EECS 270 W25 Discussion 11

April 3rd, 2025
By: Mick Gordinier

AGENDA

1. ModelSim and Debugging
2. Sequential Circuit Projects Recap
3. Register Transfer Level (RTL) Design Introduction
4. Project 7 Checkpoint Discussion
5. Going through FFC Starter Code

ModelSim and Debugging

Please Use ModelSim (Don't Use AG as Debugger)

- If you have not already done so watch these ModelSim discussions:
  - Discussion 2: Introduction to ModelSim (+Understanding Delays)
  - Discussion 5: Advanced ModelSim with Looking at Buggy Code
    - **UNDERSTANDING .do file, .mpf file, and RADICES - 18:30 to 28:30**

- Please follow the steps of getting a working Verilog Linter
  - Will allow you to quickly find small Verilog syntax errors

Sequential Circuit Projects Recap

Combinational Circuits Recap (Projects 1 - 4)

- Outputs depend ONLY on present combination of current inputs
- Stateless / No Memory
- Verilog ‘assign’ - Continuous assignment to a wire data type
- Verilog ‘always @*’ - Procedural “combinational” block

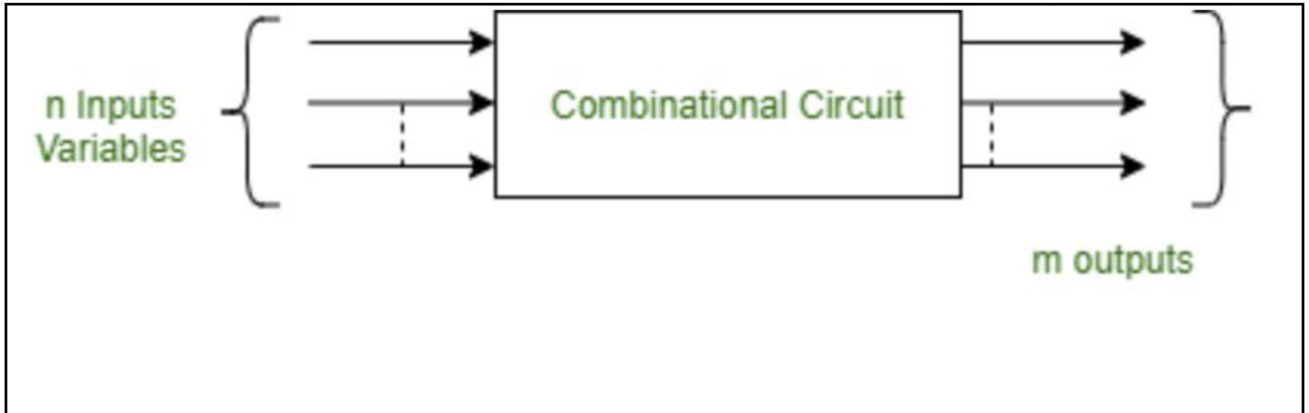
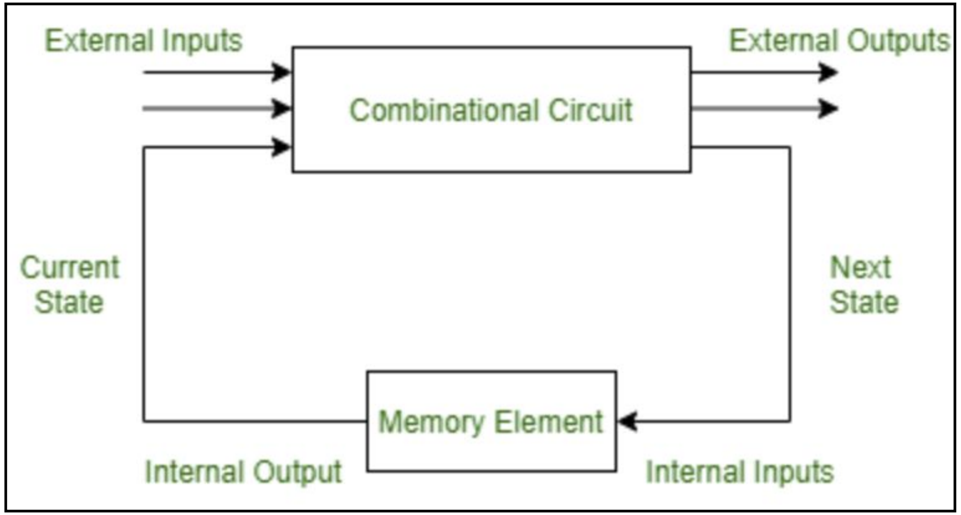


Figure - Block diagram of Combinational circuit

SRC

Sequential Circuits Recap (Project 5 - 7)

- Combinational Circuit + **Memory Element**
- Outputs depend on present combination of inputs and current state
- Use of memory to store previous state/output
- **NOTE: SEQUENTIAL CIRCUITS != SEQUENTIAL EXECUTION**

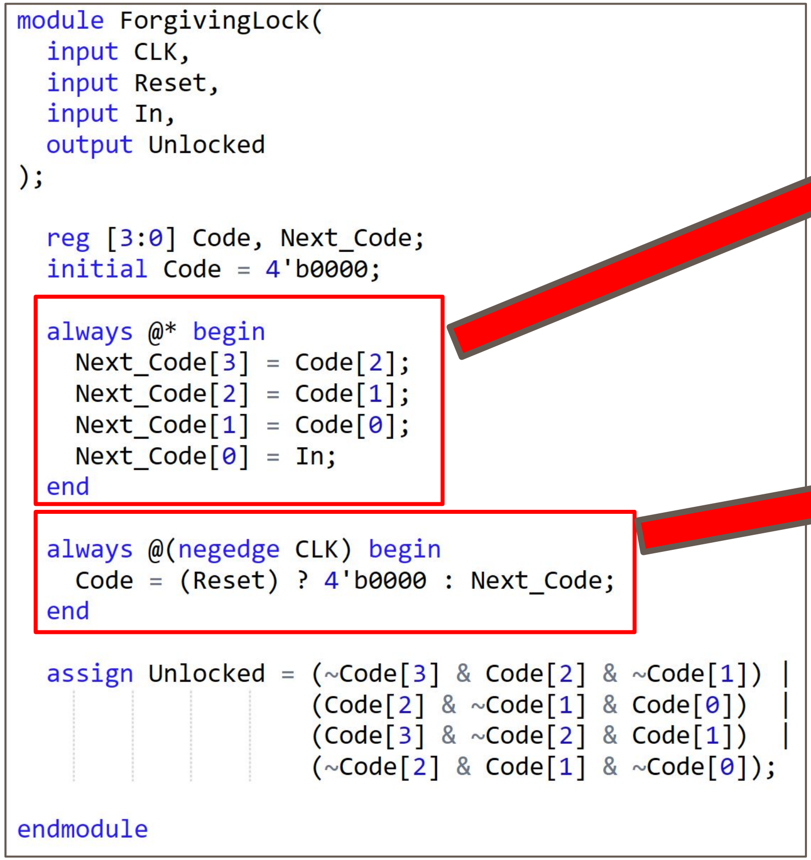


Sequential Circuit Terminology Recap

- **Synchronous Systems**
  - State changes simultaneously occur **ONLY** on an individual clock edge
  - **There should only be one global clock that your system uses**

- **Asynchronous Systems**
  - State changes can occur **AT ANY TIME**, not necessarily on a clock edge
  - Inputs or external events trigger state transitions **immediately**
  - Ex: Ripple Counters
    - The first flip-flop toggles on the clock edge
    - Each subsequent flip-flop is **triggered by the previous one's output.**

What is the Behavior Being Shown?



```
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  reg [3:0] Code, Next_Code;
  initial Code = 4'b0000;

  always @* begin
    Next_Code[3] = Code[2];
    Next_Code[2] = Code[1];
    Next_Code[1] = Code[0];
    Next_Code[0] = In;
  end

  always @(negedge CLK) begin
    Code = (Reset) ? 4'b0000 : Next_Code;
  end

  assign Unlocked = (~Code[3] & Code[2] & ~Code[1]) |
                    (Code[2] & ~Code[1] & Code[0])  |
                    (Code[3] & ~Code[2] & Code[1])  |
                    (~Code[2] & Code[1] & ~Code[0]);

endmodule
```

