Mini-Project 0

Mapping of 552 Processor

Mini Project 0

- Done as teams of 2 or 3
- Map either your own ECE552 or Eric's ECE552 pipelined processor to the DE1-SOC FPGA board
  - Map memories to internal block SRAMs
    - RF mapping is optional (but recommended)
      - May have to add bypassing
    - 16k x 16 Instruction memory
    - 8k x 16 data memory
  - Memory Map LEDs and switches to address space
  - Write ASM code to test & demo

Mini Project 0 Objectives

- To get familiar with the lab environment prior to the class project
- To get practice using verilog in your designs
- To provide a base processor you can expand for the project
- To get familiar with potential project partners

Block SRAMs

- Can be single port read/write or dual port 1-read, 1-write
- Must have synchronous read & write operation
  - Can be **posedge** or **negedge** but both read and write have to be same edge
  - This could cause major issues if your design had a purely combinational read (latch based level sensitive read/write).
    - May have to adjust timing of your pipeline

- Can be inferred with verilog (don't have to instantiate IP blocks)
  - I will deduct points if you do it with IP blocks. Why make it difficult. IP blocks don't natively simulate like pure verilog.

Block SRAMs

- Example verilog for single port memory:

```
module DM16kx16(clk,addr,re,we,wrt_data,rd_data);

////////////////////////////////////////////////
// Data memory.  Single ported, can read or //
// write but not both in a single cycle.   //
////////////////////////////////////////////////
input clk;
input [13:0] addr;
input re;                // asserted when instruction read desired
input we;                // asserted when write desired
input [15:0] wrt_data;   // data to be written

output reg [15:0] rd_data;  //output of data memory

reg [15:0]data_mem[0:16535];

////////////////////////////////////////////
// Read is synchronous on negedge clk //
////////////////////////////////////////////
always @(negedge clk)
  if (re && ~we)
    rd_data <= data_mem[addr];

////////////////////////////////////////////////////
// Model write, data is written on clock fall //
////////////////////////////////////////////////////
always @(negedge clk)
  if (we && ~re)
    data_mem[addr] <= wrt_data;

endmodule
```

