EECS 470 Lab 2

Synopsys Build System

Department of Electrical Engineering and Computer Science
College of Engineering
University of Michigan

September 4th, 2025

Overview

Administrivia

Homeworks partying-face

- Homework 1 is due Wednesday, 9th September 11:59pm on Gradescope
- Homework 2 released shortly, is due Wednesday, 17th September 11:59pm on Gradescope

Projects cowboy-hat-face

- Project 1 is due Monday, 8th September 11:59 PM (turn in via submission script)
- Project 2 released shortly, is due Monday, 15th September 11:59 PM (turn in via submission script)
- We will monitor Piazza up to the project deadline. Please create a private post if you are having trouble submitting.

Organized Design

Why Is Organization Important?

- More readable, and therefore easier to update
- Less duplicated code, and thus fewer typos

How Do we Organize?

- Split design between files
  - Implementations: `*.sv`
  - Headers: `*.svh`
  - Packages: `*.svp`
- Succint notation for repeated instantiations
  - Array connections (for multiple identical modules)
  - For loops (see last week)
  - Generate blocks (coming later)

Verilog Headers

Purpose

- Define general values and objects for shared use
  - Macros, structs, enums, ...
- De-clutter implementation files

Inclusion

- Syntax: `` `include <FILE_NAME> ``
- Pastes the contents of `<FILE_NAME>` wherever the include appears

Include Guards

- Inside the header...

```
`ifndef __FILE_NAME_H__
`define __FILE_NAME_H__

:

`endif
```

Verilog Inclusion by Example

oompa_loompa.svh

```
`ifndef __HEADER_SV__
`define __HEADER_SV__

`define CHOCOLATE_BITS      10
`define NUM_OOMPA_LOOMPAS   16334

typedef struct packed {
    logic                                       is_working;
    logic                                 [1:0] intelligence;
    logic [$clog2(`NUM_OOMPA_LOOMPAS)-1:0] id;
} oompa_loompa;

typedef enum {
    chip    = (`CHOCOLATE_BITS)'h1;
    square  = (`CHOCOLATE_BITS)'h2;
    bar     = (`CHOCOLATE_BITS)'h3;
    brehob  = (`CHOCOLATE_BITS)'h4;
} chocolate_type_t;

`endif
```

Verilog Inclusion by Example

fire_oompa.sv

```
`include "oompa_loompa.svh"

module fire_oompa(
    output logic [$clog2(`NUM_OOMPA_LOOMPAS)-1:0] id;
);
    oompa_loompa          [`NUM_OOMPA_LOOMPAS-1:0] workers;
    logic                 [`NUM_OOMPA_LOOMPAS-1:0] can_fire;

    always_comb begin
        for (int i = 0; i < `NUM_OOMPA_LOOMPAS; i++) begin
            can_fire[i] = !workers[i].is_working &&
                           workers[i].intelligence == '0;
        end
    end

    psel_gen #(.WIDTH(`NUM_OOMPA_LOOMPAS),.REQS(1))
    psel_gen  (.req(can_fire) .gnt(id));

endmodule
```

Array Connections

Motivation

- You may have a module duplicated multiple times and wired in a regular pattern
- Just like we have arrays of values, what if we had arrays of modules?

Example

- Let's try to build a ripple-carry adder
- Assume we have a module definition:
  - `one_bit_addr(a, b, cin, sum, cout);`
- All ports are 1 bit, the first three inputs, last two outputs
- How do we build an eight bit adder?

