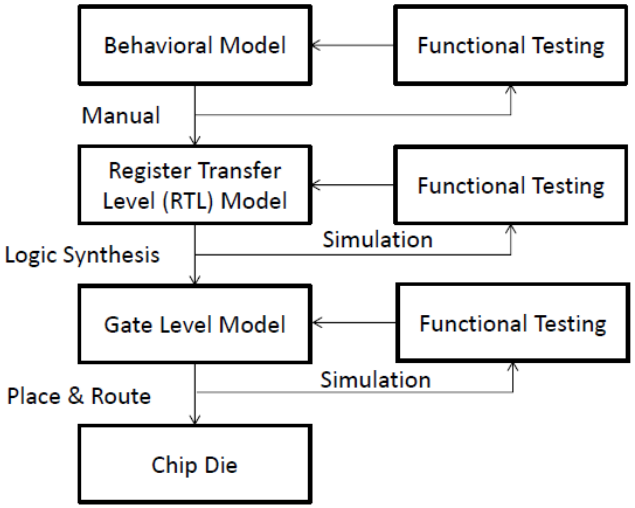
- Major languages:
  - Verilog
    - Cadence, Inc.
    - Emphasis on practical use (I/O well-defined, ability to record)
    - Possibly most widely used
  - VHDL
    - Department of Defence
    - Emphasis on abstraction (derived from Ada)
    - Popular in academic circles, Europe; required of defence contractors
  - SystemC
    - Synopsys, Inc.
    - Emphasis is "higher-level" simulation – really just a C++ template library
    - Up-and-coming

Simulation vs. Synthesis

- HDLs are used for both simulation and synthesis
- Simulation: does the design work?
  - Can also be used for "does the design meet timing?"
- Synthesis: generate a circuit that is equivalent
  - Recognize state and combinational logic
  - Optimize the circuit

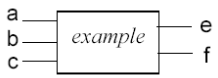
Not all valid HDL programs are synthesizable

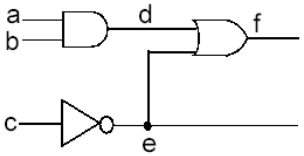
Verilog Design Flow



Example

Modules have initial, continuous, and always blocks

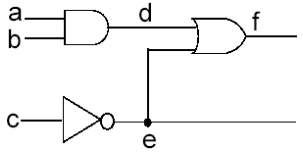





```
module example(a, b, c, f, e);
  input a, b, c;
  output f, e;
  wire d;
  and g1(d, a, b);
  not g2(e, c);
  or  g3(f, d, e);
endmodule
```

Youpyo Hong, Dongguk University

Modified Example




```
module example(a, b, c, f, e);
  input a, b, c;
  output f, e;
  assign d = a & b;
  assign e = ~c;
  assign f = d | e;
endmodule
```

Youpyo Hong, Dongguk University

- Use assign statement for combinational circuit
- Note the bit-wise operators (implies wire width…)

Elements of Verilog

Basic syntax

- `;` is the statement terminator (terminator, not separator)
- `/* */` for multi-line comments
- `//` can be used for single-line comments
- Constants can have bit-width and radix:
  <bit-width>'<radix><val>
  `32'h1234ab34` = $1234ab34_{16}$
  `8'b0100_0110` = 01000110_2 ; note that underlines are legal

Datatypes

- 4-valued logic: 0, 1, x, z
  - 0, 1 – normal binary 0 and 1
  - z – high-impedance value: this is what tri-state buffers drive when they are off
  - x – simulator doesn't know; sometimes used as "don't care"
- Integers – don't use these outside of test benches
- At simulation start, all signals have x's in every bit
- Unconnected inputs to a module have value 'z'.

Module Overall Structure

```
module the_design ( input, output );
   Declarations: ports, constants, variables(wire, reg)
   Instantiations of other modules
   Continuous assignment: assign y = ...
   Behavioral statements (initial, always) {
      procedural blocking assignment
      procedural nonblocking assignment
      }
endmodule
```

Signals

- Signal: a "variable" that can have values assigned "ahead of time"
- Can be thought of as a wire or a group of wires
- Syntax:
  <kind> [<width specifier>] name, name, …;
- Examples:

```
reg [15:0] IR;
wire [3:0] a; a[3] is MSB
wire [1:8] b; b[1] is MSB //not recommend
wire doit;
```

- Registers vs. wires:
  - Reg: inside "always" block, lhs, e.g. IR = …
  - Wire: outside "always" block, lhs, e.g. assign a = …

Wires

- Wires need not be declared unless it is multiple wires

```
module example(b, c);
  input b;
  output c;
  wire a;            // not necessary
  wire [3:0] d;      // necessary
  …
  assign c = b | a;
endmodule
```


Register Transfer Language?

