## SUMMARY

The semiconductor industry responded to customer demands in the 1980s for a configurable logic circuit that could be used to rapidly implement high-speed logic circuitry yet be reconfigurable as the design matures or system specifications change. These **complex programmable logic devices** (CPLDs) and **field-programmable gate arrays** (FPGAs) use tens of thousands of configurable logic blocks, memories, and I/O pins to provide a rich fabric for engineers to design complex systems. The configurable logic blocks usually include a multiplexer-based truth table (a look-up table) with configurable multiplexer data inputs, a D-type flip-flop, and support logic to enable interconnection to other logic blocks. Modern design automation tools allow engineers to use schematics and hardware description languages to describe designs. Synthesis tools evaluate designs for errors, create logic equations, and compile wiring configuration files used to store configuration binary patterns in the CPLD or FPGA configuration memory. The Intel MAX10 FPGA **logic element (LE)** is shown in Figure 1.

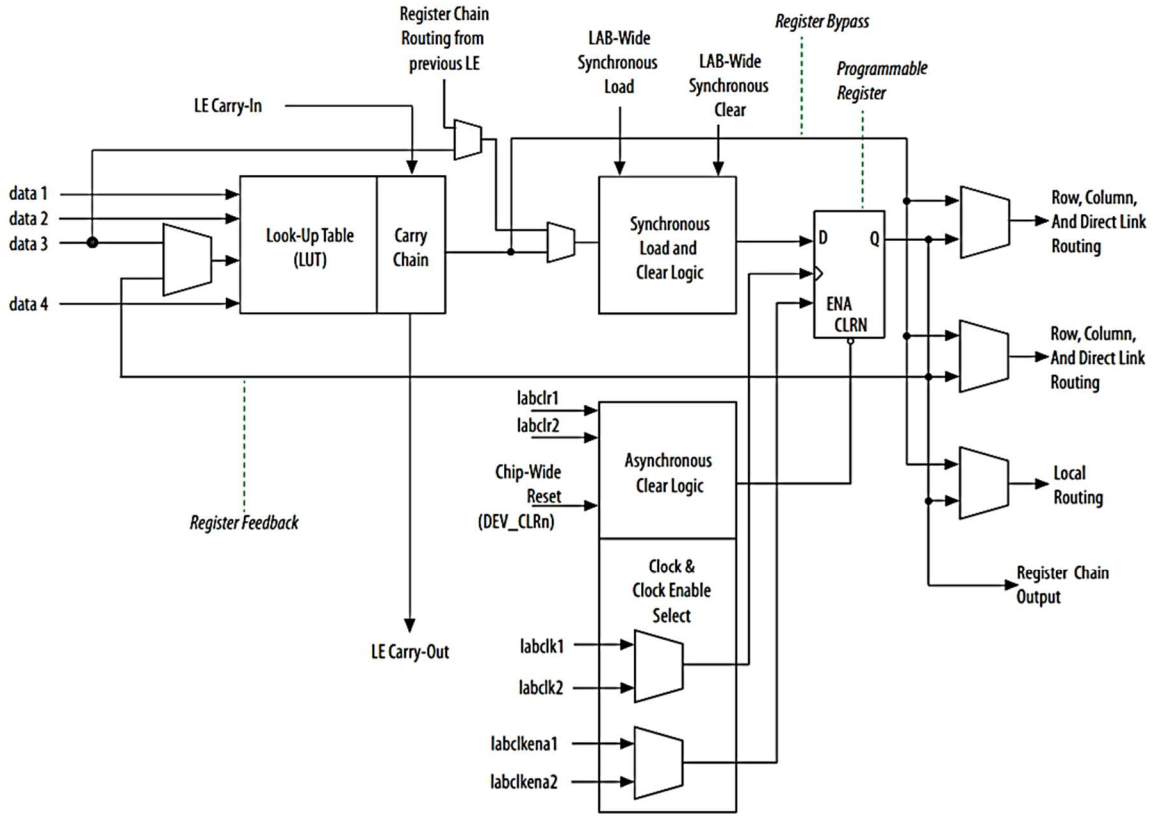


Figure 1: The MAX10 Configurable Logic Element

Intel markets the MAX10 as a set of chips with differing amounts of configurable logic elements. Each chip begins with 10MXX, where XX represents the number of logic elements in thousands. The current MSOE laboratory board uses a 10M50 device with 50,000 logic elements. In this laboratory, students implement a system level computer architecture with an ARMv4 single-cycle processor, memory-mapped I/O devices, and a reset synchronizer. This computer is configured in the Intel 10M50DAF484C7G FPGA on the DE10-Lite laboratory board.