The Error Prone Way

```
module eight_bit_addr(
    input en, cin,
    input [7:0] a, b,
    output [7:0] sum,
    output cout);

    wire [6:0] carries;

    one_bit_addr a0(en, a[0], b[0], cin, sum[0], carries[0]);
    one_bit_addr a1(en, a[1], b[1], carries[0], sum[1], carries[1]);
    one_bit_addr a2(en, a[2], b[2], carries[1], sum[2], carries[2]);
    one_bit_addr a3(en, a[3], b[3], carries[2], sum[3], carries[3]);
    one_bit_addr a4(en, a[4], b[4], carries[3], sum[4], carries[4]);
    one_bit_addr a5(en, a[5], b[5], carries[4], sum[5], carries[5]);
    one_bit_addr a6(en, a[6], b[6], carries[5], sum[6], carries[6]);
    one_bit_addr a7(en, a[7], b[7], carries[6], sum[7], cout);
endmodule
```

The Error Prone Way

- Lots of duplicated code
- Really easy to make mistake
- Now try building a 64-bit adder..., 256?
- There is a one line substitute

The Better Way

```
module eight_bit_addr(
    input en, cin,
    input [7:0] a, b,
    output [7:0] sum,
    output cout);

    wire [6:0] carries;

    one_bit_addr addr [7:0] (
        .en(en), .a(a), .b(b), .cin({carries,cin}),
        .sum(sum), .cout({cout,carries})
    );
endmodule
```

All of the ports in the `one_bit_addr` module are 1 bit wide. All of the busses we pass are 8 bits wide, so each instantiation of the module will get one, except `en` which is only 1 bit wide and thus copied to every module.

The (Even) Better Way

```
module parametrizable_bit_addr
  #(parameter WIDTH = 8)
   (input en, cin,
    input  [WIDTH-1:0] a, b,
    output [WIDTH-1:0] sum,
    output cout);

    wire [WIDTH-2:0] carries;

    one_bit_addr addr [WIDTH-1:0] (
        .en(en), .a(a),.b(b),.cin({carries,cin}),
        .sum(sum),.cout({cout,carries})
    );
endmodule
```

Array Connections: Pitfalls and Errors

What happens if a wire isn't 1 or N bits wide?

```
wire foo;
wire [3:0] bar;
wire [2:0] baz;

simple s1 [3:0](.a(foo), .b(bar), .c(baz));
```

```
test.v:14:  error:  Port expression width 3 does not match expected width 4 or 1
```

But still easy to accidentally promote a wire to a bus

Tools

What are our Tools

- In this class we use many tools to go from design to chip
- **Make**: Simplify commands and monitor dependencies
- **VCS**: Verilog simulation
- **Design Compiler**: Synthesize behavioral verilog into structural verilog

Why Learn These Tools

- These are used all over the place (in classes and industry)
- You will have to work with other people's tools your whole life
- Tools are not always perfect but are a necessary evil
  - Otherwise you'd be laying out every single transistor by hand img/melty

What is Make?

What is Make?

- Build System
  - Automatically build an *executable* from *source* files
  - Rules for building are stored in a `Makefile`
- Essentially a script of compilation commands
- Handles dependency resolution

Why do I care?

- We have lots of files that all need to be combined
- A simple command-line interface is less error-prone
- Want to avoid re-building files unnecessarily

Anatomy of a `Makefile`

`targets` are what we want to build

`dependencies` are what we need to build it

`commands` are how to build it

This looks something like the following

```
target: dep1 dep2 ...
        command1
        command2
```

Example Makefile Commands

```
build/simv: $(TESTBENCH) $(SOURCES)
	$(VCS) $^ -o $@
```

```
build/syn_simv: $(TESTBENCH) $(SYNTH_FILES)
	$(VCS) +define+SYNTH $^ $(LIB) -o $@
```

The $() syntax is a variable reference

Intro to VCS

What is VCS?

- Synopsys Verilog Compiler Simulator
- Builds Simulators from Verilog (structural or behavioral)
- We barely manage to brush the surface...

Why do we care?

- Knowledge is power...
- ...Specifically the power to debug

More VCS Options

`+define`	Define a `` `define `` macro Ex. +define+DEBUG, Ex. +define+CLK=10
`+lint=all`	Provide many more warnings
`-xprop=tmerge`	Makes `x|` propagate like they do in synthesis
`+incdir`	Adds a directory to search

Intro to Design Compiler and Synthesis

What is synthesis?

- The process of turning Behavioral Verilog into Structural Verilog
- Using a technology library
- There is a lot more to this, see the tcl file for details

Why do I care?