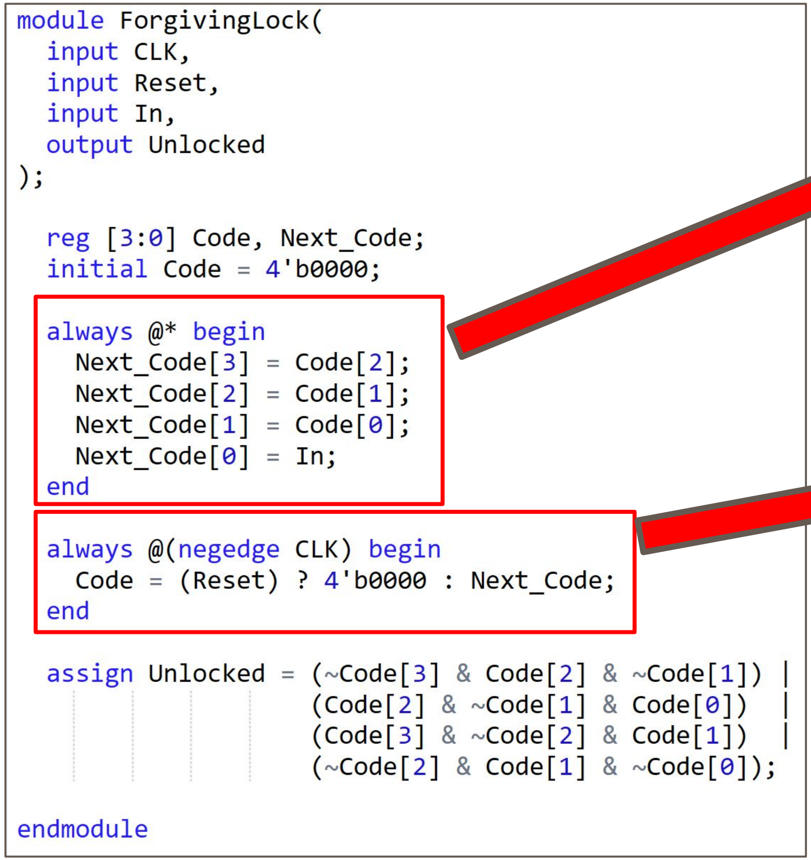
Combinational / Sequential

Asynchronous / Synchronous

Combinational / Sequential

Asynchronous / Synchronous

What is the Behavior Being Shown?



```verilog
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  reg [3:0] Code, Next_Code;
  initial Code = 4'b0000;

  always @* begin
    Next_Code[3] = Code[2];
    Next_Code[2] = Code[1];
    Next_Code[1] = Code[0];
    Next_Code[0] = In;
  end

  always @(negedge CLK) begin
    Code = (Reset) ? 4'b0000 : Next_Code;
  end

  assign Unlocked = (~Code[3] & Code[2] & ~Code[1]) |
                    (Code[2] & ~Code[1] & Code[0])  |
                    (Code[3] & ~Code[2] & Code[1])  |
                    (~Code[2] & Code[1] & ~Code[0]);

endmodule
```

Combinational

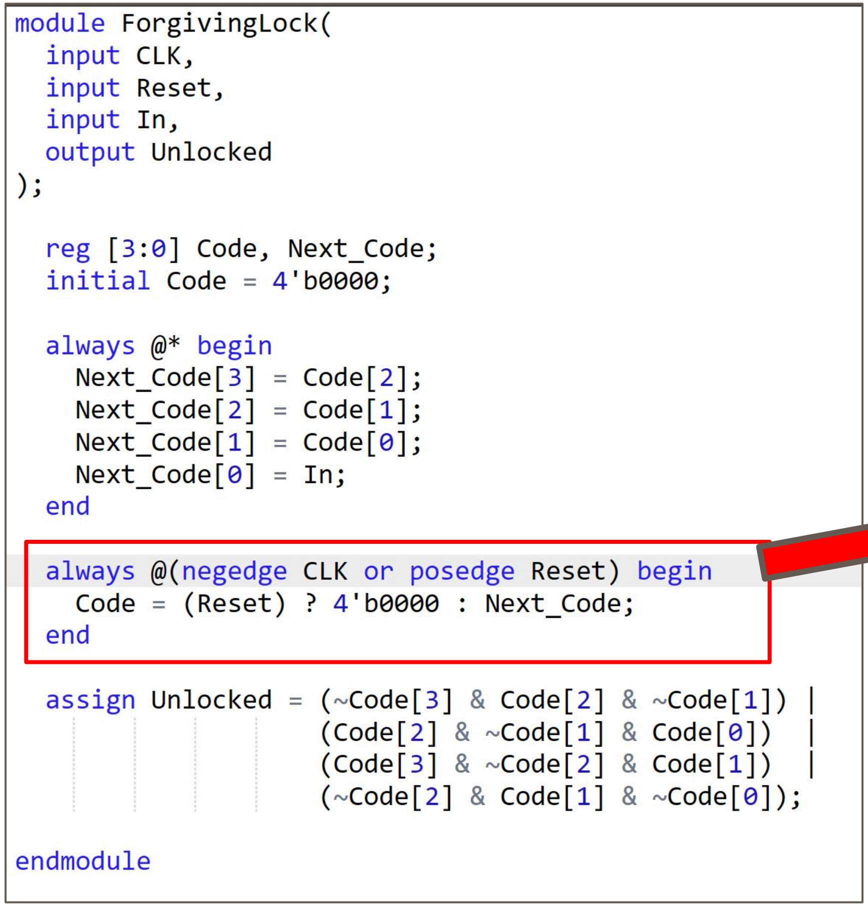
Asynchronous

Sequential

Synchronous

State change **ONLY** occurs on the edge of a shared clock

What is the Behavior Being Shown?

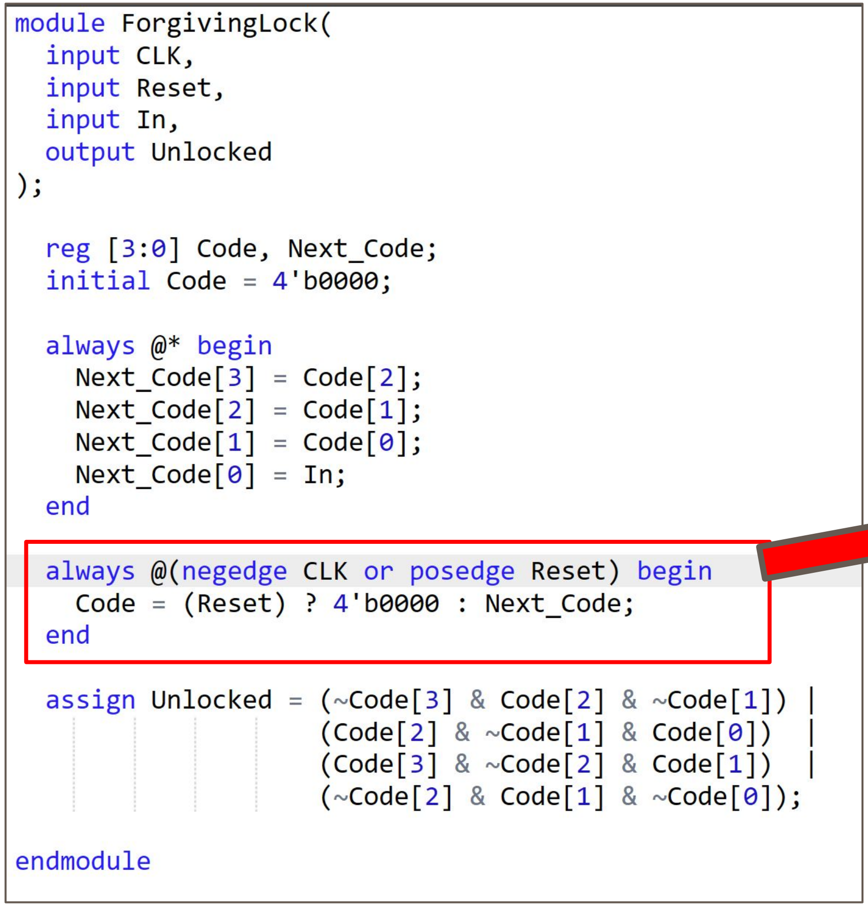


```verilog
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  reg [3:0] Code, Next_Code;
  initial Code = 4'b0000;

  always @* begin
    Next_Code[3] = Code[2];
    Next_Code[2] = Code[1];
    Next_Code[1] = Code[0];
    Next_Code[0] = In;
  end

  always @(negedge CLK or posedge Reset) begin
    Code = (Reset) ? 4'b0000 : Next_Code;
  end

  assign Unlocked = (~Code[3] & Code[2] & ~Code[1]) |
                    (Code[2] & ~Code[1] & Code[0])  |
                    (Code[3] & ~Code[2] & Code[1])  |
                    (~Code[2] & Code[1] & ~Code[0]);

endmodule
```

Combinational / Sequential

Asynchronous / Synchronous

What is the Behavior Being Shown?

