Synchronous Design

• Synchronous design means that all registers are clocked by the same signal.
• Synchronous design is always desirable in digital circuits.
• Not all events are synchronous.
Latch Instability

- Previously saw misbehaving RS latch in simulation: persistence of X values that would not happen with physical gates.
- Real RS latches can misbehave too.
- Simultaneous 0-0 (NAND) or 1-1 (NOR).
- Timing matters.
RS Latch

If R and S go to 1 simultaneously, the outputs are unpredictable and may result in oscillation.
RS Latch Timing

• Assume R=0, S = 1.
• Outputs are stable, Q=1, Qb=0.
• R switches to 1, nothing else changes.
• Q then goes to 0. Nothing else changes.
• Output is 0 0. Not correct functioning of the latch, but not dangerous.
Simultaneous Switching

- “Illegal” input condition 1-1 for NOR RS latch results in illegal output 0-0.
- System is stable.
- But what if both R and S switch from 0 to 1 at exactly the same time?
- System becomes unstable. May oscillate.
Signal Arrival Times

• In sequential circuits, signal arrival relationships matter.
• Combinational gate: makes no difference which signal arrives first. A NAND is a NAND is a NAND, no matter what the order of arrival.
• Not true for register inputs. Example: DFF requires D stable before CLK.
Before reaching final state, a change on D may have to go through 2 RS latches.

To have a stable state resulting from a change in D before the clock arrives, D must change before the clock.
Setup Time

RS latch: simultaneous arrival of inputs can cause unpredictable behavior.

In order to prevent unpredictable outputs, each flipflop will have certain operating parameters.

One of the key parameters is setup time. This is the amount of time before the clock signal arrives that other signals need to be stable.

Violation of setup times is a primary cause of malfunctioning circuits.
Timing Violations

JK Flipflop

JK FF.vsd
Asynchronous Inputs

• Clear and Preset are asynchronous in that they can change the output regardless of the clock.

• Their functioning, however, is not totally divorced from the clock.

• The “release” of an asynchronous input can still interfere with the functioning of a registers if it is not coordinated with the clock.
Release Violation

RST

CLK

D

Q X

Violation
Synchronizing Reset

- Reset must be able to put the circuit into a known state regardless of clock status—asynchronous assert.
- To avoid release violation, it must be de-asserted in sync with the clock—synchronous de-assert.
Asynchronous Assert, Synchronous De-assert
AASD Operation

• As soon as asynchronous reset goes active (low) flipflops go to reset condition, regardless of clock.
• When reset goes back high, output is no longer being forced low but isn’t forced high either.
• Going high waits for a clock pulse: synchronous de-assert.
Why Two Flipflops?

• De-assert can come any time, not synchronized with clock.
• AASD reset circuit uses two flipflops to minimize the probability of circuit failure due to metastability.
• Probability of such failure never goes to zero.
• Declines exponentially with several factors, including number of flipflops in chain.
• Two is enough for virtually all applications.
AASD Waves
Lab 4

Reset must be asynchronous assert, synchronous de-assert, per industry standard practices.

Must be coded behaviorally: no instances of primitives or even instances of flipflops.
Constants, Sort of Constants

• Previously used “define” macros to set constants.
• Parameters are similar.
• Main syntactical differences is that “defined” macros must have backtick (´) before each instance and parameters do not.
• Differences are more than syntactical.
`define

- `define may be done inside or outside of a module.
- A `define remains active until removed via `undef or over-written with a new `define.
- This makes the value associated with a `define ORDER dependent: files may use the same `define alias but associate different values with each.
`define Usage

• Define is best used only for global macros: operating information that is shared across all modules.

• A good technique is to place all `defines in a global definitions file, then include that file in compilation.
`include Compiler Directive

`include compiler directive is used to insert the contents of an entire Verilog file in another Verilog file. It has the following format:

```
`include "<file_name>"
```

- where `<file_name>` is the name of the file to be included which can be either relative or full UNIX path.

- This compiler directive is commonly used to include global and commonly used definitions and code.
- Nothing would preclude its use with operational code.
`include example

`include definitions.v → file “definitions.v”

\begin{verbatim}
`define buswidth 32
`define TRUE 1'b1
`define FALSE 1'b0
\end{verbatim}

Use relative or absolute addressing:

`include ../macros/arithmetic.v

`include ~userid/home/simulation/timing.v
`define and `include

Can be used to insure that a macro is defined before being used.

