Verilog Review

Michael McKeown
ELE/COS 475 – Computer Architecture
Princeton University
Spring 2014
Agenda

• Verilog Review
  – Overview
  – Programming paradigm
  – Syntax
  – Testbenches
  – Design advice
  – Common mistakes

• ELE/COS 475 Verilog Infrastructure
  – iverilog, gtkwave, PARC processor

• Python Preprocessor for Verilog

• PBS Job Resource Manager on Adroit

• Lab 1 – Practice Lab
  – Iterative multiplier and divider

• Any other class related questions
Verilog Review
Verilog Overview

• What
  – Hardware Description Language (HDL)
  – Textual description of hardware circuits
    • No schematics or graphical symbols
  – Model digital circuits
    • Behaviorally or structurally

• Why
  – Easy to prototype hardware
  – Easy to test hardware (simulation)
  – Widely used in industry
  – C-like syntax
  – Flexible

• How
  – Keep hardware mindset
  – Design parallel hardware, not sequential programs
Verilog in Hardware Design Process

- Behavioral Model
  - Manual
  - Register Transfer Level (RTL) Model
    - Logic Synthesis
    - Gate Level Model
      - Place & Route
      - Chip Die

- Functional Testing
  - Simulation
  - Functional Testing
  - Functional Testing

Simulation
Programming Paradigm - Module

Inputs

Verilog Module

Outputs
Programming Paradigm - Hierarchy

parc_Core

- parc_CoreCtrl
- parc_CoreDpath
  - imuldiv_IntMulDivIterative
  - parc_CoreDpathRegfile
  - parc_CoreDpathAlu
    - parc_CoreDpathAluShifter
    - parc_CoreDpathAluLogical
Modelling Digital Circuits

• Behavioral Level
  – Higher-level
    • Easier to write and understand
  – Register transfer level (RTL), High-level behavior
  – Not always synthesizable!!!

• Structural Level
  – Lower-level
    • Specifies all the details (gates, flip-flops, etc.)
  – Always synthesizable!!!

• Fine line

• We will mostly be using behavioral-level
Verilog Syntax - Logic Values and Literals

• Four value logic system
  – 0
  – 1
  – X – Don’t care/Unknown
  – Z – High impedance/Floating

• Verilog literals
  – Binary: 1’b0, 8’b1111_0000 (can add ‘_’ for readability)
  – Hexadecimal: 8’hF0
  – Decimal: 8’d240
  – There are others, but these are most useful
Verilog Syntax – Data Types

• **Wire** – connect structures together, no state
  – Single Wire
    • `wire my_wire;`
  – Vector of wires
    • `wire [7:0] my_wires;`
    • What about `wire [0:7] my_wires;`?
    • Or `wire [10:3] my_wires;`?
• **Reg** – for procedural assignments, retains value until updated again
  – Similarly, can be a single reg or an array of regs
    • `reg my_reg; or reg [3:0] my_regs;`
  – The use of reg does not necessarily imply a hardware flip-flop
    • `always @ (*), variables declared as reg but are actually wires`
• Can access specific bits of both wires and regs
  – `my_wire[3:0] or my_reg[1:0]`
  – `my_wire[0] or my_reg[3]`
Verilog Syntax – Data Types

• Parameter – to parameterize modules
  – `parameter WIDTH = 64;
  – `reg result[WIDTH-1:0];
  – Parameters for a module can be overridden during instantiation
  – Use `localparam to specify a parameter that may not be overridden

• Macros – constants and functions
  – `define MY_MACRO 4'b1100
  – a = b + `MY_MACRO

• There are other types, but these are the most common
Verilog Syntax - Operators

- **Arithmetic Operators**
  - Two types: binary and unary
  - Binary
    - Add (+), subtract (-), multiply (*), divide (/), power (**) , modulus (%) 
  - Unary
    - Positive (+), negative (-) (2’s complement)
    - Sign of an operand: -4, +5

- **Logical Operators**
  - Logical and (&&), Logical or (||), logical not (!)
  - Evaluate to 0 (false), 1 (true), or x (ambiguous)
  - Anything not equal to 0 is interpreted as 1