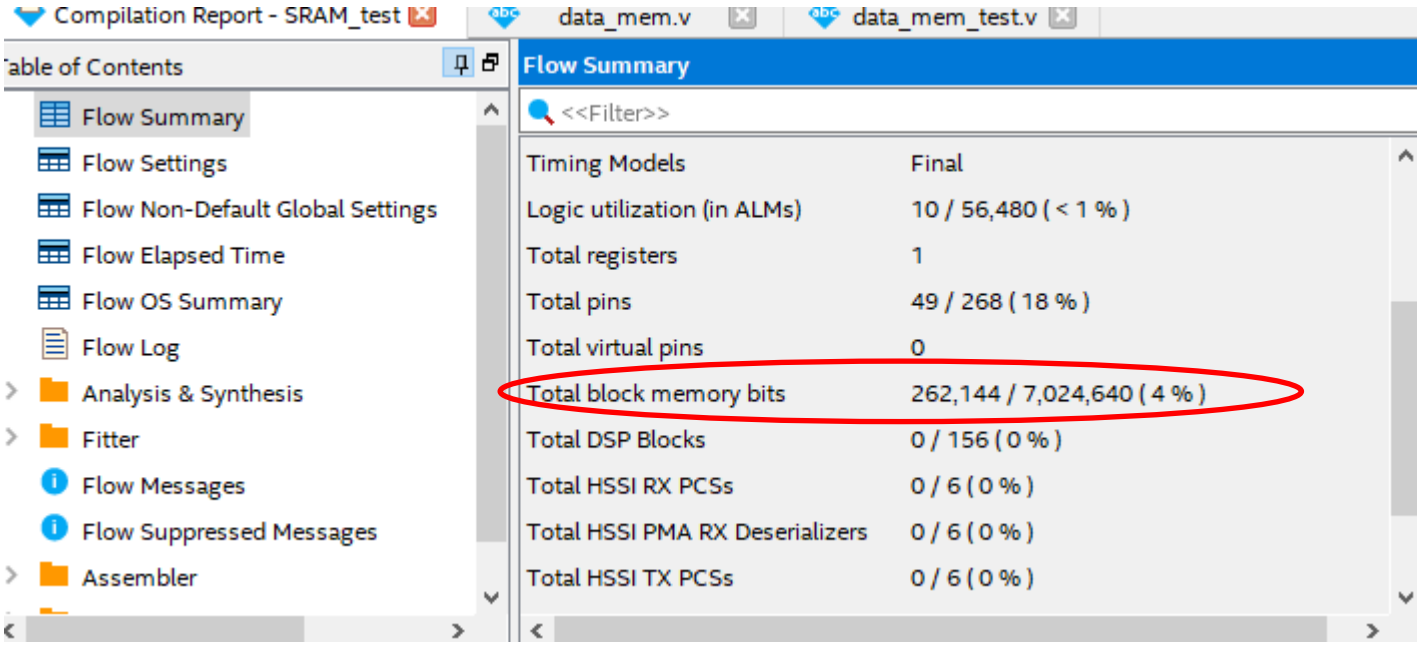
Block SRAMs

- Example verilog for Dual Port memory

```
module dualPort1024x16(clk,we,waddr,raddr,wdata,rdata);

  input clk;                  // RAM clock.
  input we;                   // active high write enable
  input [9:0] waddr;          // 10-bit write enable (0 thru 1023)
  input [9:0] raddr;          // 10-bit read enable (0 thru 1023)
  input [15:0] wdata;         // data to write
  output reg [15:0] rdata;    // data being read

  reg [15:0] mem [1023:0];

  always @(posedge clk) begin
    if (we)
      mem[waddr] <= wdata;
    rdata <= mem[raddr];
  end

endmodule
```

Block SRAMs

- How do you know when you are inferring block memory vs a bunch of flops and logic?



