Logic Synthesis

Design Challenges

Correct functionality requirements
High-Level Design Issues

- You may think design is a straightforward logical process
  - Start with the idea of what you need to build
  - And then build it
- Real design is not like that
  - Think you have an idea of what you need to build
  - Through the design process you figure out what you really want to build
  - Need to validate basic idea early in the process
- What you build depends on the implementation capabilities and constraints
  - Implementation issues will change the specification

Need a language that helps with the real (interactive) design process

Hardware Description Languages

- Need a description level up from logic gates
- Work at the level of functional blocks, not logic gates
  - Complexity of the functional blocks is up to the designer
  - A functional unit could be an ALU, or could be a microprocessor
- The description consists of functional blocks and their interconnections
  - Describe functional block (not predefined)
  - Support hierarchical description (function block nesting)
- To make sure the specification is correct, make it “executable”
  - Run the functional specification and check what it does
Hardware Description Languages

- A hardware description language (HDL) translates the specification of a hardware device into a software medium; This software medium can be
  - Verified via simulation
  - Translated into an optimized, technology-specific, gate-level implementation

- Register Transfer Level (RTL) synthesis explicitly defines register boundaries and the combinational logic between them:
  - Data flow, control flow, and machine states are explicitly defined by the coding style that is used

Hardware Description Languages (Examples)

- There are many different systems for modeling and simulating hardware
  - Verilog
  - VHDL
  - L-language, M-language (Mentor)
  - DECSIM (DEC)
  - Aida (IBM / HaL)
  - and many others

- The two most standard languages: Verilog & VHDL
  - For this class we will be using Verilog
  - Given to UCLA for classes
    * Have both a simulator and synthesis tools that work with Verilog
**Verilog from 20,000 Feet**

- Verilog descriptions look like programs:
  - C / Pascal
  - Verilog
  - Procedures/Functions
  - Procedure parameters
  - Variables
  - Modules
  - Ports
  - Wires / Regs

- **Block structure** is a key principle
  - Use hierarchy/modularity to manage complexity

- But they aren’t ‘normal’ programs
  - Module evaluation is concurrent
  - Model is really that of communicating blocks

**Verilog (or any HDL) View of the World**

- A design consists of a set of communicating **modules**

- There are graphic-input tools for Verilog
  - Come to EE219A
  - Matlab/Simulink → RTL

- We will use the text method
  - EEM216A

- Basic philosophy
  - Label the wires, and pass them between modules as you would parameters in function calls
  - Wires are I/O nets for modules
Example Verilog

```
ModuleName InstanceName (wires);
module system;
    wire [7:0] bus_v1, const_s1;
    wire [2:0] regSpec_s1, regSpecA_s1, regSpecB_s1;
    wire [1:0] opcode_s1;
    wire Phi1, Phi2, writeReg_s1, ReadReg_s1, nextVector_s1
clkgen clkgen(Phi1, Phi2);
datapath datapath(Phi1, Phi2, regSpec_s1, bus_v1, writeReg_s1, readReg_s1);
controller controller1(Phi1, Phi2, regSpec_s1, bus_v1, const_s1, writeReg_s1, readReg_s1, opcode_s1, regSpecA_s1, regSpecB_s1, nextVector_s1);
patternsource patternsource(Phi1, Phi2, nextVector_s1, opcode_s1, regSpecA_s1, regSpecB_s1, const_s1);
```

In this example the instance name and the module name are the same, except for controller1.

Ways to Describe A Function

- **Structural**
  - Consists only of module calls

- **Declarative**
  - Concurrently executed combinational logic

- **Procedural**
  - Sequentially executed program
    - A state machine (with storage)
    - Or combinational logic

- **Functional**
  - Function/task calls
**Structural Description**

- **ModuleName InstanceName (wires);**
  - Maps a physical structure into Verilog
    - Example is one shown earlier (repeated here)
  - Possible hierarchically
    - List of functions
    - List of sub-functions
    - List of gates
    - List of transistors
  - Typically don’t need to go below a gate level list
    - Standard cells

```
module system;
  wire [7:0] bus_v1, const_s1;
  wire [2:0] regSpec_s1, regSpecA_s1, regSpecB_s1;
  wire [1:0] opcode_s1;
  wire Phi1, Phi2, writeReg_s1, ReadReg_s1, nextVector_s1;
  clkgen clkgen(Phi1, Phi2);
  datapath datapath(Phi1, Phi2, regSpec_s1, bus_v1, writeReg_s1, ReadReg_s1);
  controller controller1(Phi1, Phi2, regSpec_s1, bus_v1, const_s1, writeReg_s1, ReadReg_s1, opcode_s1, regSpecA_s1, regSpecB_s1, nextVector_s1);
  patternsource patternsource(Phi1, Phi2, nextVector_s1, opcode_s1, regSpecA_s1, regSpecB_s1, const_s1);
```

Compose a module out of module calls. Specify components and wiring.