- **Relational Operators**
  - Greater than (>), less than (<), greater than or equal (>=), less than or equal (<=)
  - Evaluate to 0 (false), 1 (true), or x (ambiguous)
Verilog Syntax - Operators

• Equality Operators
  – Logical equality (==), logical inequality (!=), logical case equality (===), logical case inequality (!==)
  – Evaluate to 0 (false), 1 (true), or x (ambiguous)
  – Logical case allows checking for x and z
    • Do not use!!
    • Definitely not synthesizable

• Bitwise Operators
  – Negation (~), and (&), or (|), xor (^), xnor (^-, -^)

• Reduction Operators
  – And (&), nand (~&), or (|), nor (~|), xor (^), xnor (^~, ~^)

• Shift Operators
  – Right shift (>>, left shift (<<), right shift arithmetic (>>>), left shift arithmetic (<<<)
Verilog Syntax - Operators

• Concatenation Operators
  – {,} appends signals together
  – a = {b, c}

• Replication Operators
  – {{}} repetitive concatenation of the same value
  – a = {4{b}}

• Conditional Operators
  – ? : operates like the C syntax
  – Conditional expression ? True expression : false expression;
  – Good for multiplexers and tri-state buffers

• Not all operators are synthesizable!!

• X-propagation
module my_module
(
    clk,
    addr,
    din,
    ...
    valid,
    dout,
    stall,
    ...
    bus,
    ...
);

input          clk;
input [31:0]   addr;
input [31:0]   din;
...
output         valid;
output [31:0]  dout;
output reg [31:0] stall;
...
inout          bus;
...

endmodule
Verilog Syntax – Module Structure

module my_module
(
    input clk,
    input [31:0] addr,
    input [31:0] din,
    ... 
    output valid,
    output [31:0] dout,
    output reg stall,
    ... 
    inout [31:0] bus,
    ... 
);

... 

endmodule
module my_module
#
    parameter BIT_WIDTH = 32,
    ...
)
(
    input         clk,
    input [BIT_WIDTH-1:0] addr,
    input [BIT_WIDTH-1:0] din,
    ...
    output        valid,
    output [BIT_WIDTH-1:0] dout,
    output reg    stall,
    ...
    inout [BIT_WIDTH-1:0] bus,
    ...
);
...
endmodule
Verilog Syntax – Module Instantiation

my_module mod
(
    clk_in,
    addr_in,
    din,
    ...
    valid_out,
    dout,
    stall_out,
    ...
    bus_inout,
    ...
);

Verilog Syntax – Module Instantiation

my_module mod
(
  clk_in,
  addr_in,
  din,
  ...
  valid_out,
  dout,
  stall_out,
  ...
  bus_inout,
  ...
);

Verilog Syntax – Module Instantiation

my_module mod
(
    .clk(clk_in),
    .addr(addr_in),
    .din(din),
 ...
    .valid(valid_out),
    .dout(dout),
    .stall(stall_out),
 ...
    .bus(bus_inout),
    ....
);

Verilog Syntax – Module Instantiation

my_module
#(
    .BIT_WIDTH (64),
    ...
) mod
(
    .clk(clk_in),
    .addr(addr_in),
    .din(din),
    ...
    .valid(valid_out),
    .dout(dout),
    .stall(stall_out),
    ...
    .bus(bus_inout),
    ...
);
my_module
#

  .BIT_WIDTH (64),
    ...

) mod [3:0]
(

  .clk(clk_in),
  .addr(addr_in),
  .din(din),
  ...
  .valid(valid_out),
  .dout(dout),
  .stall(stall_out),
  ...
  .bus(bus_inout),
  ....

);}
my_module
#
.
.BIT_WIDTH (64),
...
) mod [3:0]
(
.
.clk(clk_in),
.addr(addr_in),
.din(din),
...
.
.valid(valid_out),
.
.dout(dout),
.
.stall(stall_out),
...
.
.bus(bus_inout),
....
);

What if din is more than one bit?
So now we know the basic data types, basic operators, how to structure a module, and how to instantiate modules.....