- Variables correspond to the hardware registers
- "always" blocks

```
Module D_FlipFlop(Q, D, CLK);
  output Q;
  reg Q;
  input D, CLK;
  always @(negedge CLK) // Sensitivity List
  begin  // Not needed for 1 statement
    Q <= D;  // Note: No assign
  end
endmodule
```


Module Overall Structure

```
module the_design ( input, output );
   Declarations: ports, constants, variables(wire, reg)
   Instantiations of other modules
   Continuous assignment: assign y = ...
   Processes(initial, always) {
      procedural blocking assignment
      procedural nonblocking assignment
      }
endmodule
```

Modules

- Allow you to organize the design hierarchically
- Allow structural reuse of the design

- Defining a module:
  - Example:

```
module flop (clk, D, Q);  // name, list of ports
  input clk;       // define directions of ports
  input D;
  output Q;

…. // code to do things

  endmodule
```

- Instantiating a module:

```
flop U1 (.clk(someclk), .D(someD), .Q(someQ));
```


Module Overall Structure

```
module the_design ( input, output );
   Declarations: ports, constants, variables(wire, reg)
   Instantiations of other modules
   Continuous assignment: assign y = ...
   Processes(initial, always) {
      procedural blocking assignment
      procedural nonblocking assignment
      }
endmodule
```


Kinds of assignments

- Continuous (`assign =`)
  - Only to "wire"
  - Always sensitive to things on the right-hand side

- Blocking (`=`)
  - Made inside a process
  - Only to "reg"
  - Value of lhs changes immediately

- Non-blocking (`<=`)
  - Made inside a process
  - Only to "reg"
  - Value of lhs changes only after all rhs have been evaluated
  - A time-spec after the statement says to wait before changing the value
  - Continue executing process even if there is a time-spec

Assignment

`a = b`	Do Sequentially (blocking, sequentially)
`A <= b`	Do RHS first (nonblocking, grab at t=0)

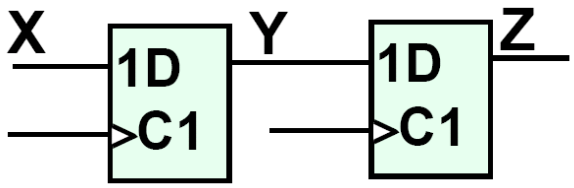
Which 3 are Shift Registers?

`Z<=Y; Y<=X;`

`Y<=X; Z<=Y;`

`Z=Y; Y=X;`

`Y=X; Z=Y;`



Processes

- A group of sequential statements; three kinds:
  - Initial - runs only at simulation start
  - Always – runs over and over
  - Continuous – special syntax for a common kind of always block
- Can be suspended, waiting for something to happen
  - For a signal in a group of signals to change
  - For an amount of time to pass
  - For a signal to become true (I've never needed this)

Process examples

```
reg rst_l, Q;
wire cachehit;

initial begin
  rst_l = 0;
  #300;
  rst_l = 1;
end

always @(posedge clk)
  Q <= #1 D;

assign cachehit = comparator_matches & valid;
```


Assignment examples

```
reg rst_l, Q;
wire cachehit;

initial begin
  rst_l = 0;       // blocking
  #300;
  rst_l = 1;       // blocking
end

always @(posedge clk)
  Q <= #1 D;  // non-blocking

assign cachehit = comparator_matches &
                  valid; // continuous
```


Blocking a process

- Wait for signals to change:
  ```
  always @(signal1 or signal2)
  ```
- Wait for a signal edge:
  ```
  always @(posedge clk or negedge rst_l)
  ```
- Wait for time:
  ```
  # 300;
  # 300 a = b;     // can also put before a statement
  ```
- Wait for logical value:
  ```
  wait(b);      // I've never used this
  ```

Operators

- Most of the operators look like C:
  - Bitwise operators: &, |, ~, ^
  - Logical operators: &&, ||, !
  - Comparisons: ==, !=, <, <=, >, >=
    - If either operand is x or z, they return FALSE.
  - Arithmetic: +, -, *, /, %, <<, >>
  - Choice: ?:
- Order of operations may differ from C; use parenthesis
- Special operators:
  - Comparison with x or z values: ===, !==
  - Bitwise operators can be used as reduction operators:
    `(|x)` = or all the bits of x together.

Statements

- Control structures
  - C-like: `if, while, for`
  - `forever` – do following statement forever
  - `repeat (<#>)` – do following statement a number of times
  - `case` (see next slide)
- Parallel control structures
  - `fork`/`join` – do statements in between in parallel
- Signal overrides
  - `force` – override the value of a signal
  - `release` – stop overriding the value of a signal

Case statements

- Rather different from C switch statements:

```
case (myvar)
1 : dothis = 1;
2 : dothat = 3;
3 : dosomemore = 4;
default: noneoftheabove = 1;
endcase
```

- Note that there are no fall-throughs
- Multiple statements inside one case require `begin`/`end` around them

Example: 4 to 1 Mux

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
  output out;
  input i0, i1, i2, i3;
  input s1, s0;
  reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
  begin
    case ({s1, s0})
      2'b00: out = i0;
      2'b01: out = i1;
      2'b10: out = i2;
      2'b11: out = i3;
      default: out = 1'bx;  // x is don't care
    endcase
  end
endmodule
```

Parameters

- Increases ability to reuse modules
- Defining a parameter (with a default value):