---

**Declarative Statements**

- Provides the **logical relations** between inputs and outputs
  - **Assign outputs to be some function of the inputs** (continuously)
    - Key word is assign
  - Models a piece of combinational logic
  - Uses a C-like expression syntax
  - Denoted by keyword assign

**Examples** (all execute in parallel):

```
assign nor = ~(b | c);
assign a = x & y, o = x | y;
assign sum[4:0] = a[3:0] + b[3:0];
assign out = (Sel) ? in1: in2;  //conditional
```

Outputs are wires, and can be a single bit or multiple bits.
- It is good practice to **declare all variables** even though Verilog allows undeclared single bit wires
Declarative Order of Execution

- Don’t assume any particular order.
  - Each statement occurs concurrently.
    assign x = aaa;
    assign aaa = bbb;
  - Unlike C, the order of the above statements does not matter

- Internally, however, declarative statements are still executed in a particular order
  - Verilog has an internal event linked list
  - But, there is no guarantee of that order
    assign out = aaa;
    assign out = bbb;
  - This yields a warning and is not allowed

Procedural Statements

- Procedural statements = sequential order

- Keyword always
  - provides functionality of a tiny program that executes sequentially

- Inside an always block, standard control flow statements:
  
  if (<conditional>) then <statements> else <statements>;
  case (<var>) <value>: <statements>;
  ... default: <statements>

- Note: Case statements are actually prioritized
  - The second case entry can’t happen unless the first does not match.
  - May not be what the actual hardware implies – especially when cases are mutually exclusive.
  - Need additional directives (parallel-case) to indicate this. More later.

- Statements can be compound
  - Use begin and end to form blocks
Procedural Statements (Cont.)

- **Example**

```plaintext
always @ (Activation List...stuff we still need to talk about)
begin
    // more than 1 statement allowed inside here
    if (x==y) then
        out= in1
    else
        out = in2;
end
```

always Block Issues

Two issues with always blocks...

- **Issue #1:** Unset outputs
  - Are all outputs given a value with an explicit assignment statement at the end of the block?
  - If not, then it is “unset”
  - If the output is always set, then the always block is no different from a combinational logic

- **Issue #2:** Activation list
  - Determines when to execute the always block.
Unset Outputs

- Occur when an output is not set on all the paths in the code

**Example:**

```verilog
always @ (Activation List...stuff we need to talk about)
begin
  // more than 1 statement allowed inside here
  if (x==y) then
    out= in;
  // no else statement. So, if x!= y then out is unset.
end
```

- In Verilog, this creates storage
  - The value of the output remains the previous value
  - In synthesized result, it appears as an explicit FF or latch

- Is this storage what we want?
  - Be careful to not build storage elements when you don’t intend to

- The outputs of always blocks might act as storage elements
  - Left-hand sides of expressions in `always` blocks must be declared as registers (regs)
    - That does not mean the synthesized result contains registers
    - If output is set on all paths, there is no storage

Unset Outputs

```verilog
always begin
  Data_Out = In_B;
  if (Enable)
    begin
      Data_Out = In_A;
    end
end
```

Did he want latch or combinational logic?
Unset Outputs

```verilog
always @(Enable or In_A or In_B)
begin
    if (Enable)
        begin
            Out_1 = In_A;
        end
    else
        begin
            Out_2 = In_B;
        end
end
```

Intentionally Creating Storage in Verilog

- To make a simple latch in Verilog is easy
  - Just make the output of an `always` block not get set when you want to hold its value

- Example:
  ```verilog
  reg myout; // a latch
  always @ (stuff we still need to talk about)
  begin
      if (Enable) then
          myout = in;
  end
  ```
  - When `Enable` is high, the output `myout` is updated
  - When `Enable` is low, `myout` will hold its last value.
  - This is like the simple pass transistor latch in Lecture 10

- In this example, `myout` would need to be declared a register, because it is the LHS of an expression in an `always` block
Is this an infinite loop?

Example 1

```verilog
assign nor = ~(b | c);
```

Example 2

```verilog
always
nor = ~(b | c);
```

Activation List

- The last tricky part about the `always` block...
- Activation List
  - Tells the simulator when to run this block
  - Allows the user to specify when to run the block and makes the simulator more efficient
    - If not sensitized to every input, you get a storage element
  - But also enables subtle errors to enter into the design

Two forms of activation list in Verilog

- `@(signalName1 or signalName2 or ...)`
  - Evaluate this block when any of the named signals change (either positive or negative change)
- `@(posedge signalName);` or `@(negedge signalName);`
  - Makes an edge triggered flop. Evaluates only on one edge of a signal
  - Can have `@(posedge signal1, posedge signal2)`
    - Implied OR (not AND) because edges are singular events
    - Not used in this class because difficult to map to an actual gate
Activation Lists

- **Example:**
  ```verilog
  always @ (Enable or In) // latch
  if (Enable) then
    out = In;
  always @ (x or y or in1 or in2) // combinational logic
  begin
    if (x == y) then
      out = in1
    else
      out = in2;
  end // same as out = (x==y) ? in1 : in2;
  ```

- To represent Combinational Logic
  - The activation lists must contain **everything** on the RHS of the expressions (and both side of conditionals)
  - Why?