How do we create functionality?
Verilog Syntax – Always @ Block

```verilog
always @ (...) sensitivity list ...)
begin
    ... elements ...
end

always @ (...) sensitivity list ...)
begin
    B <= A;
    C <= B;
    D <= C;
end

always @ (...) sensitivity list ...
begin
    B = A;
    C = B;
    D = C;
end
```

[ Non-blocking ]

[ Blocking ]
Verilog Syntax – Always @ Edge

- Sequential Logic, or registers
- Use non-blocking (<=)
- Only regs on LHS
- Change value at positive or negative edge of clock

```verilog
always @ (posedge clk)
begin
    q <= din;
end

always @ (negedge clk)
begin
    q <= din;
end

always @ (posedge or negedge clk)
begin
    q <= din;
end

always @ (posedge clk)
begin
    if (rst)
        q <= 1'b0;
    else if (en)
        q <= din;
end
```
Verilog Syntax – Always @ *

- Combinational Logic, or logic gate
- Use blocking (=)
- Only regs on LHS
- Infer an element(s) that changes its value as soon as one or more of its inputs change
- Will generate a latch if you do not assign every element a value under any condition

\[
\text{always @ (A or B)} \\
\text{begin} \\
\text{C = A & B;} \\
\text{end}
\]

\[
\text{always @ *} \\
\text{begin} \\
\text{C = A & B;} \\
\text{end}
\]
Verilog Syntax – Blocking vs. Non-Blocking

// Blocking
always @ (posedge clk)
begin
    a = b;
    b = a;
end

// Non-Blocking
always @ (posedge clk)
begin
    a <= b;
    b <= a;
end

What is the difference between the two?
What is the result of each?
Verilog Syntax – Continuous Assign

- Combinational logic, or logic gate
- Continuous assignment using “assign” and “=“
- Only wire on LHS
- “=“ is blocking assignment – it takes place immediately
  - Can cause other events to trigger on the same cycle

```
assign C = A & B;
```

```
assign C = SEL ? B : A;
```
Verilog Syntax – Other Syntactical Structures

• if/else if/else

```verilog
always @ (*)
begin
  if (sel)
    C = B;
  else
    C = A;
end

always @ (*)
begin
  case (sel)
  1'b0: C = A;
  1'b1: C = B;
  endcase
end
```

• case (value)

• initial blocks
  – Not synthesizable

```verilog
initial
begin
  rst = 1'b0;
  #5
  rst = 1'b1;
  #20
  rst = 1'b0;
end
```
Verilog Syntax – Non Synthesizable Subset

• A large part of Verilog is not synthesizable into hardware
  – For example, the divider in Lab 1

• Still useful for
  – Prototyping functionality that will later be elaborated
  – Making testbenches that are not synthesizable

• The code that implements real hardware should be synthesizable
Verilog Syntax – Non-Synthesizable Subset

// Division
always @ (posedge clk) begin
  q = d / 7;
end

Not synthesizable, but often useful for quick prototyping

// Delay
always begin
  #1 clk = ~clk;
end

Not synthesizable, but often used to create testbenches
Verilog Syntax – Simple Examples

// Full Adder
module fa
(input a,
 input b,
 input cin,
 output s,
 output cout
);
assign s = a ^ b ^ cin;
assign cout = (a & b) | (a & cin) | (b & cin);
endmodule

// Shift register
module shiftreg
(input clk,
 input rst,
 input step,
 input shiftIn,
 output reg [3:0] regOut
);
always @ (posedge clk)
begin
if (rst)
regOut <= 4'b0;
else if (step)
regOut <= {regOut[2:0], shiftIn};
end
endmodule
Verilog Syntax – FSM Example

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
Verilog Testbenches

• Each module should have a corresponding testbench
  – A testbench is a new module containing only the design under test (DUT) and a comprehensive set of varying inputs
  – Outputs should demonstrate the module works as expected
Verilog Testbenches - Example

module foo
(
    input clk,
    input d,
    output q
);

initial
begin
clk = 0;
d = 0;
#20 $finish;
end

always begin
    clk = ~clk;
end

always begin
    d = ~d;
end

endmodule

module footb;
    reg clk;
    reg d;
    wire q;