## PROJECT MANAGEMENT

This project should be completed with the CE1921 basic single-cycle processor. **Ensure** you have a backup archive of your working single-cycle processor before beginning the modifications required for this laboratory. **Use** Project → Archive to make a backup archive.

Read this entire document before beginning your work. The reading will help you understand the changes you will make as you progress up the grading scale. **Read carefully and when you reread and begin your work, work slowly**.

## SYSTEM ARCHITECTURE

All computers contain the five components listed by John Von Neumann in section two of the classic paper he wrote in 1945 entitled ***First Draft of a Report on the EDVAC***. These five components are **input, output, arithmetic circuits, memory,** and **control circuits**. They are generally organized into the classic system architecture shown in Figure 2.

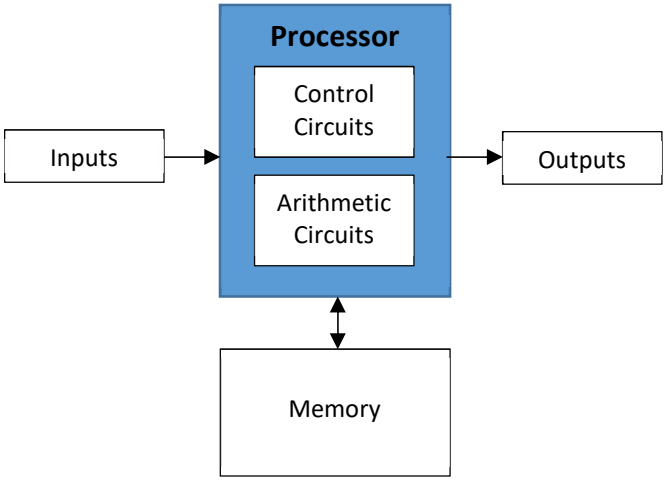


Figure 2: The Basic Organization of a Stored-Program Computer

In this model, computer instructions are stored as binary numbers in memory. The processor fetches the next instruction from memory, retrieves stored numbers as data, completes the calculation, and stores the calculated result back to memory.

- The **Princeton organization** shares one memory for instructions and data through a single connection to the processor. This single connection prevents simultaneous instruction and data movement. This bottleneck limits performance. It is an optimization of circuit size over speed.
- The **Harvard organization** uses two separate memories to enable simultaneous instruction and data movement and thus improved performance. This is an optimization of speed over size.

Inputs and outputs are peripheral devices that provide ways to interact with users. Input devices and output devices are accessed using one of two standard methods.

- A **memory-mapped** I/O device appears as one or more memory locations in the computer memory. These memory locations correspond to control, data, and status registers in the I/O device. Access to these registers is made through the standard load and store instructions. The system architecture contains an address decoder module that monitors the memory address and asserts the appropriate control signals on the registers of the I/O device whenever the device is accessed using a load or store instruction. The ARM instruction set specifies that I/O devices must be memory-mapped. Accessing I/O devices exactly like memory keeps to the RISC philosophy of **regularity implies a simple and smaller design**.
- A **port-mapped** I/O device is accessed using special I/O instructions defined in the instruction set. The device is accessed using these special input and output instructions. The Intel x86 instruction set includes the **in** and **out** instructions. Added instructions increase instruction set complexity.

In this laboratory, two output devices and one input device are added to the ARM single-cycle processor. This system architecture is then configured in the MAX10 FPGA on the DE10-Lite laboratory board. The system architecture is shown in Figure 3 and the memory and I/O devices are document in the memory-map of Table 1.