- Beware, if an always block has **no** activation list (or # delay statements), then the simulator goes into an infinite loop

- **always@* syntax**

### Activation Errors: Examples

<table>
<thead>
<tr>
<th>always @(phi)</th>
<th>always @(phi)</th>
<th>always @(phi or in)</th>
</tr>
</thead>
<tbody>
<tr>
<td>outA = in;</td>
<td>if(phi) outB = in;</td>
<td>if(phi) outC = in;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>phi</th>
<th>in</th>
<th>outA</th>
<th>outB</th>
<th>outC</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="waveform1.png" alt="Waveform" /></td>
<td><img src="waveform2.png" alt="Waveform" /></td>
<td><img src="waveform3.png" alt="Waveform" /></td>
<td><img src="waveform4.png" alt="Waveform" /></td>
<td><img src="waveform5.png" alt="Waveform" /></td>
</tr>
</tbody>
</table>

What would synthesis results look like?
Procedural Order of Execution

- Be careful of the sequential nature! (C-like behavior)
- **Case 1**
  always @(posedge clock) begin
    q2=q1;
    q1=q0;
  end
- **Case 2**
  always @(posedge clock) begin
    q1=q0;
    q2=q1;
  end
- **Case 3** – Which one is this case more similar to?
  always @(posedge clock) begin
    q1=q0;
  end
  always @(posedge clock) begin
    q2=q1;
  end

Non-Blocked Assignment

- A newer feature of Verilog helps by eliminating the order of evaluation
  - Instead of "="; known as a blocking assignment
    * Blocks future action until RHS is updated
  - Use "="; known as non-blocking assignment
    * All LHS are changed first before the RHS is updated
  ```verilog
  always @(posedge clock)
  begin
    a[0] <= inp;
    a[1] <= a[0];
    a[2] <= a[1];
    a[3] <= a[2];
  end
  ```
- The above is equivalent to `a[3:0]={a[2:0],inp};`
- If we had used "=" instead of "="; known as non-blocking assignment, then `a = 4{inp};`
Parallel Case Definition

- A case statement is called **parallel case** if its case items do not overlap:
  - If synthesis determines that the case statement is parallel case then a multiplexer or equivalent logic will be synthesized
  - Otherwise priority-checking logic will be synthesized

Parallel Case Example

```vhdl
case (Sel) 2'b00 : D_Out = A; 2'b01 : D_Out = B; 2'b10 : D_Out = C; 2'b11 : D_Out = D; endcase
```

No priority logic needed
**Initial Block**

- This is another type of *procedural block*
  - Does not need an activation list
  - It is *run just once*, when the simulation starts

- Used to do extra stuff at the very start of simulation
  - Initialize simulation environment
  - Initialize design
    - This is usually only used in the first pass of writing a design
    - Beware, real hardware does not have initial blocks
  - Allows testing of a design (outside of the design module)

- Best to use *initial* blocks only for non-hardware statements (like `$display`)

---

**Summary of Verilog Variables**

- There are two types of “physical” variables in Verilog:
  - *Wires* (all outputs of `assign` statements must be wires)
  - *Regs* (all outputs of `always` blocks must be regs)

- Both variables can be used as inputs anywhere
  - Can use regs or wires as inputs (RHS) to `assign` statements
    ```verilog
    assign bus = LatchOutput + ImmediateValue
    ```
    - `bus` must be a wire, but `LatchOutput` can be a reg
  - Can use regs or wires as inputs (RHS) in `always` blocks
    ```verilog
    always @ (in or clk)
    if (clk) out = in
    ```
    - `in` can be a wire, `out` must be a reg
Summary of Verilog Variables (Cont.)