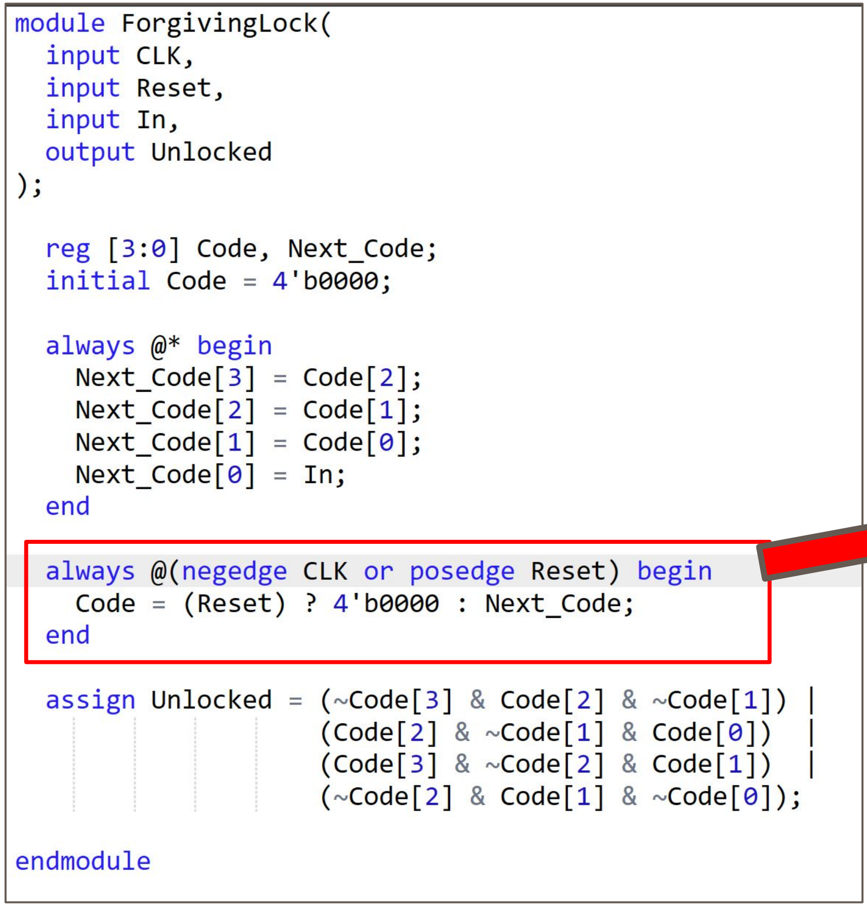


```
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  reg [3:0] Code, Next_Code;
  initial Code = 4'b0000;

  always @* begin
    Next_Code[3] = Code[2];
    Next_Code[2] = Code[1];
    Next_Code[1] = Code[0];
    Next_Code[0] = In;
  end

  always @(negedge CLK or posedge Reset) begin
    Code = (Reset) ? 4'b0000 : Next_Code;
  end

  assign Unlocked = (~Code[3] & Code[2] & ~Code[1]) |
                    (Code[2] & ~Code[1] & Code[0])  |
                    (Code[3] & ~Code[2] & Code[1])  |
                    (~Code[2] & Code[1] & ~Code[0]);

endmodule
```

Sequential

Asynchronous / Synchronous

What is the Behavior Being Shown?



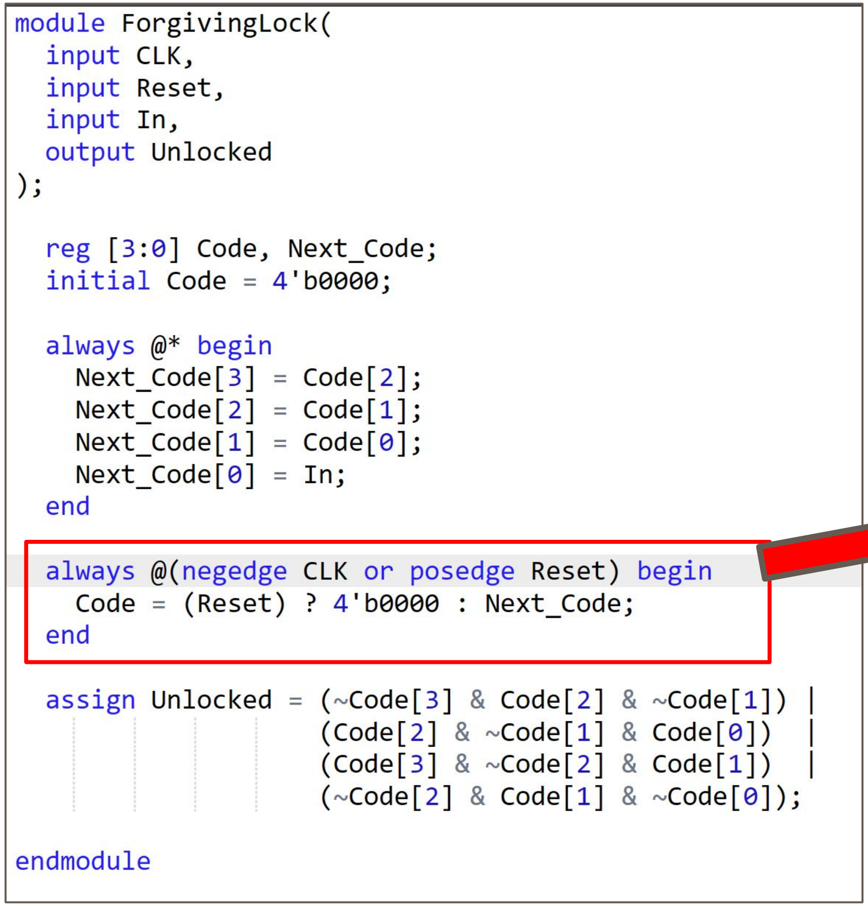
```verilog
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  reg [3:0] Code, Next_Code;
  initial Code = 4'b0000;

  always @* begin
    Next_Code[3] = Code[2];
    Next_Code[2] = Code[1];
    Next_Code[1] = Code[0];
    Next_Code[0] = In;
  end

  always @(negedge CLK or posedge Reset) begin
    Code = (Reset) ? 4'b0000 : Next_Code;
  end

  assign Unlocked = (~Code[3] & Code[2] & ~Code[1]) |
                    (Code[2] & ~Code[1] & Code[0])  |
                    (Code[3] & ~Code[2] & Code[1])  |
                    (~Code[2] & Code[1] & ~Code[0]);

endmodule
```

Sequential

ASYNCHRONOUS

State change **CAN** occur when reset goes high **(Ignoring the clock)**

PLEASE DO NOT USE ASYNCHRONOUS RESETS



```verilog
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  reg [3:0] Code, Next_Code;
  initial Code = 4'b0000;

  always @* begin
    Next_Code[3] = Code[2];
    Next_Code[2] = Code[1];
    Next_Code[1] = Code[0];
    Next_Code[0] = In;
  end

  always @(negedge CLK or posedge Reset) begin
    Code = (Reset) ? 4'b0000 : Next_Code;
  end

  assign Unlocked = (~Code[3] & Code[2] & ~Code[1]) |
                    (Code[2] & ~Code[1] & Code[0])  |
                    (Code[3] & ~Code[2] & Code[1])  |
                    (~Code[2] & Code[1] & ~Code[0]);

endmodule
```

"My friend told me to add it"

~~Sequential~~

ASYNCHRONOUS

State change **CAN** occur when reset goes high **(Ignoring the clock)**

Register Transfer Level (RTL) Design Introduction

Verilog’s Many Levels of Abstraction

“BEHAVIORAL VERILOG”	**Behavioral Level**	Project 3B - 7
	Dataflow Level	
“STRUCTURAL VERILOG”	**Gate Level**	Projects 0 - 3A
	Switch Level	Because of Shannon, we can ignore

Verilog's Many Levels of Abstraction

“BEHAVIORAL VERILOG”

Behavioral Level

Register Transfer Level (RTL)

Dataflow Level

Project 3B - 7

“STRUCTURAL VERILOG”

