Agenda

• Purposes of HDL
• Verilog HDL
  • Paradigm
  • Differences from programming languages
  • Syntax
  • Types of variables
  • Types of assignments
  • Common styles
  • Module structure
  • Parametrization
  • Synthesizable VS not synthesizable constructs
• Design verification
• Common Errors
• Design guidelines
Purposes of HDL
Purposes of HDL

- Initially was created to simplify design simulation and verification

Verilog = Verification + Logic
VHDL = VHSIC Hardware Description Language
Purposes of HDL

• Initially was created to simplify design simulation and verification

  Verilog = Verification + Logic
  VHDL = VHSIC Hardware Description Language

• Increasing logic complexity drove added support for synthesis
Purposes of HDL

- Initially was created to simplify design simulation and verification

  \[
  \text{Verilog} = \text{Verification} + \text{Logic} \\
  \text{VHDL} = \text{VHSIC Hardware Description Language}
  \]

- Increasing logic complexity drove added support for synthesis

- Nowadays **SystemVerilog 2012** is an extension of **Verilog** designed with emphasis on verification
Verilog HDL
Poll #1

1. Who knows Verilog HDL?
Poll #1

1. Who knows Verilog HDL?

2. Who can write a synthesizable FIFO?
Poll #1

1. Who knows Verilog HDL?

2. Who can write a synthesizable FIFO?

3. Who can write a parameterized FIFO?
Verilog paradigm

- Design consists of hierarchical modules connected on the same level of hierarchy
- Each module is a set of interconnected wires, flip-flops, and combinational logic
- Only a subset of Verilog is synthesizable
- Non synthesizable part is used for testing
Differences from programming languages

**Verilog**

Variable declaration

wire [7:0] sum;

Procedural blocks

if (a == 8'b0) begin
  ...
end
else begin
  ...
end

**C/C++**

Variable declaration

int sum;

Procedural blocks

if (a == 0) {
  ...
}
else {
  ...
}
Differences from programming languages

**Verilog**

```verilog
wire [N-1:0] a;
generate
    for (i=0; i<N; i=i+1) begin
        assign a[i] = b & c;
    end
endgenerate
```

**C/C++**

```c
for (i=0; i<N; i++) {
    a[i] = b + c;
}
```

Describes Hardware

Describes Sequence of Instructions
Verilog Syntax
Syntax

• Logic values:
  ‘0’, ‘1’
  ‘x’ – unknown value (‘0’, ‘1’ or ‘z’)
  ‘z’ – high-impedance

• Literals  <width>’<base><number>
  Binary: 1’b0, 4’b1100, 8’b1001_1101
  Decimal: 8’d7
  Hexadecimal: 32’hBAC007

• Integer
  int i;    - 32 bit signed value;
**Syntax**

- **Wires**
  ```verilog
code
wire single_wire;
wire [7:0] one_dimensional_bus_1; or
wire [0:7] one_dimensional_bus_2;
wire [31:0] two_dimensional_bus [7:0];
code```

- **Slicing**
  ```verilog
code
wire [31:0] bus; - declare 4 byte bus
bus[7:0] - lowest byte of bus
code```

- **Concatenation**
  ```verilog
code
{bus[7:0], bus[15:8], bus[23:16], bus[31:24]}
code```

- **Replication**
  ```verilog
code
{8{4'b1010}}; - replicate binary value 4'b1010 8 times
code```
• **Register** – same declaration as for wires
  Slicing and concatenation also works for registers
  ```
  reg   empty;
  reg   [15:0] state;
  reg   [63:0] buffer  [2:0];
  {state[15], state[7], state[0]}
  ```

• Declaration of a register **does not always imply flip-flop** implementation (see following slides)
  • Depends on assignment type
## Syntax

### Operators

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Symbols</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bitwise</strong></td>
<td>~ &amp;</td>
<td>^</td>
</tr>
<tr>
<td><strong>Logical</strong></td>
<td>! &amp;&amp;</td>
<td></td>
</tr>
<tr>
<td><strong>Reduction</strong></td>
<td>&amp; ~&amp;</td>
<td>~</td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td>+ - * / ** %</td>
<td>4'b1110 - 1</td>
</tr>
<tr>
<td><strong>Relational</strong></td>
<td>&gt; &lt; &gt;= &lt;= == != === !===</td>
<td>4’d5 &lt; 4’d3</td>
</tr>
<tr>
<td><strong>Shift</strong></td>
<td>&gt;&gt; &lt;&lt; &gt;&gt;&gt; &lt;&lt;&lt;</td>
<td>4’0110 &lt;&lt; 1</td>
</tr>
<tr>
<td><strong>Concatenation</strong></td>
<td>{}</td>
<td>{2'b10, 2'b01}</td>
</tr>
<tr>
<td><strong>Replication</strong></td>
<td>{n{m}}</td>
<td>{8{1'b1}}</td>
</tr>
<tr>
<td><strong>Conditional</strong></td>
<td>? :</td>
<td>empty ? 1'b1 : 1'b0</td>
</tr>
</tbody>
</table>