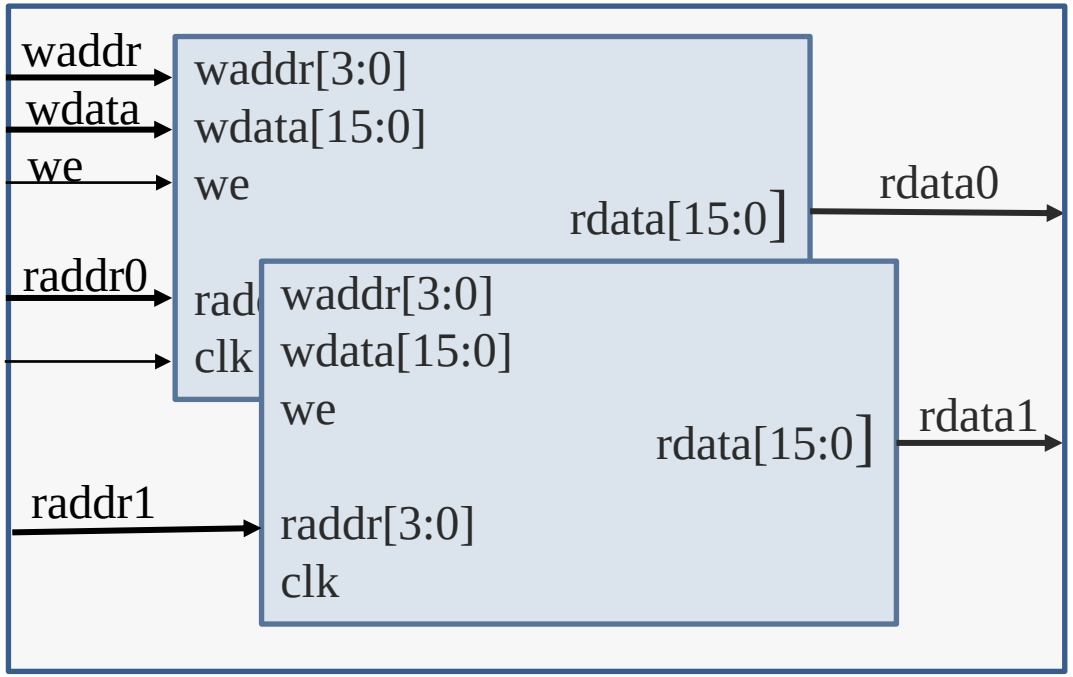
Block SRAMs

- How do I initialize my instruction memory with code?

```
module IM(clk,addr,instr);

input clk;
input [12:0] addr;

output reg [15:0] instr;     //output of insturction memory

reg [15:0]instr_mem[0:2047];

//////////////////////////////
// Synch read on clock rise //
//////////////////////////////
always @(negedge clk)
    instr <= instr_mem[addr];

initial begin
  $readmemh("C:/Users/EricHoffman/Documents/DE1_SoC/Tools/Systen
end

endmodule
```

Block SRAMs…Making RF

- How do make a 2-Read, 1-Write RF if only have Dual Port?



- Use 2 dual port RAMs and write same thing to both
  - A little wasteful, but still uses less total resources than if implemented with CLBs.

Memory Mapped I/O

- Your 552 processor probably did not have much interface to outside world.

- You will need to bring out **addr**, **we**, **wdata**, **re**, and **rdata** of your processor as I/O. *(The data memory interface)*
- Be careful to fully qualify external **we** and **re**.

```
assign mm_re = |dst_EX_DM[15:13] & dm_re_EX_DM;  // External and a read
assign mm_we = |dst_EX_DM[15:13] & dm_we_EX_DM;  // External and a write
```

- Internal data memory write too!

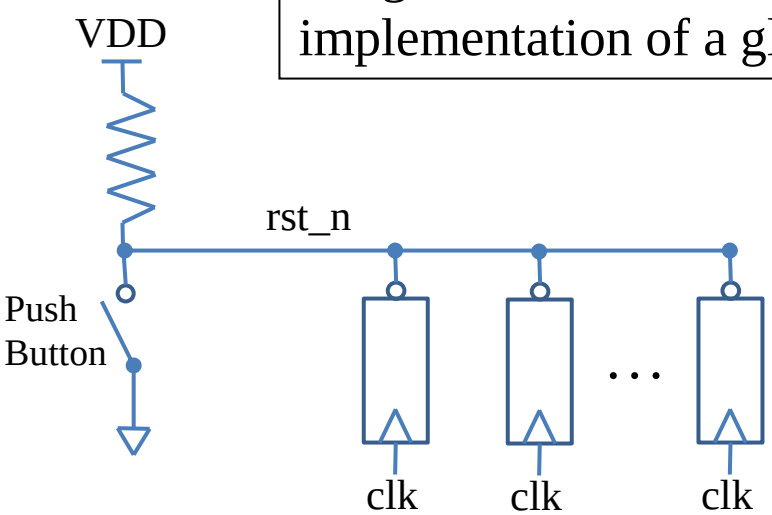
```
assign DM_we = ~|dst_EX_DM[15:13] & dm_we_EX_DM;    // qualified internal DM we
```

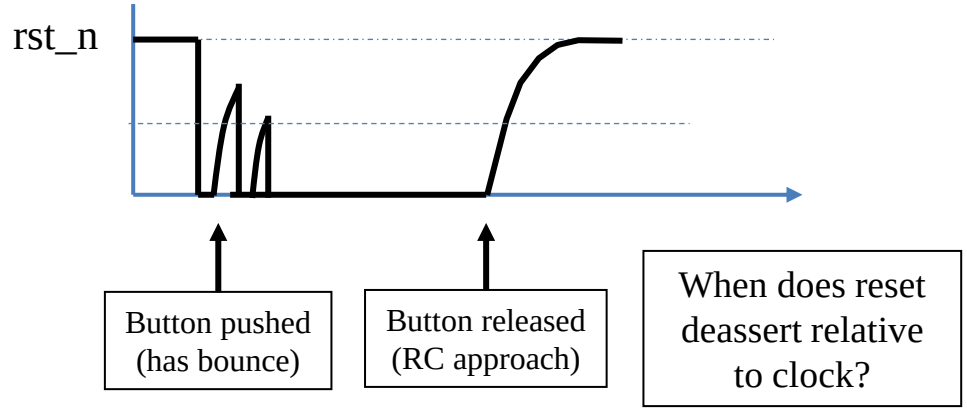
Address:	Description:
0xC000	Write to this address will write to LEDR[9:0] of board
0xC001	Read from this address will return state of SW[9:0] of board

Reset Synch

- On the FPGA board we have push buttons. We will use one as the source for our asynchronous reset.
- It is simply a momentary push button switch to ground with a pull-up resistor.