    $dumpfile("waveform.vcd");
    $dumpvars(0, footb);

    foo dut (.clk(clk), .d(d), .q(q));
endmodule
Verilog Debugging

- **$display** – Print to stdout
  - $display ($time, "Count changed to %d", count);
- **$monitor** – Watch for any changes to a variable
  - $monitor ($time, "Count is now %d", count);
- **$dumpfile and $dumpvars** – dump all signal values to a file for later viewing
  - $dumpfile("output.vcd");
  - $dumpvars (0, module_to_dump_signals);
Verilog Design Advice

• Always design the hardware first

• Leaf vs non-leaf modules
  – Leaf modules contain flip-flops, logic, etc.
  – Non-leaf modules just instantiate sub-modules and connects them together
  – Keeps hierarchy clean

• Don’t mix structural and behavioral code

• Write testbenches for every module

• Comment liberally
Verilog - Common Mistakes

```verilog
always @ (posedge clk)
begin
    if (rst)
        q <= 1'b0;
    if (en)
        q <= d;
end
```

What is the output?
Verilog – Common Mistakes

always @ (posedge clk)
begin
  q1 = d;
  q2 = q1;
  q3 = q2;
end

What circuit does this code model?
Verilog – Common Mistakes

always @ (posedge clk)
begin
    q1 = d;
    q2 = q1;
    q3 = q2;
end

What circuit does this code model?

Moral: Only use non-blocking assignment for sequential logic!
Verilog – Common Mistakes

```verilog
always @ (posedge clk)
    q1 = d;
always @ (posedge clk)
    q2 = q1;
always @ (posedge clk)
    q3 = q2;
```

What circuit does this code model?
always @ (posedge clk)
    q1 = d;
always @ (posedge clk)
    q2 = q1;
always @ (posedge clk)
    q3 = q2;

What circuit does this code model?

Moral: Understand where race conditions occur and how to avoid them!
ELE/COS 475 Verilog Infrastructure
ELE/COS 475 Verilog Infrastructure

• Icarus Verilog (iverilog)
  – Open-source Verilog simulation and synthesis tool
  – Compiled simulator
  – Compiles Verilog to vvp assembly
  – vvp executes the compiled vvp assembly
    • Writes VCD-format log file as output
• gtkwave is an open-source waveform viewer that displays VCD (and other) files graphically
  – Great for debugging Verilog code
• You can install both of these tools locally
  – Available for Linux, Mac OS X, and Windows
ELE/COS 475 Verilog Infrastructure

• PARC Processor
  – Educational processor developed at Cornell
  – Models a simplified MIPS32-like instruction set
    • Essentially MIPS without branch delay slots
  – Our goal will be to improve the microarchitecture for better performance
  – C compiler and assembler available on adroit
    • We have binaries available for distribution for Linux x86_64
    • We do not have permission to distribute the source code
Python Preprocessor for Verilog
Python Preprocessor for Verilog

• PyHP for short

• What
  – Generate Verilog with Python code
  – Embed Python code into Verilog files (.pyv)
    • Similar to how you can embed PHP into HTML files to generate HTML
  – Compiles the .pyv to .v

• Why
  – Can be tedious to write out multiple module instantiations and connect them
  – Something that is not synthesizable in Verilog could potentially be generated from Python in a synthesizable way
    • For loops for example (yes, Verilog has for loops, but they are not synthesizeable!!)
    • Could unroll the loop in Verilog, but this is tedious to do
    • Can generate the unrolled Verilog code with Python!
    • Makes it easier to make changes to long blocks of code that would otherwise be tedious to make
Python Preprocessor for Verilog

• Available on adroit for your use in labs or the final project:
  – `/u/mmckeown/ele475/dropbox/pyhp.tar.gz`
• Extract the tarball
  – `tar -zxvf pyhp.tar.gz`
• The main script for compiling `.pyv` to `.v` is `pyhp.py`
• Example is in `verilog_test/`
  – Enter “`make set_of_slices.v`” to compile the provided `.pyv` to `.v`
  – I will quickly walk through this example
Python Preprocessor for Verilog