Example:

```
`ifdef CYCLE
  //do nothing
`else
  `define CYCLE 100
`endif
```

If Not Defined

• Verilog 2001 introduces `ifndef: If Not Defined.

• Usage:

  `ifndef CYCLE
    `include "definitions.vh"
  `endif
A previously-defined macro can be removed with an `undef directive.

- `undef CYCLE

Some authors recommend using this before changing a macro definition.

Better to never change a macro definition: parameters are meant to be changed, defined macros are not.
What to Define

• `define is best used for fundamental operating constants.

• Example: cycle time, the master clock frequency.

  `define CYCLE 10

  module tb_top;
  reg clk;
  initial begin
    clk = 1 ’b0;
    forever #(`CYCLE/2) clk = ~clk;
  end
Why Do That?

• Master clock is a fundamental operating constant.
• Constant?
• Constant across all modules.
• Constant until target technology changes.
• Then may need to be updated: easily done with one change to macro definition.
Parameters

• Parameters are used to declare constants. They are not variables.
• They are typically used to define delays and widths of variables.
• Parameters format is:

  \[
  \text{parameter } \text{<list_of_assignments>}\]

where:

\[
\text{<list_of_assignments>} \text{ is a comma separated list of parameters and their assigned values.}
\]
Parameter Examples

```plaintext
parameter p1 = 8,
real constant = 5.05,
bus_width = 16,
file = “/net/usr/design/file.data”;
```

Four parameters declared in one statement: an integer, a real number, another integer and a string. Only integers are useful in circuit descriptions.
More Parameters

parameter msb = 15;
parameter lsb = 0;
reg [msb:lsb] data_bus;
Parameter Usage

module register (Q, D, CLK, RST_N);
    parameter SIZE = 8;
    output [SIZE – 1 : 0] Q;
    input [SIZE – 1 : 0] D;
    input CLK, RST_N;
    reg [SIZE – 1 : 0] Q;
    always @(posedge CLK or negedge RST_N) begin
        if (!RST_N) Q <= 0;
        else Q <= D;
    end
endmodule
Verilog 2001 Enhancements

• Previous syntax was part of Verilog 1995.
• Language was expanded in 2001.
• Enhancements are piecemeal being incorporated into EDA tools.
• Not supported by Verilog XL but is now supported by NC Verilog.
module register2001 #(parameter SIZE = 8)

    (output reg [SIZE – 1 : 0] Q,

    input [SIZE – 1 : 0] D,

    input CLK, RST_N);

always @(posedge CLK or negedge RST_N)

    if (!RST_N) Q <= 0;

    else Q <= D;

endmodule
Verilog 2001 Ports & Parameters

• Previous example has several Verilog 2001 enhancements.
• Besides parameter, note “output reg” declaration.
• Now supported by simulator.
• May be used in lab.
  – Most stick with universally-accepted 1995 syntax.
Parameters for Delay Values

module params (A, B, OUT1, OUT2);
    input A, B;
    output OUT1, OUT2;
    reg OUT1, OUT2;
    parameter D1 = 5;
    parameter D2 = 10;
    always @(A or B) begin
        OUT1 = #D1 A & B;
        OUT2 = #D2 A | B;
    end
endmodule
Multiple Parameters, 1 Declaration

New (Verilog 2001) parameter declaration style:

```verilog
#parameter (D1 = 5, D2 = 10);
...
always @(A or B or C or D) begin
    OUT1 = #D1 A & B;
    OUT2 = #D2 C ^ D;
end
```
Parameter Usage

Example:

module Multiplier (product, op_a, op_b);
    parameter size = 8;
    input [size:1] op_a, op_b;
    output [2*size:1] product;
    always
        begin

        end
    endmodule
Bus Notation

• Previous example has bus bits ranging from 1 (LSB) to 8 (MSB).
• Not standard practice in any environment.
• Perfectly legal Verilog.
• Could also label bits from 3 (MSB) to 10 (LSB) just to be perverse.
• Writing obscure, convoluted code should never be a goal. It’s not cute or clever.
Bus Width Usage

• It is standard operating practice to have buses range from 0 (LSB) to width – 1 for the MSB:
  – reg [15:0] data_bus;

• Using parameters, this becomes:
  – parameter width = 16;
  – reg [width – 1 : 0] data_bus;
parameter WIDTH = 16;

reg [0 : WIDTH – 1] IBM_BUS;

//Now bit 0 is the MSB and bit 15 is the LSB.
always @(posedge CLK or negedge RST) begin
  if (!RST) IBM_BUS <= 0;
  else IBM_BUS <= IBM_BUS + 1;
//Bit 15, not bit 0, gets set on first clock after reset.
Parameters and Reusable Modules

A common usage of parameters is to create reusable, flexible modules.