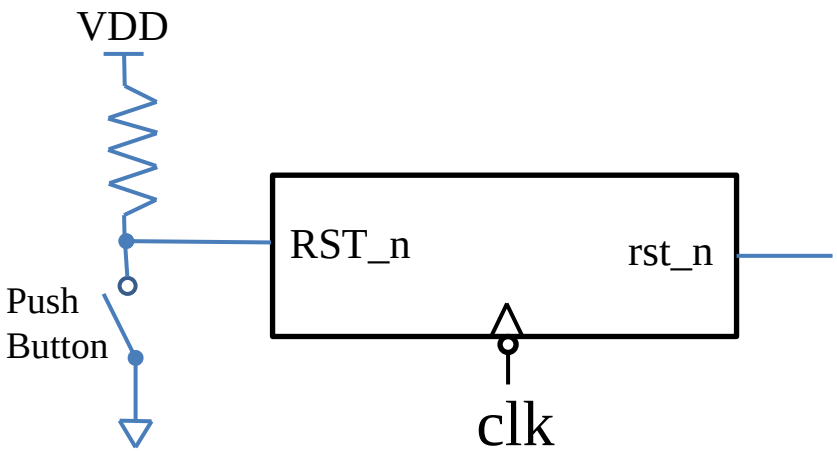
Imagine what a disaster this implementation of a global reset would be!!





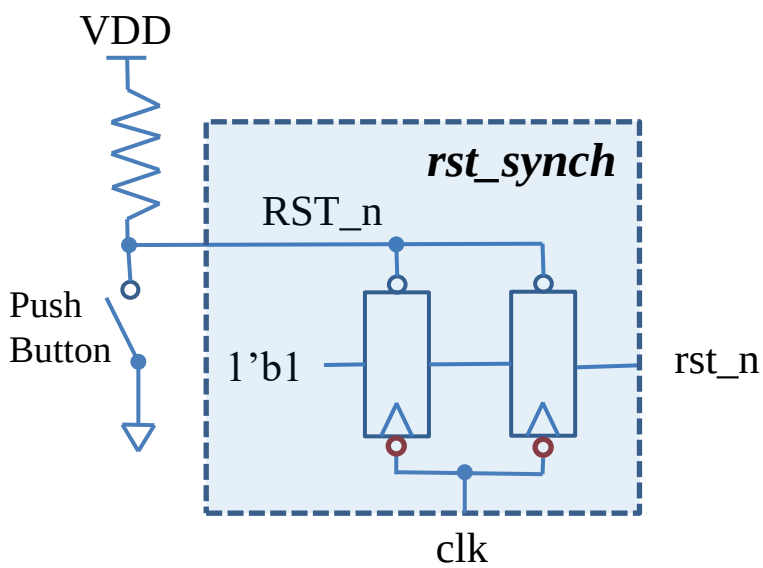
Remember…you want your reset de-asserted on the opposite edge of clock that your other flops are active on. This means we want our reset to de-assert (rise) on negative edge of clock.

Reset Synch



We want to build a reset synchronizer that takes in the raw push button signal and creates a signal that is deasserted at the negative edge of clock.