- Example of how your design might be built
- Informed guesses about...
  - Power
  - Area
  - Performance

Intro to Design Compiler and Synthesis

What is Design Compiler?

- The Synopsys synthesis tool
- Industry-leading
- Uses scripts written in the Tool command Language (TcL)

Intro to Design Compiler and Synthesis

We've simplified things!

- We've abstracted away much of the hassle of using `dc_shell` with the `470synth.tcl` file integrated with the Makefile.
- The Makefile communicates arguments to the TcL file using environment variables.
- The TcL file has an extensive description at the top, feel free to read
- The following Make target synthesizes a module whose name matches the name of the file.

```
export CLOCK_PERIOD = 10.0
TCL_SCRIPT = 470synth.tcl
synth/%.vg: %.sv | $(TCL_SCRIPT)
        MODULE=$* SOURCES="$^" dc_shell-t -f $(TCL_SCRIPT)
```

Tcl File Examples

```
set clock_name clock
set reset_name reset
set clock_period [getenv CLOCK_PERIOD]

proc eecs_470_generate_clock {clock_name clock_period} {
  set CLK_UNCERTAINTY 0.1 ;# the latency/transition time of the clock

  create_clock -period $clock_period -name $clock_name [find port $clock_name]
  set_clock_uncertainty $CLK_UNCERTAINTY $clock_name
  set_fix_hold $clock_name
}

proc eecs_470_set_wire_load {design_name} {
  set WIRE_LOAD tsmcwire
  set LOGICLIB lec25dscc25_TT

  set_wire_load_model -name $WIRE_LOAD -lib $LOGICLIB $design_name
  set_wire_load_mode top
  set_fix_multiple_port_nets -outputs -buffer_constants
}
```

Synthesis Process

Synthesis Steps

1. Elaboration: deconstructs Verilog to logic expressions
2. Logic Optimization: optimize boolean expressions between registers
3. Map to specified technology and generate netlist

Synthesis Behavior

- The tools can have weird behavior
- Synthesis will optimize to the constraints and do no more work than it needs to
- Pushing it past a previous limit may lead to new optimizations
- But, these new optimizations will incur a cost elsewhere (power, energy, area, etc.)
- Tries to match all long paths to the critical path

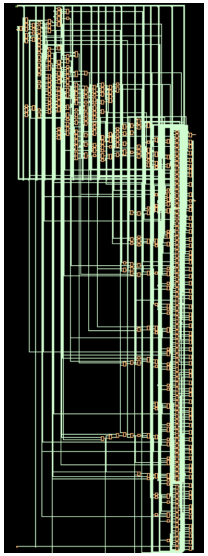
Synthesis Example: Lab 2

Lab 2 Adder Example

- Recognizes adder, can optimize that

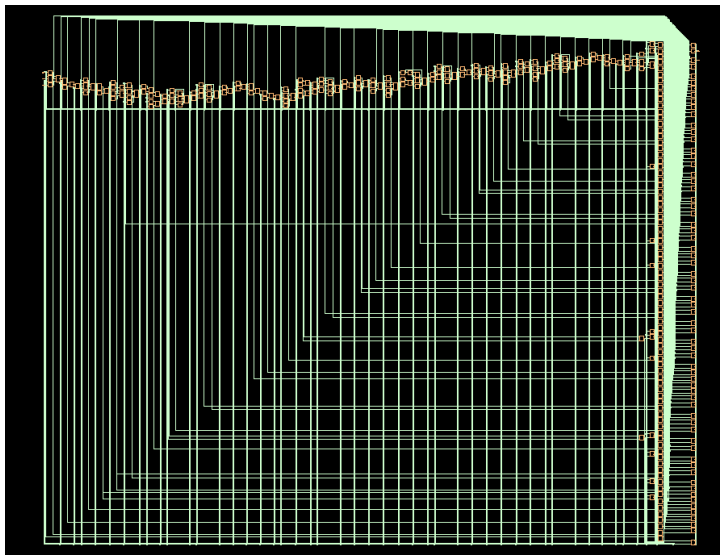
Clock period 1

- Synthesizes a ripple-carry adder
- Less area since delay is okay



Clock period 2 (smaller than 1)