*Table 1: The CE1921 ARM Computer Devices*

| DEVICE | TYPE | MEMORY ADDRESS | BEHAVIOR |
|---|---|---|---|
| SEG7 | output | 0x000000FC | Displays the lower five nibbles of its 32-bit wide data register. |
| LED | output | 0x000000F8 | Displays the lower ten bits of its 32-bit data register. |
| SLIDER | input | 0x000000F4 | Provides the lower ten bits of its 32-bit wide data register. |
| MAIN MEMORY | mem | 0x00000000 – 0x0000001F | A 32-location main memory used for general-purpose data storage. The size is kept small to enable faster simulation. |

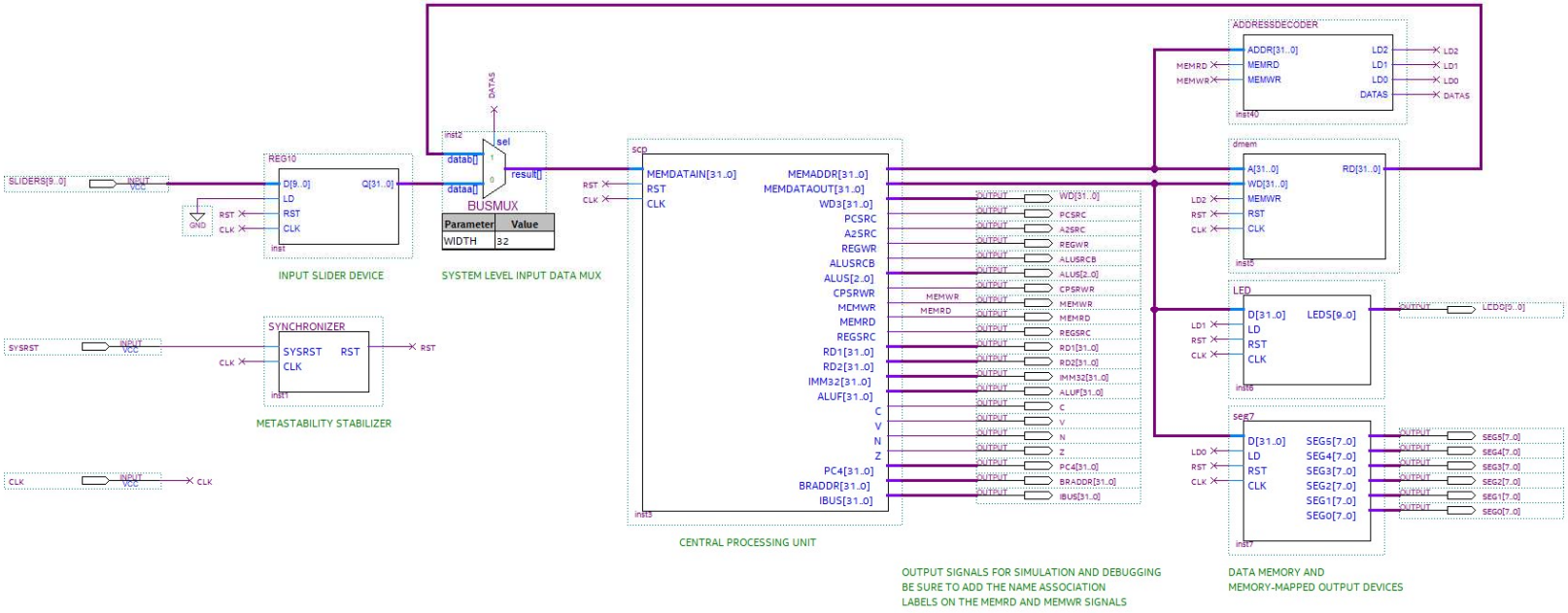


Figure 3: The CE1921 DE10-Lite Computer

## MODIFYING THE PROCESSOR

The basic CE1921 single-cycle processor included a memory circuit stage. Adding memory-mapped I/O devices requires generalizing the circuit to allow additional memory-mapped components to provide data to the REGSRC multiplexer through the same multiplexer path that the DMEM was connected to. All laboratory exercises are modifications and additions within your single-cycle processor project.

