

# EECS 470 Lab 2

## Synopsys Build System

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

January 18, 2024

# Overview

## Administrivia

## Generic Verilog Design

- Preprocessor Directives

- Parameters

- Hierarchical Design

## Tools

- Make

- VCS

- Design Compiler

## Project 2

# Administrivia

## Homework

- ▶ Homework 1 is due Thursday, 18th<sup>th</sup> January 11:59pm (tonight, or yesterday for Friday labs) on Gradescope

## Projects

- ▶ Project 1 is due Tuesday, 23<sup>th</sup> January 11:59 PM (turn in via submission script)
- ▶ Project 2 is due Tuesday, 30<sup>th</sup> January 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.

We are available to answer your questions. Office hours can be found on the course page and questions can be posted in the course Piazza.

# Administrivia

## Labs

- ▶ Lab 1 is due on 1/24 (next Wednesday)!
  - ▶ Note: The extension is for **lab 1 only**
- ▶ Lab 2 is due by 1/27!

# 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

# Verilog Headers

## 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

## and4.sv

---

```
`include "and2.v"
module and4(
    input [3:0] in,
    output logic out
);
    logic [1:0] tmp;
    and2 left (.a(in[1:0]),.x(tmp[0]));
    and2 right(.a(in[3:2]),.x(tmp[1]));
    and2 top (.a(tmp), .x(out));
endmodule
```

---

# Verilog Inclusion by Example

and2.sv

---

```
module and2(  
    input [1:0] a,  
    output logic x  
);  
    assign x = a[0] & a[1];  
endmodule
```

---

# Verilog Inclusion by Example

## Better and2.sv

---

```
`ifndef __AND_2_SV__  
`define __AND_2_SV__  
  
module and2(  
    input [1:0] a,  
    output logic x  
);  
    assign x = a[0] & a[1];  
endmodule  
  
`endif
```

---

# More Information

## Bibliography

Much of this information was taken from the **Verilog Preprocessor** paper by Wilson Snyder of Cavium Networks. It is *highly* recommended reading.

# 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 `)

# Array Connections

- ▶ Make a simple module and duplicate it several times
- ▶ 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

---

```
`define ADDR_WIDTH 8
module eight_bit_addr(
    input en, cin,
    input [(`ADDR_WIDTH-1):0] a, b,
    output [(`ADDR_WIDTH-1):0] sum,
    output cout);

    wire [(`ADDR_WIDTH-2):0] carries;

    one_bit_addr 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

# 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



# 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?

# 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?

- ▶ *simv* is an *executable*
- ▶ *Verilog* files are *source* files
- ▶ *vcs* is a compilation command

# 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

---

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

---

---

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

---

What does the `$()` syntax mean?

# Makefile Variables

A variable is a way to store commands or strings.

---

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

---

---

```
TESTBENCH = full_adder_1bit_test.sv  
SOURCES   = full_adder_1bit.sv
```

---

---

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

---

# Dependency Resolution

## What does it mean to be a dependency?

- ▶ Dependencies are used during compilation, e.g. list of source files
- ▶ Might be intermediate or primary

## How does Make resolve dependencies?

- ▶ Does the target exist?

No – Build it.

Yes – When was it built in relation to the dependencies?

After – Stop.

Before – Rebuild.

# Automatic Makefile Variables

`$@` The Current Target Name

`$^` A List of All Dependencies

`$<` The First Prerequisite

`$?` A List of All Dependencies Requiring Rebuilding

# Special Makefile Targets

- `.DEFAULT_GOAL` Sets what runs when `make` is executed with no arguments
  - `.PHONY` Dependencies will be built unconditionally
  - `.PRECIOUS` Dependencies will be kept, intermediate or not
  - `.SECONDARY` Dependencies are automatically treated as intermediates, but not deleted



# Make Resources

- ▶ **GNU Make Manual**
  - ▶ Special Targets
  - ▶ Automatic Variables
  - ▶ Makefile Conventions
- ▶ **Wikipedia - Make (Software)**

# 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

# 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

# More VCS Options

`+define` Define a ``define` macro

Ex. `+define+DEBUG`, Ex. `+define+CLK=10`

`+lint=all` Provide many more warnings

`-gui=verdi` Opens Verdi

`-o` Name of the executable generated

`-R` Run the executable after compilation

# Intro to Design Compiler and Synthesis

## What is synthesis?

- ▶ The process of turning Behavioral Verilog into Structural Verilog
- ▶ Using a technology library

## Why do I care?

- ▶ Example of how a 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!

- ▶ In new updates, 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 it!
- ▶ The following Make target synthesizes a module whose name matches the name of the file.

---

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

---

# Project 2 Overview

## Part 1: Pipelined Multiplier

- ▶ Change the pipeline depth
- ▶ Synthesize at each size

## Part 2: 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

$$\begin{array}{r}
 000111 \\
 \times 0101 \\
 \hline
 000111 \\
 000000 \\
 011100 \\
 + 000000 \\
 \hline
 100011
 \end{array}$$

## Decimal Multiplication

$$\begin{array}{r}
 7 \\
 \times 5 \\
 \hline
 35
 \end{array}$$

# Pipelined Multiplication by Example

## Example: 4-stage Pipelined Multiplication

multiplicand: 00001011

multiplier: 00000111

partial product: 00000000

$$\begin{array}{r}
 00001011 \\
 \times 00000011 \\
 \hline
 00000000
 \end{array}$$

# Pipelined Multiplication by Example

## Example: 4-stage Pipelined Multiplication

multiplicand: 00001011 << 2

multiplier: 00000111 >> 2

partial product: 00100001

$$\begin{array}{r}
 00001011 \\
 \times 00000111 \\
 \hline
 00100001
 \end{array}$$

# Pipelined Multiplication by Example

## Example: 4-stage Pipelined Multiplication

multiplicand: 00101100

multiplier: 00000001

partial product: 00100001

$$\begin{array}{r}
 00101100 \\
 \times 00000001 \\
 \hline
 00000000
 \end{array}$$

# Pipelined Multiplication by Example

## Example: 4-stage Pipelined Multiplication

multiplicand: 00101100 << 2

multiplier: 00000001 >> 2

partial product: 01001101

$$\begin{array}{r}
 00101100 \\
 \times 00000001 \\
 \hline
 00101100
 \end{array}$$

# 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:  $\sqrt{\text{value}}$

- ▶ 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  $\leq$  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

- ▶ 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