Gate Level

Projects 0 - 3A

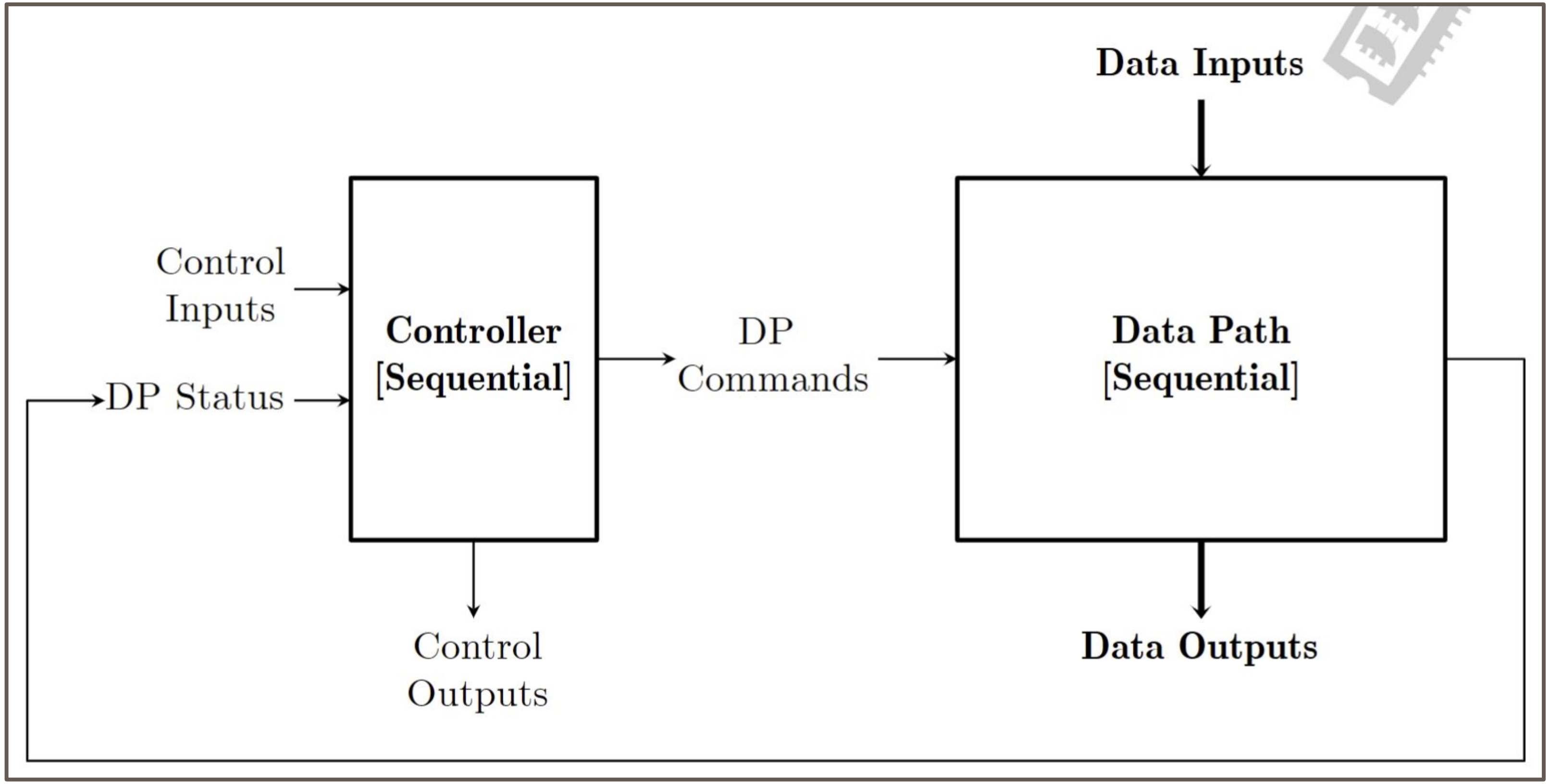
Switch Level

Because of Shannon, we can ignore

RTL Design Overview

- Splitting up design in Controller and Datapath
- **Controller** - Controls and orchestrates the Datapath operations
- **Datapath** - Computation on the signals
  - Arithmetic - Add/Subtract/Multiple/Count/…
  - Logical - Shift Left / Shield Right / Arithmetic Shift / bitwise Ops
  - Other - Clear / Load / Hold

RTL Sequential Circuit



Main Differences with Behavioral / RTL Design

Feature	**Behavioral Verilog**	**RTL Verilog**
Abstraction Level	Very High (Nearly Software)	Medium (Closer to Hardware)
Synthesizable	Sometimes (Not everything can be synthesized to hardware)	Always
Main Use Case	Simulation and quick modeling	Actual hardware implementation
Need to Explicitly Describe Underlying Gates/Hardware	No	No

Project 7 Checkpoint Discussion (Sequential Calculator)

Project 7 Deadlines

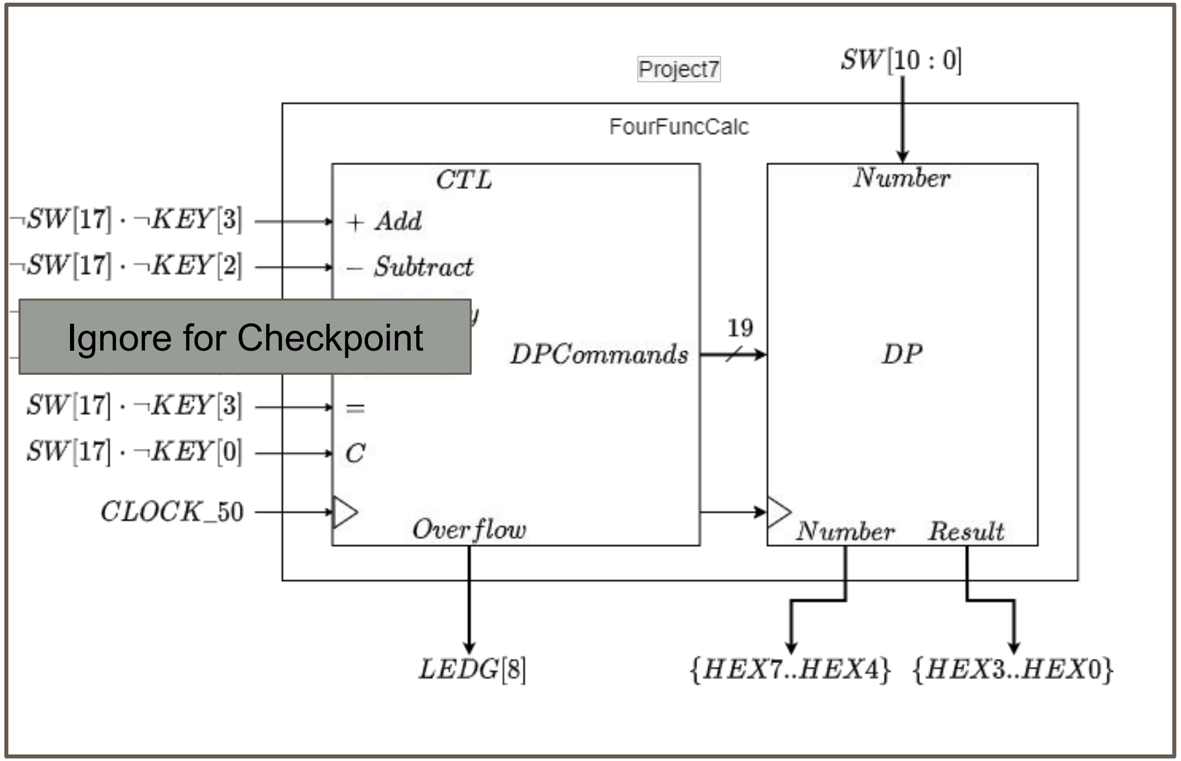
- Project 7 Checkpoint (Due: 4/15)
    - Implemented Adding and Subtracting in your Four Func Calculator
    - **Worth 10% of your grade**
    - Will require a testbench for adding and subtracting
    - **Allows us to run your project on FPGA to ensure no timing issues**

- Project 7 Final (Due: 4/25)
    - Implementing Multiplication and Division in your Four Func Calculator
    - Requires full testbench
    - **Worth 90% of your grade**

- **NOTE: THERE IS NO IN-PERSON SIGN OFFS FOR THIS PROJECT**

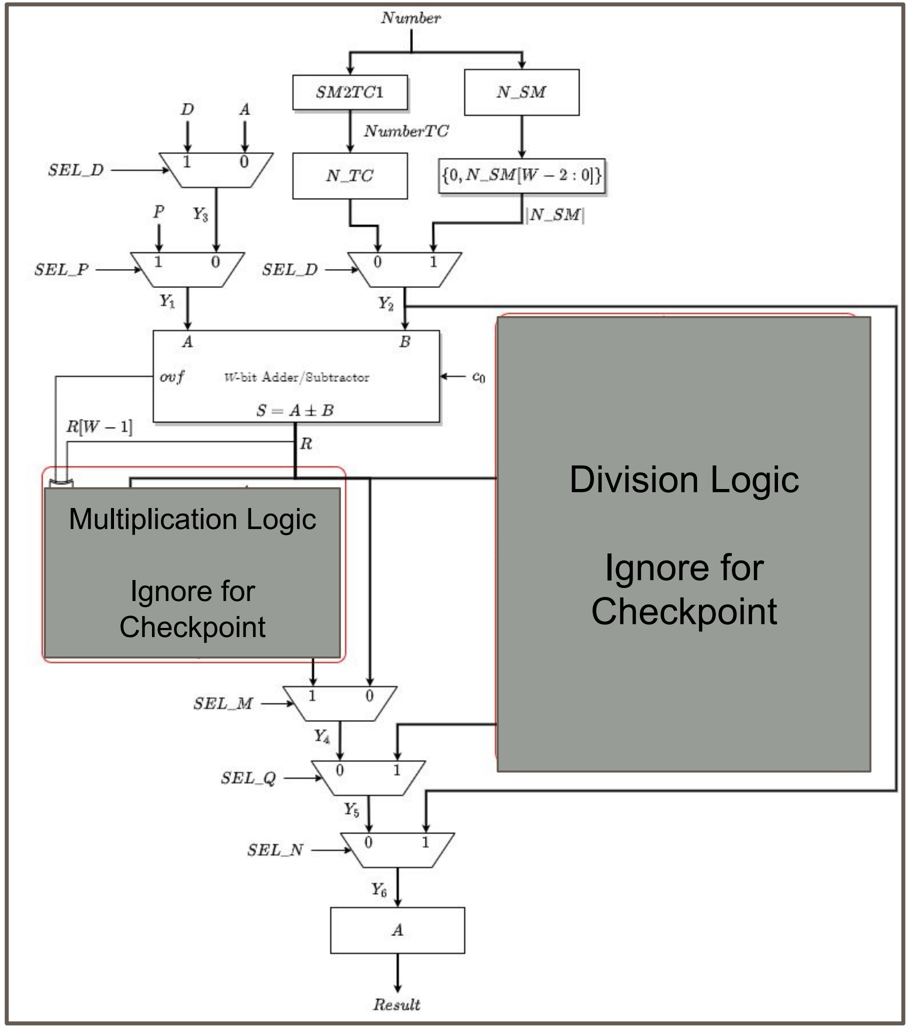
Four Function Calculation (Checkpoint) Overview

- Controller Input
  - **+, -, =, C**
  - 50 MHz Clock
- Datapath Input
  - DPCommands
  - Number
- Outputs
  - HEX Displays
  - Overflow bit



FFC Datapath (CP)