**Note:** operators **are not listed** in descending precedence order. Always use parentheses if a code can cause ambiguity.
Syntax

• Basic Verilog statements: **always block**

```verilog
always @( <sensitivity list>) begin
   <statements>
end
```

```verilog
reg stage_1_val, stage_0_val;
always @(posedge clk) begin
   stage_1_val <= stage_0_val;
end
```

```verilog
reg ready;
wire empty;
reg [3:0] next_addr, curr_addr[3:0];
always @(*) begin
   ready = ~empty;
   next_addr = curr_addr + 1;
end
```

**Important notes:**

- all **always blocks** are executed in parallel
- LHS of an **always block** is always a register
- Can describe either a flip-flop or combinational logic
Syntax

- **Blocking VS non-blocking** assignments
  - = - blocking
  - <= - non blocking, used only in always blocks

### Blocking
Statements inside always block are executed **sequentially**

```vhdl
always @(*) begin
    b = a;
    c = b;
    d = b;
end
```

### Non Blocking
Statements inside always block are executed in **parallel**. Value assigned to LHS are taken from “previous” values of RHS

```vhdl
always @(posedge clk) begin
    b <= a;
    c <= b;
    d <= b;
end
```
Syntax

**Blocking**

```plaintext
always @(*) begin
  out = a & b;
end
```

**Non Blocking**

```plaintext
always @(posedge clk) begin
  out <= a & b;
end
```
Syntax

• Assignment statement (block)

```verilog
wire any_val;
wire in_val_1, in_val_2, in_val_3;
assign any_val = in_val_1 | in_val_2 | in_val_3;
```

Important notes:

- all assign statements are executed in parallel
- LHS of assign block is always a wire
- Describes only combinational logic
• Initial block

```verilog
initial begin
    clk = 0;
    rst_n = 0;
    init = 0;
    forever begin
        #5 clk = ~clk;
    end
end
```

Important notes:

- Starts execution at time 0
- Executed only once
- Non synthesizable, used for simulation and verification
Syntax

- Conditional statements
  
  \[
  \text{if} \ <\text{condition}> \ \text{begin} \\
  \quad <\text{statement1}> \\
  \text{end} \\
  \vspace{1em}
  \text{[else if} \ <\text{condition}> \ \text{begin} \\
  \quad <\text{statement2}> \\
  \text{end]} \\
  \vspace{1em}
  \text{[Else begin} \\
  \quad <\text{statement3}> \\
  \text{end]} \\
  \]

always @(posedge clk) begin
  if (~rst_n)
    data_out <= {64{1'b0}};
  else
    data_out <= data_stage_4;
end

Important notes:

- Used in always, initial blocks
- A latch will be generated instead of a flip-flop if there is no else statement
Syntax

• Case statement
  
  ```
  case (addr):
    0: out = a;
    1: out = b;
    2: out = c;
    default: begin end
  endcase
  ```

• ```casex```: treats `x` as don’t care
• ```casez```: treats `z` as don’t care

Important notes:

- Used in `always`, `initial` blocks
- A latch will be generated instead of a flip-flop if `case` is not full and there is no default statement

Used in simulation
Syntax

- Conditional (ternary) operator

```verilog
code
wire [7:0] data_out = r_val & w_val ? data_in :
    r_val & ~empty ? buffer[id] :
    {8{1'b0}} ;

always @(posedge clk) begin
    if (~rst_n) begin
        cmd_out <= {3{1'b0}};
    end
    else begin
        cmd_out <= val_0_in ? cmd0 :
            val_1_in ? cmd1 : cmd_default;
    end
end
```

- Good thing: always has default case
**Common styles**

**Style 1**

**Combinational logic**

```verilog
title Common styles

wire empty;
assign empty = (cnt==0) & ~wr_val;

reg [ST_W-1:0] state;
always @(posedge clk) begin
  if (~rst_n)
    state <= {ST_W{1'b0}};
  else
    state <= val_1_in ? ACCEPT_ST :
                val_2_in ? WAIT_ST :
                        state;
end
```

**Sequential logic**

```verilog
title Common styles

reg state, state_next;
always @(*) begin
  if (val_1_in)
    state_next = ACCEPT_ST;
  else if (val_2_in)
    state_next = WAIT_ST;
  else
    state_next = state;
end
always @(posedge clk) begin
  if (~rst_n)
    state <= {ST_W{1'b0}};
  else
    state <= state_next;
end
```