Example: set maximum value for a counter with a parameter.

    parameter max_count = 15;
    always @(posedge CLK) begin
        if (count == max_count) count <= 0;
        else count <= count + 1;
Homework 2

- Homework 2 is on the course web site.
- Due a week from today.
Behavioral Code

• Lab 4 and all future labs are to be done in behavioral code.
• Do NOT use the mux from Lab 1 in Lab 5 or any other lab.
• Code the function using behavioral constructs rather than primitive operators.
• Do not include any delay specifications in the circuit description.
Using Verilog Simulator

• Verilog file names should end up with “.v” extension.
  Example: mux.v and tb_mux.v

• Syntax for Verilog command line is:
  verilog <command_line_options> <design_files>
  Example: ncverilog mux.v tb_mux.v

• A useful command line option is:
  -f <filename> which reads commands from the file <filename>.
  Example: ncverilog -f run.f
Force Files

• Even the command to invoke Verilog can be put in the force file.

• The more that goes in the file, the less you have to put on the command line.

• The less that goes on the command line, the fewer typographical errors you get.
Executable Force File

file tbmux.f:

```
ncverilog +gui –s +access+rwc mux.v tb_mux.v
```

File must be made executable: by default, newly-created files have read and write privileges, but not execute.

> chmod +x tbmux.f

Then run your simulation by just calling your force file from the command line:

> tbmux.f
Parameterized Modules

• Use of parameters allows flexibility in the reused of modules.
  – parameter maxcount = 15;
• File can easily be edited to change 15 to something else without searching for all places where maxcount is used.
• Parameters are more powerful than that.
Parameter Redefinition

- Parameters are constants that can be redefined.
- Redefinition can take place at compile time (simulation and/or synthesis).
- Once compiled, a parameter does not change.
- A parameter is not a variable like wire or reg.
Parameter Redefinition

• Three ways (not counting editing the file):
  – Module instance Parameter Value Assignment
  – “defparam”
  – Named parameter passing

• Named parameter passing requires 2001-compliant tools.
Parameter Redefinition

• Key concept in parameter redefinition: a parameter may be redefined only for a particular instance, leaving other instances unchanged.

• This is different from `define macros, which are global.
module tb_params();

    reg CLK, RST;
    //parameter width = 16;
    `define width 16
    wire [\`width - 1 : 0] COUNT;

    initial begin
        CLK = 1'b0;
        forever #10 CLK <= ~CLK;
    end

    initial begin
        RST = 1'b1;
        #25 RST = 1'b0;
        #20 RST = 1'b1;
    end

    PCNT   UUT(CLK, RST, COUNT);
endmodule

module PCNT(CLK, RST, COUNT);

    input CLK, RST;
    //parameter width = 4;
    `define width 4
    output [\`width - 1 : 0] COUNT;
    reg [\`width - 1 : 0] COUNT;

    always @(posedge CLK or negedge RST)
        if (!RST) COUNT <= 0;
        else COUNT <= COUNT + 1;
endmodule
`define vs. parameter

- If `define does not appear at all in second module, it will be fine: first definition will be used.
- If parameters are used, failing to put a it in both places will be a compile error.
- Width mismatch will compile with just a warning with either `define or parameter.
- After redefinition, “width” always means 4, not 16. It is applied everywhere “width” is encountered from then on.
Mismatches and Synthesis

• Width mismatches will generate a warning in simulation but the simulation will still run.

• Simulation tools generally are less sensitive than synthesis. Simulation warnings are frequently synthesis errors.

• Always fix simulation warnings even if the simulation appears to work.
Parameter Redefinition

• Changing a `define macro is discouraged.
  – Change is global.