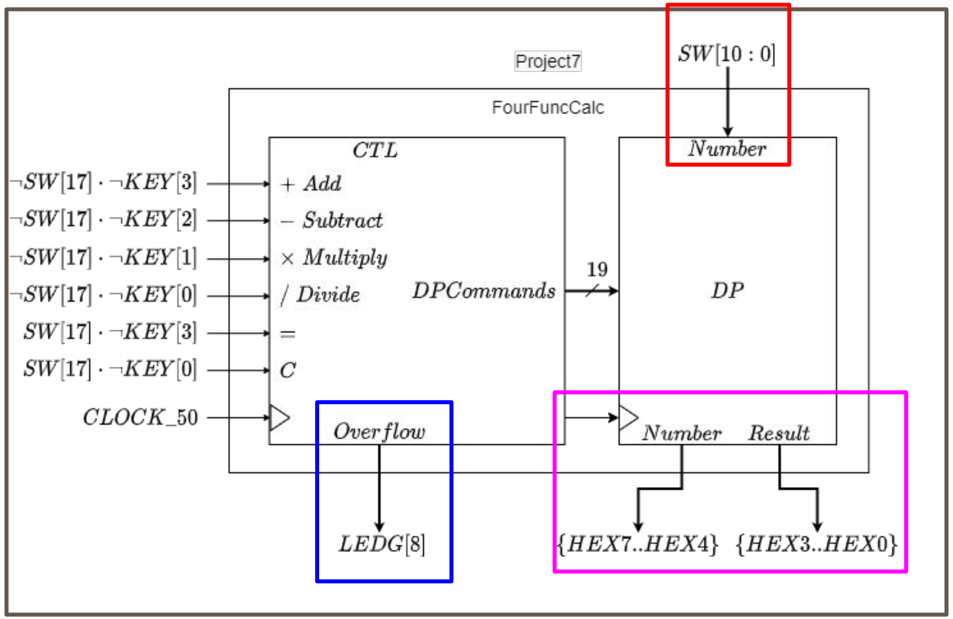
- Only need to focus on implementing add/sub datapaths
- **NOTE:** Must only have 1 instantiation of the AddSub modules

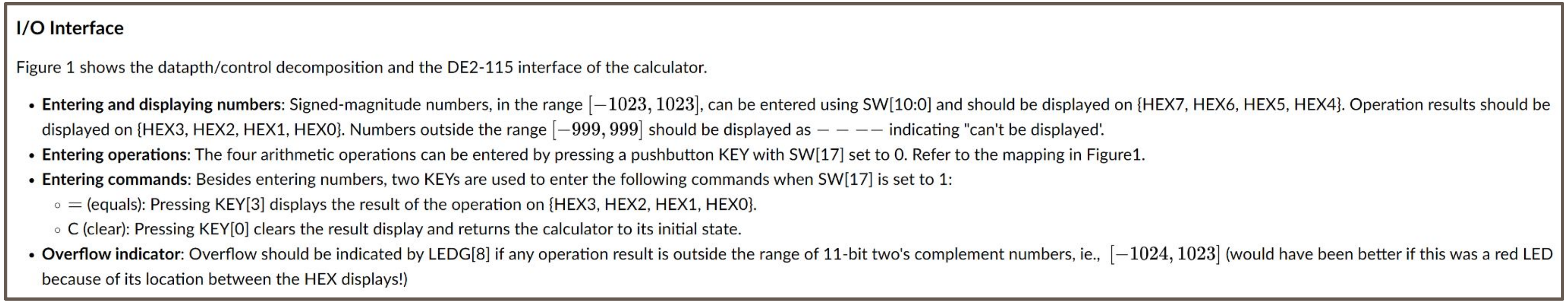


I/O Interface Notes

- Dealing with 11-bit numbers
- **Number SWs inputted as Signed-Magnitude**
- But computation in datapath assumes 2's complement

- Overflow - Outside 11-bit 2C range
  - [-1024, 1023]
- vs. Display "----" if can't be displayed
  - [-999, 999]





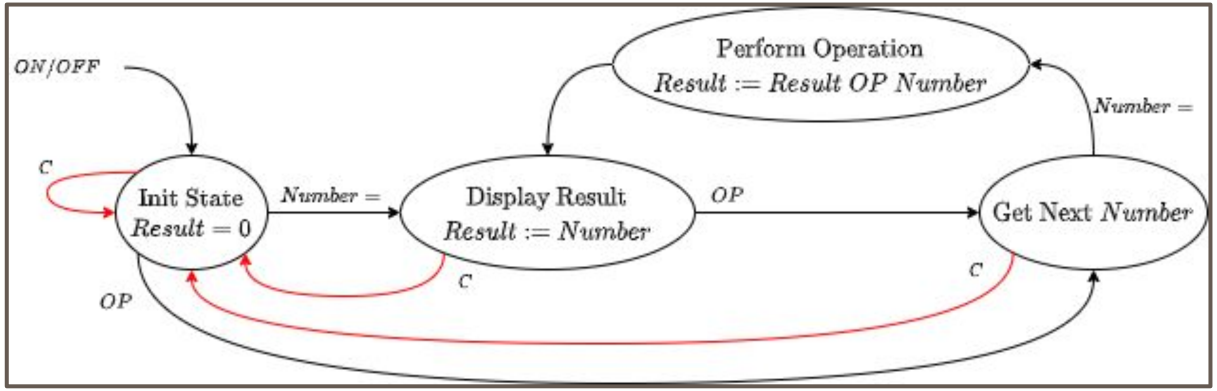
I/O Interface

Figure 1 shows the datapth/control decomposition and the DE2-115 interface of the calculator.

- **Entering and displaying numbers**: Signed-magnitude numbers, in the range $[-1023, 1023]$, can be entered using SW[10:0] and should be displayed on {HEX7, HEX6, HEX5, HEX4}. Operation results should be displayed on {HEX3, HEX2, HEX1, HEX0}. Numbers outside the range $[-999, 999]$ should be displayed as $- - - -$ indicating "can't be displayed'.
- **Entering operations**: The four arithmetic operations can be entered by pressing a pushbutton KEY with SW[17] set to 0. Refer to the mapping in Figure1.
- **Entering commands**: Besides entering numbers, two KEYs are used to enter the following commands when SW[17] is set to 1:
  - $=$ (equals): Pressing KEY[3] displays the result of the operation on {HEX3, HEX2, HEX1, HEX0}.
  - C (clear): Pressing KEY[0] clears the result display and returns the calculator to its initial state.
- **Overflow indicator**: Overflow should be indicated by LEDG[8] if any operation result is outside the range of 11-bit two's complement numbers, ie., $[-1024, 1023]$ (would have been better if this was a red LED because of its location between the HEX displays!)

FFC High-Level FSM

- When turned on, should display 0 as result
    - Can either load in a value or perform operation from initialization
- Then, you perform many OP → "Number ="
- Clearing should take you straight back to 0



Turn On	0
5 =	5
−	5
+	5
×	5
7 =	35
+	35
−	35
5 =	30
/	30
4 =	7
C	0

Timing Errors in Project 7

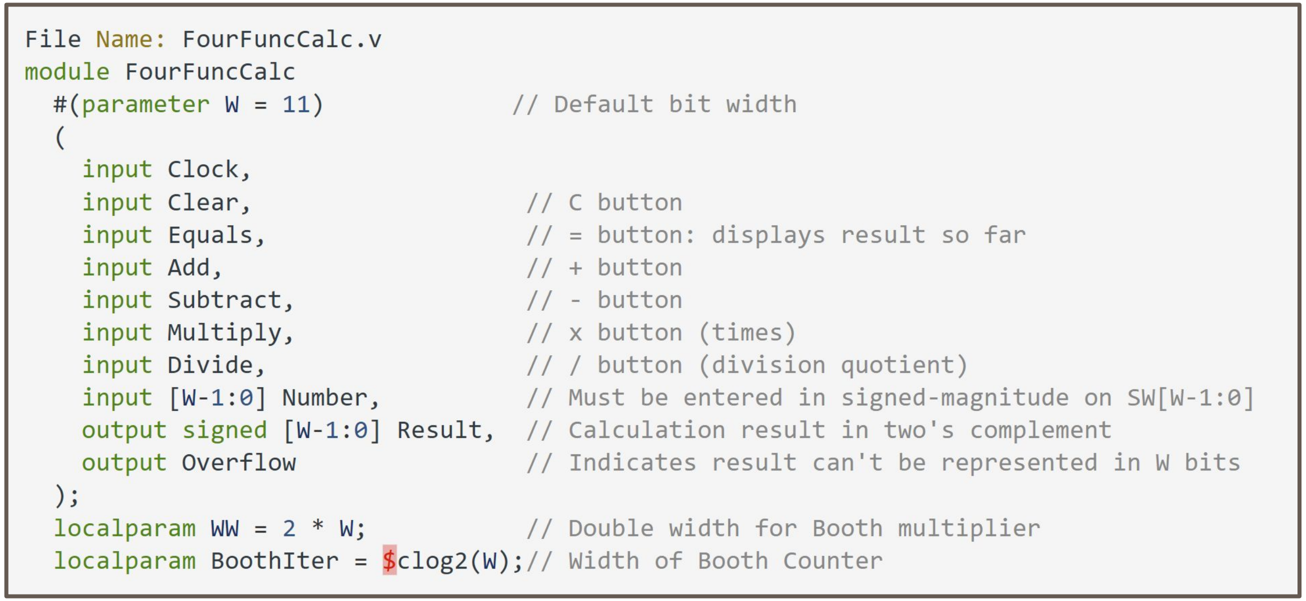
- EECS 270 does not focus on how to handle timing errors
    - **Take EECS 470 to dive deeper into timing issues**