- Optimizes structure
- Pays price in area



Project 2 Overview

Objectives

- Implement a sequential algorithm
- Understand design tradeoffs with pipelining
- Gain experience with synthesis

Part 1: Pipelined Multiplier

- Change the pipeline depth
- Synthesize at each size

Part 2: Wonka's Square Chocolate Bars (Integer Square Root)

- Finite state machine implementation
- Synthesis

Pipelined Multiplication

Partial Products

- Multiply the first n bits of the two components
- Multiply the next n bits, etc.
- Sum the partial products to get the answer

Pipelined Multiplication by Example

Binary Multiplication

$$
\begin{array}{rcccccc}
 & 0 & 0 & 0 & 1 & 1 & 1 \\
\times & & & 0 & 1 & 0 & 1 \\
\hline
 & 0 & 0 & 0 & 1 & 1 & 1 \\
 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 0 & 1 & 1 & 1 & 0 & 0 \\
+ & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
 & 1 & 0 & 0 & 0 & 1 & 1
\end{array}
$$

Decimal Multiplication

$$
\begin{array}{rcc}
 & & 7 \\
\times & & 5 \\
\hline
 & 3 & 5
\end{array}
$$

Pipelined Multiplication by Example

Example: 4-stage Pipelined Multiplication

multiplicand: 00001011

multiplier: 00000111

partial product: 00000000

$$
\begin{array}{r}
0\ 0\ 0\ 0\ 1\ 0\ 1\ 1 \\
\times\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1 \\
\hline
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0
\end{array}
$$

Pipelined Multiplication by Example

Example: 4-stage Pipelined Multiplication

multiplicand:	00001011	<< 2
multiplier:	00000111	>> 2
partial product:	00100001	

$$
\begin{array}{r}
0\ 0\ 0\ 0\ 1\ 0\ 1\ 1 \\
\times\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1 \\
\hline
0\ 0\ 1\ 0\ 0\ 0\ 0\ 1
\end{array}
$$

Pipelined Multiplication by Example

Example: 4-stage Pipelined Multiplication

multiplicand:	00101100
multiplier:	00000001
partial product:	00100001

$$
\begin{array}{r}
0\ 0\ 1\ 0\ 1\ 1\ 0\ 0 \\
\times\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \\
\hline
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0
\end{array}
$$

Pipelined Multiplication by Example

Example: 4-stage Pipelined Multiplication

multiplicand:	00101100	<< 2
multiplier:	00000001	>> 2
partial product:	01001101	

$$
\begin{array}{r}
0\ 0\ 1\ 0\ 1\ 1\ 0\ 0 \\
\times\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \\
\hline
0\ 0\ 1\ 0\ 1\ 1\ 0\ 0
\end{array}
$$

Part 2 Hints

ISR Algorithm

- Guess-and-check
- Loop from the top bit of the guess to the bottom
- Basically binary search for a solution

Hardware Implementation

- How do we implement this in hardware?

Part 2 Hints

ISR State Machine

Computing: $\sqrt{\texttt{value}}$

- On a `reset`
  - `guess` initialized to `32'h8000_0000`
  - `value` is clocked into a register
- `guess` gets the next bit set each time we cycle through the FSM again
- Square `guess` (multiply it with itself)
  - Wait until the multiplier raises its `done`
- if `guess <= value`
  - Keep the current bit
- else
  - Clear the current bit
- Move to the next bit
- After the last bit, raise `done`

Part 2 Hints

Reminders

- Remember to declare bitwidths for signals, e.g. `64'hFFFF_FFFF_FFFF_FFFF`
- It must take less than 600 cycles to compute a square root
- Remember to use the 8-stage multiplier for this
- Remember to check for proper reset behavior

Lab Assignment

Objectives

- Learn strategies for writing testbenches
- Synthesize a design

Logistics

- Assignment is posted on the course web site.
- If you get stuck...
  - Ask a neighbor!
  - Ask us in Office Hours!
  - Ask on Piazza!
- When you are finished, put yourself on help queue to get checked off
- Check-off is due at the end of the day of next lab

Appendix

Note

Everything from here to the end of the presentation is from an older version of this presentation, but kept here for reference.