• Redefining a parameter in an instance is standard procedure.
  – Only a particular instance is changed, scaling it for its unique environment.
  – Define is a global macro substitution. Parameter is instance-specific.
Parameter Value Assignment

module paramregs(CLK, D, Q);
  parameter WIDTH = 16;
  input CLK;
  input [WIDTH - 1 : 0] D;
  output [WIDTH - 1 : 0] Q;
  wire [WIDTH - 1 : 0] Q;
  pa_reg #(9) reg1(CLK, D[15:7], Q[15:7]);
  pa_reg #(7) reg2(CLK, D[6:0], Q[6:0]);
endmodule

module pa_reg(CLK, D, Q);
  parameter WIDTH = 1;
  input CLK;
  input [WIDTH - 1 : 0] D;
  output [WIDTH - 1 : 0] Q;
  reg [WIDTH - 1 : 0] Q;
  always @(posedge CLK) Q <= D;
endmodule
Parameter Value Assignment

• On the right: complete, synthesizable module of a flipflop.
• On the left: a module instantiating two copies of first module.
• Only it instructs the compilers (synthesis and simulation) to make one a nine-bit register and the other a seven-bit register.
• A single-bit flipflop module is never built.
Multiple Parameters

• Arbitrary numbers of parameters can be handled in the same way.
• Just need to write new values to each in order.

module FIFO(CLK, DATA_IN, DATA_OUT);

    parameter WIDTH = 16;
    parameter DEPTH = 256;

    ...

endmodule
Multiple Parameters

module top(CLK, RST, D, Q);
  input CLK, RST;
  input [11:0] D;
  output [11:0] Q;
  wire [11:0] Q;
  FIFO #(8, 1024) F1(CLK, D[7:0], Q[7:0]);
  FIFO #(4, 4096) F2(CLK, D[11:8], Q[11:8]);
  ... 
endmodule
Multiple Parameter Limitation

• Can’t skip over any parameters in the list.
• Don’t need to write to all, though.

\[
\text{FIFO } #(4) \ F2(\text{CLK, D}[11:0], \text{Q}[11:0]);
\]

• Changes only width, leaves depth at default (256).
• Trying to change only depth without writing to width would be an error.

\[
\text{FIFO } #(,4096) \ F2(\text{CLK, D}[11:0], \text{Q}[11:0]); //\text{Illegal syntax}
\]
Second Method: *defparam*

- Defparam is no longer recommended syntax.
- Proposal is for removing it from the language specification. Already gone from System Verilog (Verification Language).
- Not supported by all synthesis tools.
- Careless use can cause design errors.
- Will be used in lab exercise anyhow.
defparam usage

module paramregs(CLK, D, Q);
    parameter WIDTH = 16;
    input CLK;
    input [WIDTH - 1 : 0] D;
    output [WIDTH - 1 : 0] Q;
    wire [WIDTH - 1 : 0] Q;

    pa_reg #(9) reg1(CLK, D[15:7], Q[15:7]);
    defparam reg2.WIDTH = 7;
    pa_reg reg2(CLK, D[6:0], Q[6:0]);
endmodule

The way it’s supposed to work: just like parameter assignment, just different syntax.
defparam: the Problem

- The problem is that there are no limitations on defparam scope.
- Redefinition can go back up the hierarchy, sideway, anywhere.
Misuse of defparam

module pa_reg(CLK, D, Q);
  parameter WIDTH = 1;
  defparam tb_paramregs.WIDTH = 17;
  input CLK;
  input [WIDTH - 1 : 0] D;
  output [WIDTH - 1 : 0] Q;
  reg [WIDTH - 1 : 0] Q;
  always @(posedge CLK) Q <= D;
endmodule

This will change width in top-level to 17.
It will not resize modules in synthesis.
Synthesis/simulation mismatch.
More Misuse of defparam

```verilog
defparam r2.SIZE=4; // Design error!
endmodule
```

r2 width will be 4, not 16. Verilog calls for last defparam to over-ride any previous one.
Another *defparam* Problem

- Syntactically, there is no limit to the number of times a parameter can be redefined via *defparam*.
- If a parameter is redefined in multiple files, there is no guaranteed order of execution.
- There is no way to predict which value will be used.
Compiling defparams

- Because there is no limit on scope of a *defparam*, the final value of a hierarchical design using *defparams* can not be determined until all files have been read.
- One file may over-write all parameters in the entire design.
- This includes non-synthesizable files.
- Design rules often forbid the use of *defparam* at all.
Global Parameter and *defparam*

- Generally, once created a parameter is available for use throughout the module hierarchy.
- A value set with a global parameter statement can be overridden with a *defparam*:
  - Example:
    ```
    parameter chip_id = 16'h8754;
    defparam chip_id = 16'h9135;
    ```
Using defparam

• It is better not to use it at all.
• Any such use needs to be thoroughly documented and justified.
module dff (q, qb, data, clk);
output q, qb;
input data, clk;
parameter delay = 1;
//default delay parameter
dff_udp #(delay) (q, data, clk);
  not (qb, q);
endmodule
# Means a Parameter List Follows

- # means delay.
  