- If you occur timing errors
    - Use Clock_Div module
    - **Slow down the clock**

Going through FourFuncCalc starter code

FourFuncCalc Starter Code

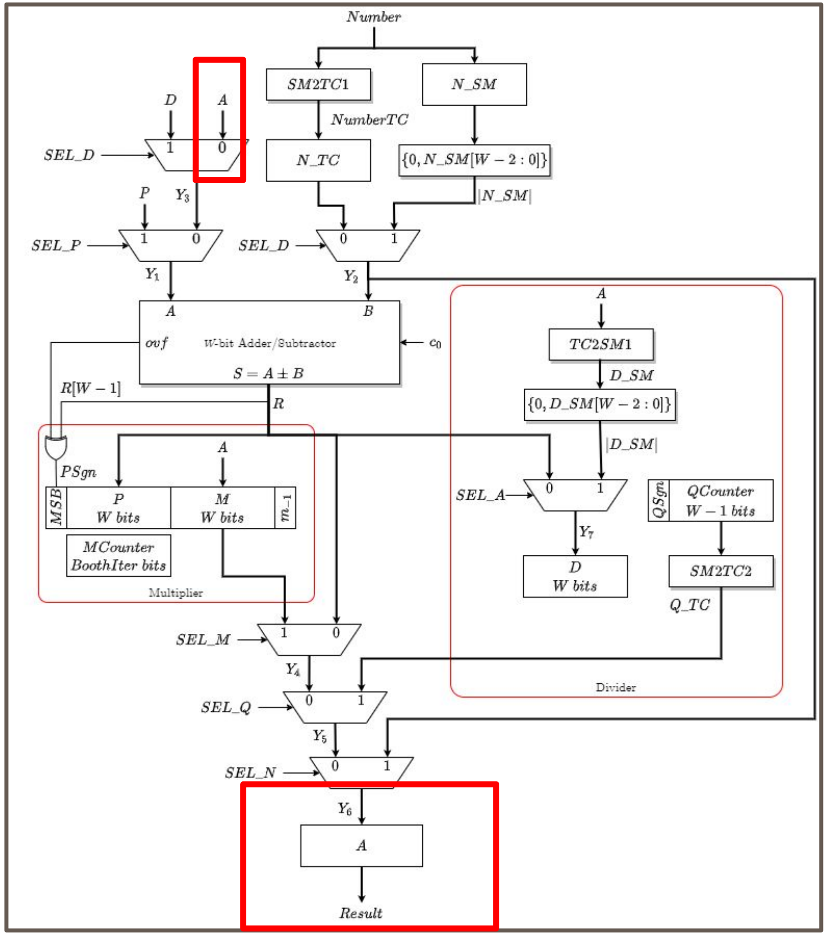
- **Module that handles both the controller and datapath logic**
- Instantiated by the top level module “Project7”
- **IT IS NOT COMPLETED**



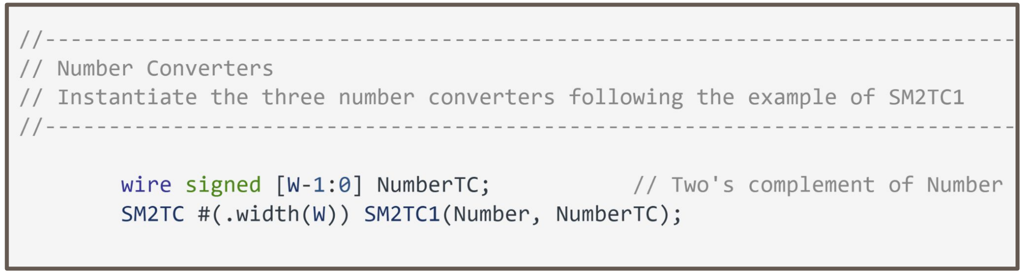
```
File Name: FourFuncCalc.v
module FourFuncCalc
  #(parameter W = 11)             // Default bit width
  (
    input Clock,
    input Clear,                   // C button
    input Equals,                  // = button: displays result so far
    input Add,                     // + button
    input Subtract,                // - button
    input Multiply,                // x button (times)
    input Divide,                  // / button (division quotient)
    input [W-1:0] Number,          // Must be entered in signed-magnitude on SW[W-1:0]
    output signed [W-1:0] Result,  // Calculation result in two's complement
    output Overflow                // Indicates result can't be represented in W bits
  );
  localparam WW = 2 * W;           // Double width for Booth multiplier
  localparam BoothIter = $clog2(W);// Width of Booth Counter
```

Datapath Components (1)

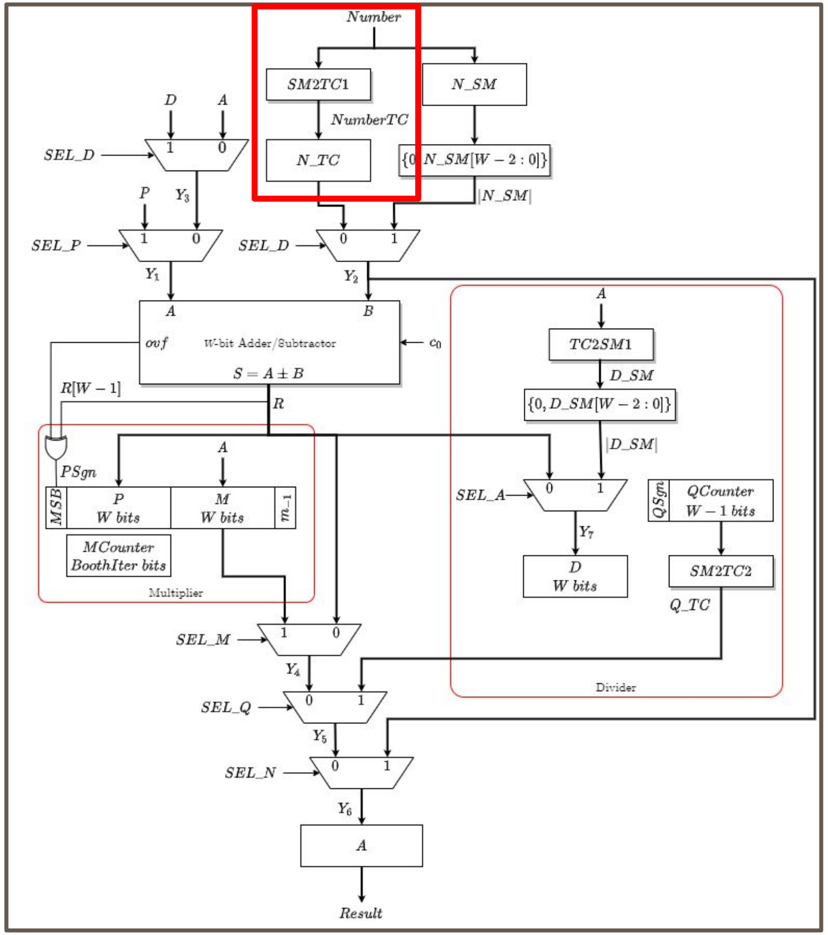
```
//---------------------------------------------------------------------------
// Registers
// For each register, declare it along with the controller commands that
// are used to update its state following the example for register A
//---------------------------------------------------------------------------

      reg signed [W-1:0] A;                   // Accumulator
      wire CLR_A, LD_A;                       // CLR_A: A <= 0; LD_A: A <= Q
```



Datapath Components (2)



```
//---------------------------------------------------------------------------
// Number Converters
// Instantiate the three number converters following the example of SM2TC1
//---------------------------------------------------------------------------

      wire signed [W-1:0] NumberTC;          // Two's complement of Number
      SM2TC #(.width(W)) SM2TC1(Number, NumberTC);
```



REMEMBER: Number is coming in as Signed Magnitude representation

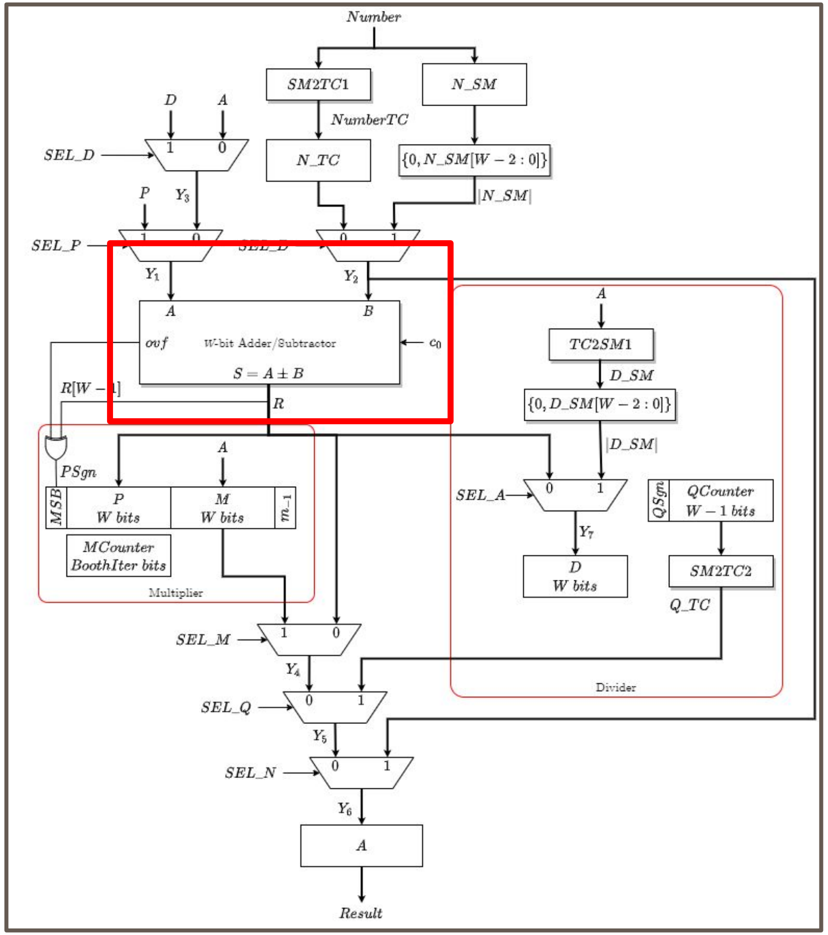
REMEMBER: Adder/Subtractor does computation on 2C representation