Makefile

PYHP_BIN: =../pyhp.py

%.v: %.v.pyv

$(PYHP_BIN) $< > $@

Can modify Makefiles in labs in a similar way to compile .pyv files to .v files
module set_of_slices(a_out, clk, reset, b_in);

<%
# this would be better if it was broken out into an include file
NUMBER_SLICES = 5
%

//transforms from python to verilog defines
`define NUMBER_SLICES <%= NUMBER_SLICES %>

//outputs
output [`NUMBER_SLICES-1:0] a_out;
<%
for a in range(NUMBER_SLICES):
    s = "slice s%d(a_out[%d], clk, reset, b_in[%d]);" % (a, a, a)
    print s
%

endmodule
Python Preprocessor for Verilog

• `<%  %> tags are used to specify Python code
  – Can be inline or blocks of standalone Python code
  – Anything outside these tags is ignored and appears as is in the generated file

• The scope of variables extend across `<%  %> blocks

• Convenient to name file .v.pyv and structure your make file to simply strip off the .pyv suffix for the output file
  – So a convenient name for your file may be
    • `<module_name>.v.pyv` -> `<module_name>.v`
Python Preprocessor for Verilog

set_of_slices.v

module set_of_slices(a_out, clk, reset, b_in);

//transforms from python to verilog defines
`define NUMBER_SLICES 5

//outputs
output [`NUMBER_SLICES-1:0] a_out;

//clk and reset inputs
input clk;
input reset;

input [`NUMBER_SLICES-1:0] b_in;

slice s0(a_out[0], clk, reset, b_in[0]);
slice s1(a_out[1], clk, reset, b_in[1]);
slice s2(a_out[2], clk, reset, b_in[2]);
slice s3(a_out[3], clk, reset, b_in[3]);
slice s4(a_out[4], clk, reset, b_in[4]);

endmodule
PBS Job Resource Manager on Adroit
PBS Job Resource Manager

• What
  – A job resource manager
  – A job is a computational task, such as Verilog simulations
  – PBS provides job queuing and execution on a batch cluster

• Why?
  – When you log into adroit, you log into the head node
  – Anything you run executes on the head node
  – The head node should only be used to edit files, compile code, etc.
    • Not job execution
  – With PBS you can make use of other machines on the cluster to execute your job
PBS Job Resource Manger - Parameters

• Walltime – specifies the maximum real time (not CPU time)
  – Will kill your job if you exceed this
  – Too long a walltime will cause your job to be queued longer
• Nodes – specifies the number of nodes (servers)
• Processors – specifies the number of processors per node
• There are others, but these are the most common
• There are 3 different queues on adroit that you may be put in depending on your parameters
  – Short queue
    • 4 hour walltime limit, 16 cores max
  – Medium queue
    • 24 hour walltime limit
  – Long queue
    • 15 day walltime limit, 8 cores maximum
PBS Job Resource Manager – Interactive Jobs

• Interactive jobs
  – qsub -l <parameters>
    • -l is upper-case i
  – Parameters are specified with -l option (lower case L)
    • nodes=<number of nodes>:ppn=<number of processors per node>
    • walltime=<HH.MM.SS>
    • For others read “man qsub”
  – Example interactive command command:
    • qsub -l -l nodes=1:ppn=1 -l walltime=4:00:00
    • Requests an interactive job with 1 node and 1 processor on that node for 4 hours
PBS Job Resource Manager – Non-Interactive Jobs

• PBS Scripts (.cmd)
• Set of commands at top of script
  – #PBS -l walltime=<HH:MM:SS>
  – #PBS -l nodes=<number of nodes>:ppn=<number of processors per node>
  – There are others but these are most useful
• Example PBS script:
  #PBS -l walltime=4:00:00
  #PBS -l nodes=1:ppn=1

[Put commands to run when job is executed here]
Lab 1 – Practice Lab
Lab 1 – Practice Lab

- Iterative multiplier and divider
  - Algorithm flow charts and datapath diagrams are provided for you

- You will implement both the datapath and control for each
  - In separate modules (REQUIRED)

- You will then combine them into an iterative multiplier divider module

- Testing is required! You will need to explain your methodology

- Start early!!! (On all labs!!!
Questions?

(Also, come to my office hours!)