  ```
  # 10 A = 1'b1; /*wait 10 time units, then assign a value of 1 to variable A*/
  ```

- # means a parameter list follows when used with an instantiation.
  
  ```
  FIFO #(width, depth) F1 (DIN, DOUT);
  ```

- Meaning of # is context-dependent

- Format is module name, # parameter values, instance name, instance port list
module reg4 (q, d, clock);
output [3:0] q;
input [3:0] d;
input clock;
wire [3:0] q, d;
wire clock;

//port order connection, 2nd port not connected
dff u1 (q[0], , d[0], clock);

//port name connection, qb not connected
dff u2 (.clk(clock),.q(q[1]),.data(d[1]));

//explicit parameter redefine
dff u3 (q[2], ,d[2], clock);
defparam u3.delay = 3.2;

//implicit parameter redefine
dff #(2) u4 (q[3], , d[3], clock);
endmodule
Parameters Redefinition
Using \textit{defparam} statement.

Example:
module Test\_para ( ... );

Multiplier Mult1 (result1, op\_a1, op\_b1);
Multiplier Mult2 (result2, op\_a2, op\_b2);

\textbf{defparam} Mult1.size = 16, Mult2.size = 32;
// defparam could be anywhere, even in a different module.
endmodule
Redefining Parameters

• Redefinition of a parameters via defparam should be done sparingly and with great care.
• It has the potential to cause a great deal of mischief.
• It will almost certainly result in more convoluted, difficult to read Verilog.
• It ALWAYS ALWAYS ALWAYS needs to be documented.
Named Parameter Redefinition

• Also known as *in-line explicit redefinition*.
• Part of Verilog 2001 language expansion.
• Now supported by NC Verilog.
  – Not in Verilog XL
• Allows redefinition of a parameter anywhere in the parameter list without allowing hierarchical anarchy.
• Allowing such anarchy is Verilog tradition. Progress?
module paramregs(CLK, D, Q);
    parameter WIDTH = 16;
    input CLK;
    input [WIDTH - 1 : 0] D;
    output [WIDTH - 1 : 0] Q;
    wire [WIDTH - 1 : 0] Q;
    pa_reg #(9) reg1(CLK, D[15:7], Q[15:7]);
    pa_reg #( .WIDTH(7)) reg2(CLK, D[6:0], Q[6:0]);
endmodule

module pa_reg(CLK, D, Q);
    parameter WIDTH = 1;
    input CLK;
    input [WIDTH - 1 : 0] D;
    output [WIDTH - 1 : 0] Q;
    reg [WIDTH - 1 : 0] Q;
    always @(posedge CLK) Q <= D;
endmodule
Local and Global Parameters

- `localparam` can be used to define parameters for use only within a module.
- They have the odd property of NOT allowing value override with a `defparam` statement. Good or bad?
- Part of Verilog 2001: not supported by all tools.
Local Parameter Example

• A possible usage is in defining the states of a state machine: it would likely go badly if the parameter was changed in some other module but other modules could indeed have parameters for states.

  - Example:

    ```
    localparam state1 = 4'b0001,
    state2 = 4'b0010,
    state3 = 4'b0100,
    state4 = 4'b1000;
    ```
State Parameter Example

localparam state1 = 4’b0001,
state2 = 4’b0010,
state3 = 4’b0100,
state4 = 4’b1000;

• What kind of encoding is this? What’s the point?
• Using mnemonic names for states is good practice, but they should be meaningful names, not state1, state2, etc.
Use Meaningful Mnemonics

localparam IDLE = 2’b00,
FETCH = 2’b01,
DECODE = 2’b10,
EXECUTE = 2’b11;

/*Note parameter values are separated by commas: this is syntactically all one line*/
Parameter Review

• Parameters may be changed locally, unlike `define macros, which are global.

• A parameter must be declared in any instance in which it is referenced.

• A parameter always must have a default value.

• Parameters may be changed via module instance Parameter Value Assignment, named parameter passing or defparam statements.
Parameter Review

• Use of defparam is out of favor.
• Local parameters are used for mnemonic convenience but may not be redefined.
• Verilog does not have enumerated types like VHDL/SystemVerilog. Local parameters are as close as it comes.