Datapath Components (3)

```
//--------------------------------------------------------------------------------------
// Adder/Subtractor
//--------------------------------------------------------------------------------------

        wire c0;                                // 0: Add, 1: Subtract
        wire ovf;                               // Overflow
        AddSub #(.W(W)) AddSub1(Y1, Y2, c0, R, ovf);
        wire PSgn = R[W-1] ^ ovf;               // Corrected P Sign on Adder/Subtractor overflow
```

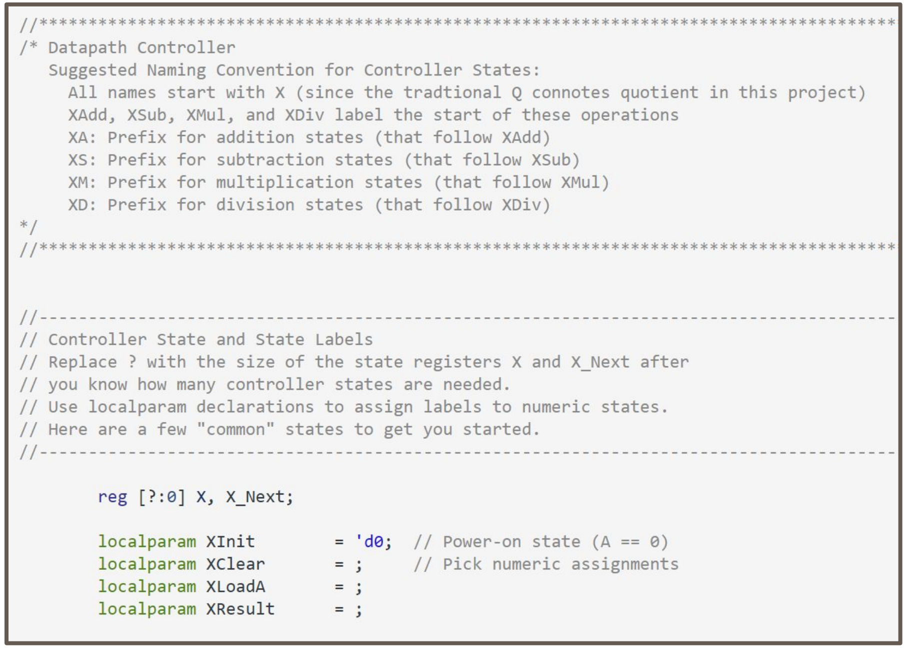
```
// Full Adder
module FullAdder
        (
                input a, b, cin,
                output s, cout
        );

        assign s = a ^ b ^ cin;
        assign cout = a & b | cin & (a ^ b);
endmodule // FullAdder
```

```
module AddSub
        #(parameter W = 16)
        (
                input [W-1:0] A, B,
                input c0,
                output [W-1:0] S,
                output ovf
        );

        wire [W:0] c;
        assign c[0] = c0;

// Instantiate and "chain" W full adders
        genvar i;
        generate
                for (i = 0; i < W; i = i + 1)
                        begin: RCAddSub
                                FullAdder FA(A[i], B[i] ^ c[0], c[i], S[i], c[i+1]);
                        end
        endgenerate

// Overflow
                assign ovf = c[W-1] ^ c[W];
endmodule // AddSub
```



AddSub has been implemented for you

Controller State Assignment

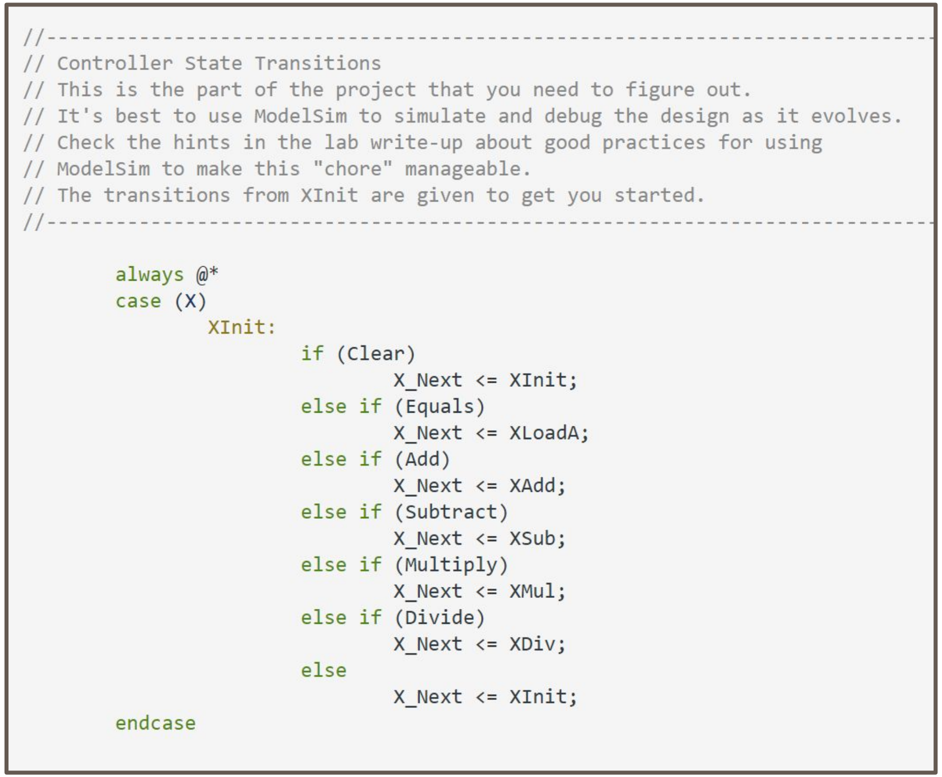


```
//***************************************************************************************
/* Datapath Controller
   Suggested Naming Convention for Controller States:
     All names start with X (since the tradtional Q connotes quotient in this project)
     XAdd, XSub, XMul, and XDiv label the start of these operations
     XA: Prefix for addition states (that follow XAdd)
     XS: Prefix for subtraction states (that follow XSub)
     XM: Prefix for multiplication states (that follow XMul)
     XD: Prefix for division states (that follow XDiv)
*/
//***************************************************************************************


//---------------------------------------------------------------------------------------
// Controller State and State Labels
// Replace ? with the size of the state registers X and X_Next after
// you know how many controller states are needed.
// Use localparam declarations to assign labels to numeric states.
// Here are a few "common" states to get you started.
//---------------------------------------------------------------------------------------

        reg [?:0] X, X_Next;

        localparam XInit        = 'd0;  // Power-on state (A == 0)
        localparam XClear       = ;     // Pick numeric assignments
        localparam XLoadA       = ;
        localparam XResult      = ;
```

You need to determine all the states to your FFC Controller.

We have given you a few states to first consider. Can change and add your own.

Controller State Transition (Next State Logic)



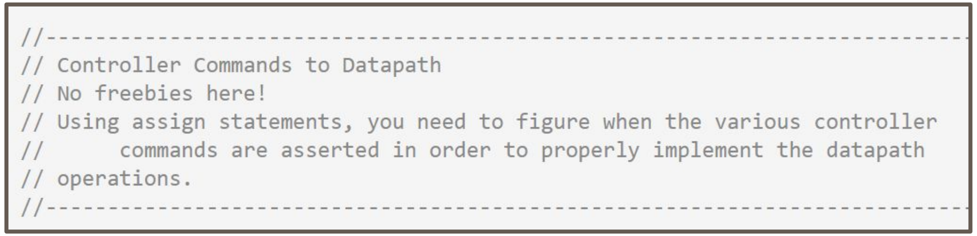
```
//------------------------------------------------------------------------------
// Controller State Transitions
// This is the part of the project that you need to figure out.
// It's best to use ModelSim to simulate and debug the design as it evolves.
// Check the hints in the lab write-up about good practices for using
// ModelSim to make this "chore" manageable.
// The transitions from XInit are given to get you started.
//------------------------------------------------------------------------------

        always @*
        case (X)
                XInit:
                        if (Clear)
                                X_Next <= XInit;
                        else if (Equals)
                                X_Next <= XLoadA;
                        else if (Add)
                                X_Next <= XAdd;
                        else if (Subtract)
                                X_Next <= XSub;
                        else if (Multiply)
                                X_Next <= XMul;
                        else if (Divide)
                                X_Next <= XDiv;
                        else
                                X_Next <= XInit;
        endcase
```

You need to determine all the controller transitions of your FFC Controller.