**Style 2**

**Combinational logic**

```verilog
reg empty;
always @(*) begin
  empty = (cnt==0) & ~wr_val;
end
```

**Sequential logic**

```verilog
reg state, state_next;
always @(*) begin
  if (val_1_in)
    state_next = ACCEPT_ST;
  else if (val_2_in)
    state_next = WAIT_ST;
  else
    state_next = state;
end
always @(posedge clk) begin
  if (~rst_n)
    state <= {ST_W{1'b0}};
  else
    state <= state_next;
end
```
## Syntax

### Module structure

```verilog
module <module name> ( 
  input          clk, 
  input          rst_n, 

  input [31:0]   data_in,  
  input          w_val, 

  output reg     data_out,  
  output reg     r_val 
);

// Register declaration
reg [3:0]      position;

// Wire declaration
wire [3:0]     position_next;

// Sequential logic

// Combinational logic

endmodule
```

### Module instantiation

```verilog
<module name> <instance name> ( 
  .clk         (rst_n   ),
  .rst_n       (rst_n   ),

  .data_in     (data_in ),
  .w_val       (w_val    ),

  .data_out    (data_out),
  .r_val       (r_val    )
);
```
Syntax

• Parametrization
  Verilog has preprocessor – can use defines

```
`include <defines.h>

module example(
  input clk,
  input rst_n,

  input [`DATA_IN_W-1:0] data_in,
  output reg [`DATA_OUT_W-1:0] data_out
);

always @(posedge clk) begin
  if (~rst_n)
    data_out <= `{`DATA_OUT_W{1'b0}};
  else
    data_out <= data_in[`DATA_OUT_W-1:0];
end

endmodule
```
• Parametrization

Parameters

```verilog
module example #(parameter WIDTH = 16)
  (input clk,
   input rst_n,
   input [`WIDTH-1:0] data_in,
   output reg [`WIDTH-1:0] data_out);

  ...

endmodule
```

• Overwriting parameters

```verilog
example exempl_inst #
  .WIDTH   (32)
)
)
(
  .clk       (clk    ),
  .rst_n     (rst_n  ),
  .data_in   (data_in ),
  .data_out  (data_out )
);
```
Verilog

• Parametrization
  Local parameters

`define WIDTH 16

module example (  
  input       clk,  
  input   rst_n,  
  input      [`WIDTH-1:0] cmd
);
localparam INIT_ST    = 0;
localparam WAIT_DATA_ST = 1;
localparam WAIT_CMD_ST = 2;

parameter BUF_SIZE = 64;

...

endmodule

Notes:

➤ Parameters can be overwritten from outside, localparms cannot
Non-synthesizable Verilog & Other topics
Verilog

• **Nonsynthesizable** part of Verilog is used to:
  • Simulate delays
  • Monitor signal values
  • Model external signals
  • Stop/continue simulation

• Simulation time precision
  `timescale 1ns/1ps
module tb_top();
...
endmodule

• Model external signals, delays
  initial begin
    clk = 0;
    rst = 0;
    #200
    rst = 1;
  end
• Generate system clock
  initial begin
    forever
      clk = #10 ~clk;
  end

• Monitor signal values
  always @(posedge clk) begin
    if (val_in)
      $display("Data written: \%x", data_in, $time);
  end

• Dump waveforms*
  initial begin
    $dumpfile("waves.vcd");
    $dumpvars(0, fpga_top.cmp_top.mc_top.pkt_trans_dp_wide);
    $dumpvars(0, fpga_top.cmp_top.mem_io_splitter);
  end
  *for loop may be required to dump multidimensional buses

• Finish simulation
  always @(posedge clk)
    if ($time >= MAX_SIM_TIME)
      $finish;
Verilog

... and many others:

- Tasks
- Functions
- While, for loops
- $\text{	extdollar reset, $\text{	extdollar stop}$}
- $\text{	extdollar random}$
- $\text{	extdollar fopen, $\text{	extdollar fdisplay, $\text{	extdollar fwrite}$}$}
Python Preprocessor for Verilog

- PyHP preprocessor can run Python code to generate Verilog code
  - Simplifies verbose Verilog code
  - Can use Python functions and/or loops to dynamically generate Verilog
- Useful for
  - Connecting large modules together
  - Filling out a complex control signal table
  - Instantiating code that is difficult to create using a generate statement
Available on adroit for your use in labs or the final project:
  /home/ee475/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
Job Scheduling on Adroit

• Logging into Adroit logs into the head node
• The head node should be used for small tasks like file editing, compilation, etc.
  • Simulations should not run on the head node

• SLURM – Simple Linux Utility for Resource Management
  • Manages hardware resources
  • Schedules when jobs run on which machines