- **Module outputs** can be either *regs* or *wires*
  
  ```verilog
define div_ctrl(ctl1, ctl2, dp1, clock, reset, start);
  output  ctl1, ctl2;
  input   dp1, clock, reset, start;
  ```

- **Integer and real do not map into hardware**
  
  - Useful for initial functional description but not for implementation

Delays in Verilog

- **Verilog simulated time is in “units” or “ticks”**
  
  - Simulated time is unrelated to the wall-clock to run the simulator
  - Simulated time models the time in the modeled machine
    
    * When the computer completes with all the “events” that occur at the current simulated time
    * The computer increases time until another signal is scheduled to change values

- **User must specify delay values explicitly to Verilog**
  
  - `# delayAmount`
    
    * When the simulator sees this symbol, it stops “evaluating”, and pause delayAmount of simulated time (# of ticks).
    * Delays are often used to model the delay in functional units
    * Can be tricky to use properly
  
  - We will design our logic to have zero (or unit) delay
    
    * The standard cell library we use can annotate delay information
**RTL Subset**

- HDLs were designed for simulation
- HDLs were designed for simulation
- Subset of the language is supported for Synthesis
- Unsupported Verilog language constructs:
  - Delays
  - “initial” blocks
  - $display
  - Tool Specific
- Organize synthesis and simulation specific code into separate files
  - RTL
  - Test bench

**Verilog Summary**

- An HDL provides a means for the user to specify a design at a higher level than just gates
  - This lecture addresses mostly form and not content
    - How to represent combinational logic and state machines
    - We can now use this tool to specify any machine with state
  - A good question to ask is
    - “What should my code look like?”
    - “Are there certain styles of hardware that are easier to understand / build / test?”
    - This gets back to the question of abstractions, and is really asking whether there are some hardware abstractions that work well
  - The answer is briefly introduced in the examples above
    - Partitioning of the problem into
      - Finite State Machines
      - Data flows
Now What?

**Synthesis = Translation + Logic Optimization + Mapping**

```
residue = 16'h0000;
if (high_bits == 2'b10)
    residue = state_table[index];
else
    state_table[index] = 16'h0000;
```

HDL Source

Translate (read)

Generic Boolean (GTECH)

Optimize + Map (compile)

Target Technology

Courtesy: Synopsys

---

Synthesis Design Flow

Spec ➔ Select Architecture ➔ Code RTL ➔ RTL Code Check ➔ RTL Verification ➔ Formal Verification ➔ ATPG ➔ Gate-level Verification ➔ Floorplan

Gate-level netlist/db ➔ Static Timing Analysis ➔ Constraints

CWLM ➔ Lib ➔ DW ➔ Synthesis (Logic Synthesis, Test (Scan/JTAG), Power Reduction, Datapath Synthesis, Testbench)

Testbench ➔ RTL Verification ➔ Formal Verification

GDSII ➔ Placement Info ➔ Physical Design

Courtesy: Synopsys
**The Importance of HDL Coding Styles**

- Different coding styles that are functionally equivalent may synthesize into hardware that have different timing and area characteristics.
- Synthesis engineers cannot rely solely on the synthesis tool to "fix" a poorly coded design.
- Alternative coding styles should be explored, so that the synthesis algorithms will have the best possible starting point.

*Synopsys*
Synthesis Is Constraint-Driven

You set the goals (through constraints).

Synthesis Tool optimizes the design to meet your goals.

Setup Example: Adder.tcl (1/2)

```tcl
set_attribute library /w/apps/apps.16/cadence/gscLib090_v2.9/timing/typical.lib
set_attribute hdl_language v2001
read_hdl adder.v
read_hdl SynLib.v
elaborate adder

dc::current_design adder
dc::set_time_unit -picoseconds
dc::set_load_unit -femtofarads

define_clock -name clk -period 1000 -design /designs/adder
{/designs/adder/ports_in/clk}
dc::set_input_delay 20 -clock clk [all_inputs]
dc::set_output_delay 100 -clock clk [all_outputs]
set_attribute external_driver [find [find / -libcell DFFX1] -libpin D]
{/designs/adder/ports_in/*}
set_attribute external_pin_cap 26.5488 {/designs/adder/ports_out/*}
set_attribute lp_power_unit mW /
set_attribute max_dynamic_power 0.5 /designs/adder
```

Define lib

Read HDL

Elaborate

Set timing

Set env.

Goal
Setup Example: Adder.tcl (2/2)

```bash
synthesize -to_mapped -effort high
report area > adder_area.rpt
report power > adder_power.rpt
report timing > adder_timing.rpt
report clocks > adder_clocks.rpt
write_encounter design adder -basename adder -lef [format "%s %s %s"
   /w/ee.00/dejan/rashmi/ee216a_test/gpdk090_9lm.lef
   /w/apps/apps.16/cadence/gsclib090_v2.9/lef/gsclib090_tech.lef
   /w/apps/apps.16/cadence/gsclib090_v2.9/lef/gsclib090_macro.lef]
```

What’s “under the hood”?

Synthesis Summary

- Synthesis tools are like compilers
  - Allow the user to work at a higher level
  - Show you what the details look like

- Use tools to understand the parts that need extra work
  - Optimize the parts that don’t meet the constraints
  - Don’t improve what is not broken

- Tools leverage your creativity
  - Not a substitute for thinking
  - Need to compete with others using the same tools!