```
parameter me = 3;
```

- Overriding the default value

```
defparam myunit.me = 4;
```

  - Can also use a #(value) syntax when instantiating for modules with small numbers of parameters:
    e.g. module my_module #(parameter BIT_WIDTH = 32, …)
    (input clk, input [BIT_WIDTH-1:0] addr, output a…)

Using the mux module

- It's a 4-input mux for busses with a 'width' parameter for the size of the bus

```
wire [7:0] A,B,C,D, result;
wire [1:0] sel;

mux4 mymux (.out(result), .in0(A), .in1(B),
  .in2(C), .in3(D), .sel(sel));
defparam mymux.width = 8;
```

OR

```
mux4 #(8) mymux (.out(result), .in0(A), .in1(B),
   .in2(C), .in3(D), .sel(sel));
```


Preprocessor directives

- Always begin with a back-tick
- `` `define name value ``
  - defined values are referenced as `` `name ``

A Sample Finite State Machine

```
reg [1:0] state;
`define IDLE 2'b00
`define READ 2'b01
`define WRITE 2'b10
`define DONE 2'b11

always @(posedge clk or negedge rst_l)
  if (~rst_l) state <= `IDLE;
  else
    case (state)
    `IDLE:
        if (req)
          if (wr) state <= `WRITE;
          else state <= `READ;
        else state <= `IDLE;  // not strictly necessary
    `READ:
        state <= `DONE;
    `WRITE:
        state <= `DONE;
    `DONE:
        state <= `IDLE;
    default:
        state <= `IDLE;
    endcase
```

Dealing with Synthesis

- Things you can't synthesize reliably:
  - Some for loops (hard to guess)
  - @() in the middle of a process
  - Time specifiers
  - force/release
- Watch out for hidden latches!

```
always @(a)
  c = a | b;
```

  - Actually creates a latch
  - All things on the rhs must go on the sensitivity list unless you are trying to create a state element
- Using don't cares:
  - If you really don't care about a value, assign x to it, and the synthesis tool will choose a value which simplifies the logic

My Rules

- Make all flops (edge-sensitive always blocks) use a non-blocking assigment with #1 delay
  - Easier to see what's going on in waverforms
- Make all other always blocks use blocking assignment and use complete sensitivity lists
  - always @(*) for next state logic combinational
  - always @(posedge clk) for sequential
- Do combinational logic in continuous assignments except for next state logic

Another Sample Finite State Machine

```
module fsm
(
                input           clk,
                input           rst,
                input           in,
                output          redOut,
                output          yellowOut,
                output          greenOut
);

localparam RED = 0;
localparam YELLOW = 1;
localparam GREEN = 2;

reg [1:0]       state_f;
reg [1:0]       state_next;

// Sequential Logic
always @ (posedge clk)
begin
                if (rst)
                  state_f <= RED;
                else
                  state_f <= state_next;
end
```

```
// Combinational Logic
always @ *
begin
                state_next = state_f;
                case (state_f)
                  RED:
                    if (in)
                      state_next = GREEN;
                  YELLOW:
                    if (in)
                      state_next = RED;
                  GREEN:
                    if (in)
                      state_next = YELLOW;
                  default: state_next = RED;
                endcase
end

assign redOut = (state_f == RED);
assign yellowOut = (state_f == YELLOW);
assign greenOut = (state_f == GREEN);

endmodule
```


Thank you

Using the regfile2 module

- You should use this module for your register file.
  - It synthesizes properly
  - It has a nice task for displaying the register file contents
- Interesting characteristics:
  - One read ports, one read/write port
  - Port A writes if the enable bits are set (bytes are controlled individually); it will always read as well
  - Port B can only read
  - Clocked on the positive edge of the clock (you can't do the write in first half, read in second half trick from P & H)
- If you name your datapath DATA (the instance name, not the module name), and make IR_Enable a signal indicating when the IR is to be loaded with a new instruction, you can uncomment code in monitors.v to get automatic register dumps before each instruction.

Functions and tasks

- Functions are like C functions, but cannot have side effects
- Tasks do not have return values, but can have side effects
- Examples:

```
function [31:0] add1;
  input [31:0] x;
begin
  add1 = x + 1;
end
endfunction

task addtask;
  input [31:0] x;
  output [31:0] y;
begin
  y = x + 1;
  mycrazysignal = 3;
end
endtask
```


I/O

- Displaying data:

  `$display(<format string>, ….);`
  - Like printf, but
    - %b gives binary; %h gives hex
    - %t formats a time value
    - automatically puts on a newline
- Recording signals:
  - `$dumpfile(<file name>);` – set the dumpfile name
  - `$dumpvars(<# levels>, <hierarchy>);`
    - Adds signals below a particular point in the hierarchy to the list of signals to dump; only decends the hierarchy for the number of levels specified; 0 means no limit
  - `$dumpon;` - turns on dumping
  - `$dumpoff;` - turns off dumping

Simulation control

- `$finish;` – end simulation and exit the simulator
- `$stop;` – stop simulating; go to the command-line interface