• Why use SLURM?
  • Takes advantage of the entire cluster
  • Don’t want to unfairly take compute resources away from others
Job Scheduling on Adroit

- `sinfo` – displays cluster resources
- `Squeue` – displays job queue

```
[pjj@adroit3 ~]$ sinfo
PARTITION AVAIL TIMELIMIT NODES STATE NODELIST
all* up 15-00:00:0 3 mix adroit-[01,05-06]
all* up 15-00:00:0 5 alloc adroit-[02-04,07-08]
```

```
[pjj@adroit3 ~]$ squeue
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
380774_[9] all ffs-ar9 hajakbar PD 0:00 1 (Resources)
380775_[10] all ffs-ar10 hajakbar PD 0:00 1 (Priority)
380370_[4] all ffs-ar4 hajakbar PD 0:00 1 (Priority)
380778_[5] all ffs-ar5 hajakbar PD 0:00 1 (Priority)
380779_[6] all ffs-ar6 hajakbar PD 0:00 1 (Priority)
380780_[7] all ffs-ar7 hajakbar PD 0:00 1 (Priority)
380752_[11] all ffs-ar11 hajakbar PD 0:00 1 (Dependency)
380753_[12] all ffs-ar12 hajakbar PD 0:00 1 (Dependency)
380760_[11] all ffs-ar11 hajakbar PD 0:00 1 (Dependency)
```

Job Scheduling on Adroit

• A submission script is used to define what commands are needed in order to run the job
  • `srun` indicates a command to be run
• Submission scripts also contain information about the job itself
  • Use SBATCH parameters
    
    ```bash
    #!/bin/bash
    
    #SBATCH --job-name=pjjtest    // Job name
    #SBATCH --output=result.txt   // Log file for console output
    #SBATCH --ntasks=1            // Number of tasks (>1 when parallelizing tasks)
    #SBATCH --time=10:00          // Allocated time to complete job
    #SBATCH --mem-per-cpu=100     // Memory allocation per cpu(MB)
    
    srun make clean
    srun make
    srun make check
    srun sleep 60
    
    ```
Job Scheduling on Adroit

- Submit jobs using `sbatch` command
- Information about submitted jobs can be found using `sstat` command

```
[pjj@adroit3 build]$ sbatch submit.sh
Submitted batch job 407990
[pjj@adroit3 build]$ sstat 407990
```

| JobID  | MaxVMSize | MaxVMSizeNode | MaxVMSizeTask | AveVMSize | MaxRSS | MaxRSSNode | MaxRSSTask | AveRSS | MaxPages | MaxPagesNode | MaxPagesTask | AvePages | MinCPU | MinCPUNode | MinCPUTask | AveCPU | NTasks | AveCPUFreq | ReqCPUFreq | ReqCPUFreqGov | ConsumedEnergy | MaxDiskRead | MaxDiskReadNode | MaxDiskReadTask | AveDiskRead | MaxDiskWrite | MaxDiskWriteNode | MaxDiskWriteTask | AveDiskWrite |
|--------|-----------|----------------|--------------|-----------|--------|------------|------------|--------|----------|-------------|-------------|-----------|--------|------------|-------------|--------|--------|------------|------------|--------------|---------------|-------------|----------------|----------------|-------------|--------------|----------------|---------------|-------------|--------------|
| 407990.4 | 214764K   | adroit-06      |              | 0        | 4080K  | 1288K      | adroit-06  | 0     | 548K     | 0           | 0          | 0        | 00:00.000 | adroit-06  | 0        | 00:00.000 | 1          | 2.50M | Unknown |               |            |              |               |             |              |               |            |              |

Design verification

Make sure that the module is doing what it supposed to do

• **For labs:**
  • Check all functions, performed by a module
  • Check corner case
  • Explain why those test cases are corner cases

• **For real-world designs:**
  • Coverages: by lines, by branches, by FSM states
  • Formal verification
Common Errors

• No clock signal
• Wrong reset level
• If-else, case statements without default case
• No initial values for registers (which do require it)
• Blocking assignment (=) in @(posedge clk) blocks
• Describing a combinational logic instead of flip-flop
• Width mismatch
• Counter overflow/undeflow (do no assume all counter to be power of 2, especially in parametrized modules)
• Deadlock on val/rdy interface (circular dependence)
Design guidelines

• Use version control

• Think in terms of hardware

• Write unit tests for every module

• Use incremental development

• **START EARLY!!** Do not postpone labs till the last day 😊
Have more questions?

- Refer to recommended Verilog book

- Come to office hours!
  - Tuesdays/Thursdays 12:30-1:30pm
  - Message me if you are coming and need access to F210
  - Also by appointment