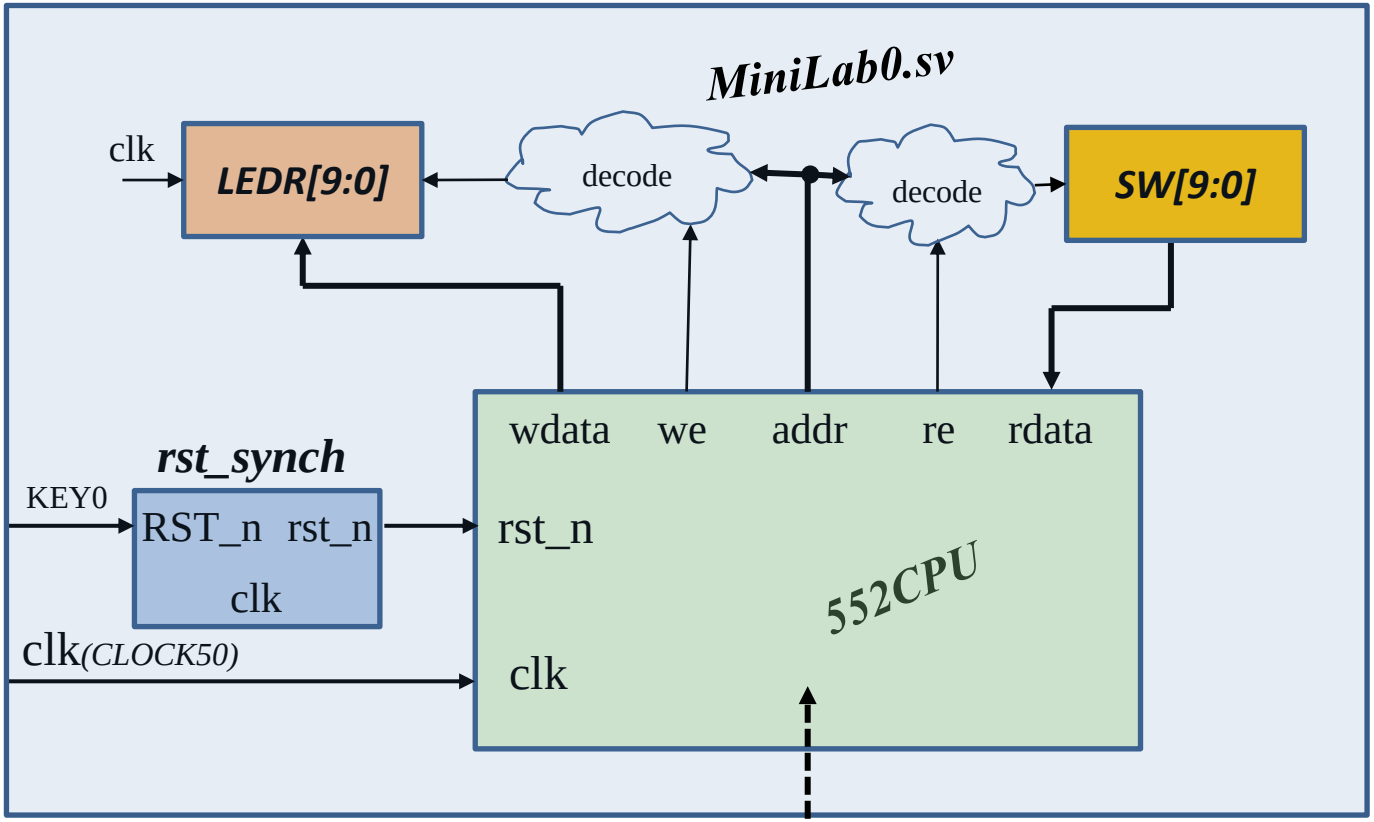
It will have an interface of:
RST_n = raw input from push button
clk = clock, and we use negative edge
rst_n = our syncrhronized output which will form the global reset to the rest of our chip.



Push of the button will asynch reset the two flops. When button is released we have a double flopping (metastability reasons) to produce our global **rst_n**. The flops are negative edge triggered so our global reset will deassert on the opposite edge of all our other flops.

Code this reset synch unit (**rst_synch.sv**)

TopLevel



ASM code can be as simple as continually reading the switches and displaying the value on the LEDs

Code to demo (from .asm)

Miniproject 0 Report

- Verilog code for your design with clear comments. This includes **MiniLab0_tb.sv** as well as **MiniLab0.sv** and **all** its children
- ASM code used to test
- Problems encountered and solutions employed
- Your 2/3-person team must demo to Eric or Tananun.