REMEMBER: Controller inputs are:

1. **State of the datapath and**
2. **User inputs**

Controller State Command Bits (Output Logic)

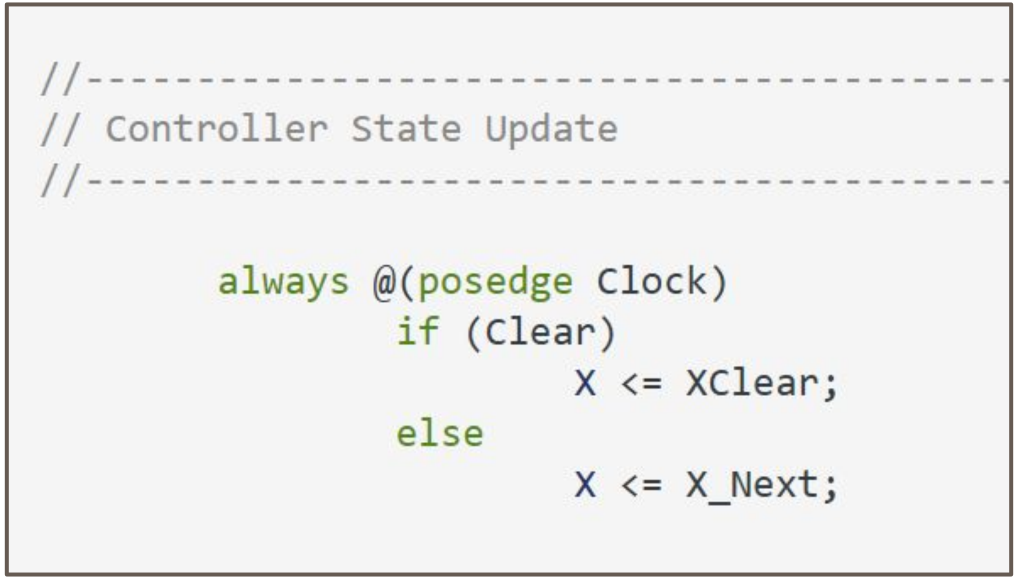


```
//----------------------------------------------------------------------------
// Controller Commands to Datapath
// No freebies here!
// Using assign statements, you need to figure when the various controller
//      commands are asserted in order to properly implement the datapath
// operations.
//----------------------------------------------------------------------------
```

You need to determine which controller command bits goes high given what controller state you are in

Assume the controller bits are Moore outputs

Controller State Update (Memory Logic)



```
//-----------------------------------------
// Controller State Update
//-----------------------------------------

        always @(posedge Clock)
                if (Clear)
                        X <= XClear;
                else
                        X <= X_Next;
```

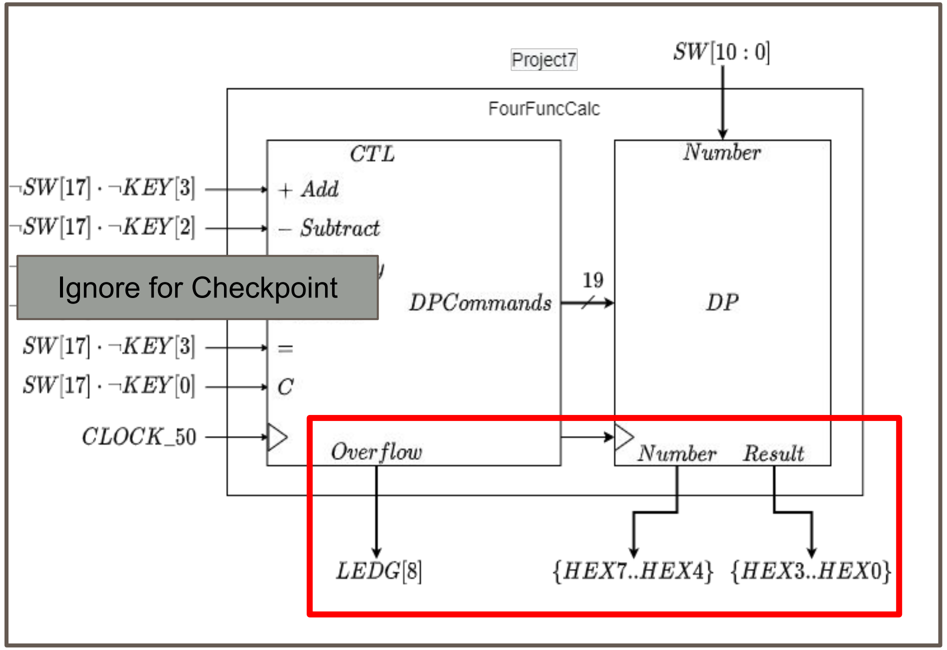
Datapath State Update (Memory Logic)

```
//----------------------------------------------------------
// Datapath State Update
// This part too is your responsibility to figure out.
// But there is a hint to get you started.
//----------------------------------------------------------

        always @(posedge Clock)
        begin
                N_TC <= LD_N ? NumberTC : N_TC;

        end
```

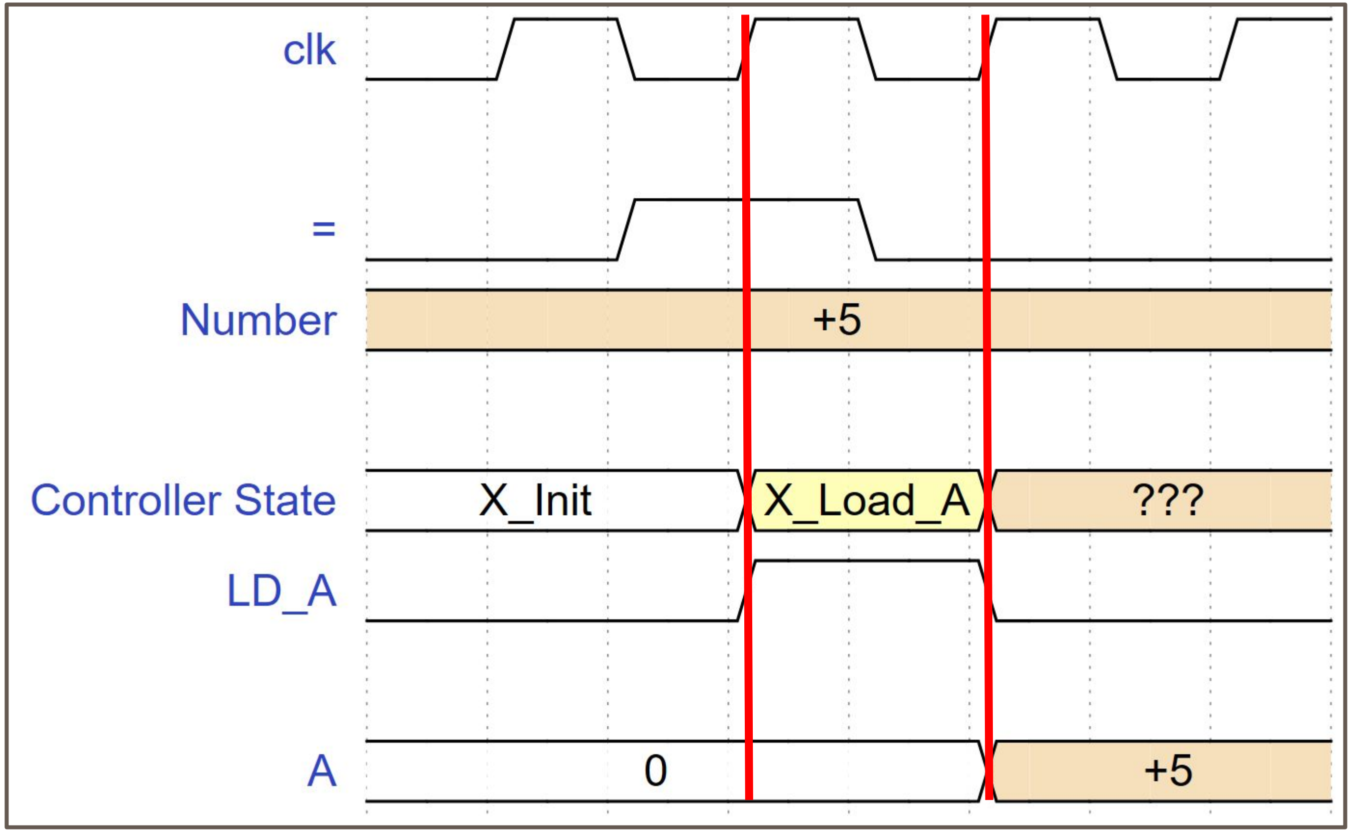
Must Finish

Calculator Outputs

```
//-----------------------------------------------------------------
// Calculator Outputs
// The two outputs are Result and Overflow, get it?
//-----------------------------------------------------------------
```



Understanding RTL Example



NOTE: Not showing N_TC register

A is loaded on the cycle AFTER the Load_A controller state

THANK YOU