Preprocessor Directives

What is a preprocessor?

- Prepares code/designs for compilation
  - Combines included files
  - Replaces comments and whitespace

Why do I care?

- Programmable
- Generic Designs (parameterized caches anyone?)

Verilog Macros

Definitions

- Syntax: `` `define <NAME> <value> ``
  - Note: That is a **tick**, not an apostrophe before define
  - The tick character is found with the ~, usually above tab (on US standard layout keyboards)
- Usage: `` `<NAME> ``
- Good for naming constants (can be used like wires)
- Convention dictates that macro names be in all caps
- Can also `` `undef <NAME> ``

Verilog Macros

Flow Control

Text inside flow control is conditionally included in the compiled source file.

- `` `ifdef `` Checks if something is defined
- `` `else `` Normal else behavior
- `` `endif `` End the if
- `` `ifndef `` Checks if something is not defined

Verilog Macros by Example

```
`define DEBUG
`define LOCKED 1'b0
`define UNLOCKED 1'b1
module turnstile(
    input coin, push,
    input clock, reset
    `ifdef DEBUG
        ,output logic state
    `endif
    );
    `ifndef DEBUG
        logic state;
    `endif
    always_comb begin
        next_state = state;
        if (state==`LOCKED  &&coin) next_state = `UNLOCKED;
        if (state==`UNLOCKED&&push) next_state = `LOCKED;
    end
    always_ff @(posedge clock) begin
        if (reset) state <= `LOCKED;
        else       state <= next_state;
    end
endmodule
```

Verilog Headers

What is inclusion?

- Paste the code from the specified file where the directive is placed

Where will I use this?

- System defines separated out, e.g.

```
`define ALU_OP_ADD 5'b10101
`define DCACHE_NUM_WAYS 4'b0100
```

- Macro assertion functions, printing functions, etc.
- Note: headers are named with the `.svh` extension

Parameters

What is a `parameter`?

- Constant defined inside a module
- Used to set module properties
- Can be overridden on instantiation

How do I use them?

- Definition

```
parameter NUM_CACHE_LINES = 8;
```

Setting Parameters

Overriding

- Set parameters for a module on instantiation
- Allows for different versions different places
- Usage:

```
cache #(.NUM_CACHE_LINES(4)) d_cache(...);
```

`defparam`

- Set parameters for a module
- Usage:

```
defparam dcache.NUM_CACHE_LINES = 4;
```

Macros Vs. Parameters

Macros

- Possibly globally scoped – namespace collision
- Use for modules that can change, but only have one instance
- Particularly for caches and naming arbitrary constants
- Needs the ` in usage

vs.

Parameters

- Locally scoped – no namespace collision
- Use for modules with many instances at different sizes
- Particularly for `generate` blocks (which are in a later lab)
- Does not need extra characters (like the `)

VCS by Example

```
VCS = SW_VCS=2020.12-SP2-1 vcs -sverilog +vc -Mupdate -line -full64 -kdb -lca
↪  -debug_access+all+reverse
```

`SW_VCS`	CAEN specific; sets which one of the installed VCS versions is run
`-sverilog`	Interpret designs using the SystemVerilog standard
`+vc`	Allow direct C code hooks in the design
`-Mupdate`	Compile incrementally
`-line`	Allow interactive debugging (might need other options)
`-full64`	Build 64 bit executables
`-kdb`	Generates Verdi Knowledge Database (KDB). KDB stores detailed design information from the source files
`-lca`	Enables Limited Customer Availability (LCA) features in Verdi.
`-debug_access`	Enables dumping to Fast Signal Database (FSDB) used by Verdi. FSDB stores the simulation results in a compact format. +all enables all debug capabilities and +reverse enables reverse debugging