1. **Replace** the data memory component in your single-cycle processor with a memory address output bus, a memory data output bus, and a memory data input bus as shown in Figure 4. **Note** all you are doing is removing DMEM and adding these busses as shown between the execute and WB mux. Green text can be added using the schematic editor text toolbar icon.

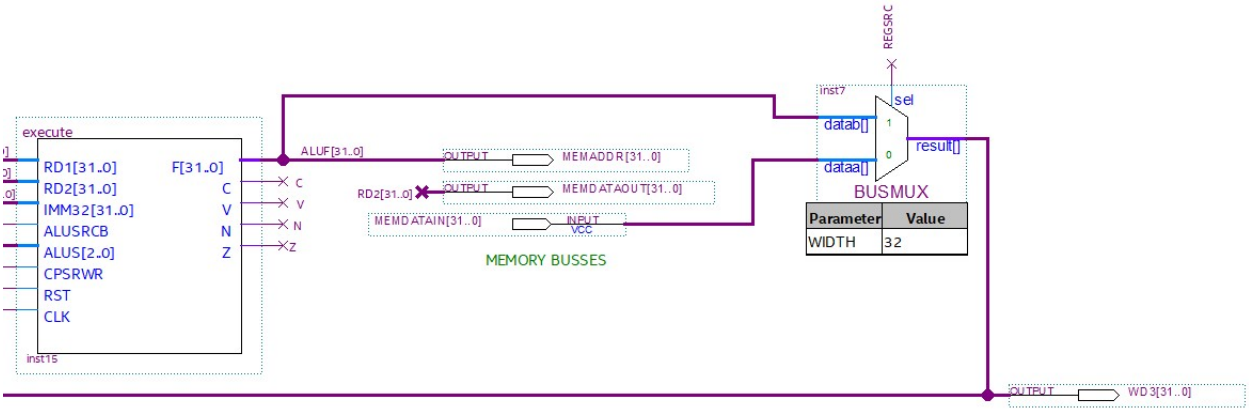


Figure 4: Memory busses replace the memory circuit in the single-cycle processor

2. **Add** an active-low MEMRD output to the controller. This signal activates when executing LDR.
3. **Use** File → Create/Update → Create Symbol File to update the controller component symbol.
4. **Replace** the old controller symbol with the new one in the top-level diagram.
5. **Add** processor outputs for all control signals, the ALU status signals, and the fetch stage signals as shown in Figure 5. These outputs will facilitate both simulation and FPGA debugging.

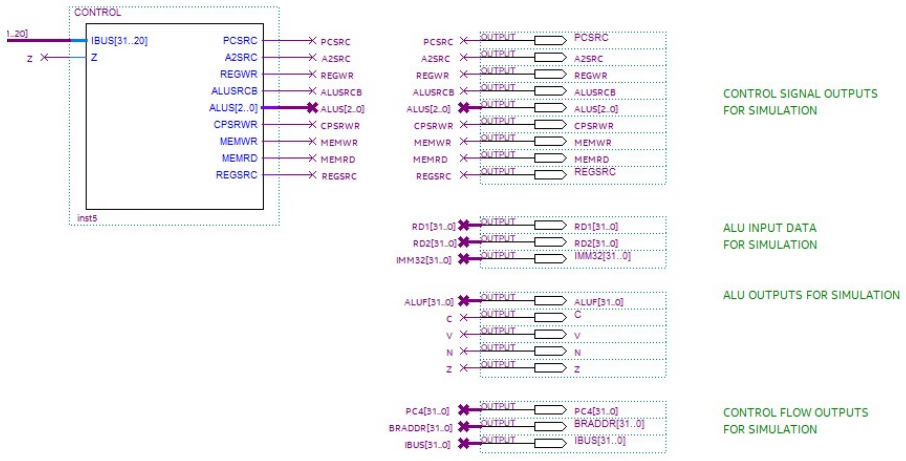


Figure 5: The Processor Signal Outputs

6. **Create** a new component symbol for the single-cycle processor using File → Create/Update → Create Symbol File. This component will be integrated into a system level diagram. The component should look something like Figure 6 but the signals may be in different order based on how you placed inputs and outputs in your schematic diagram.

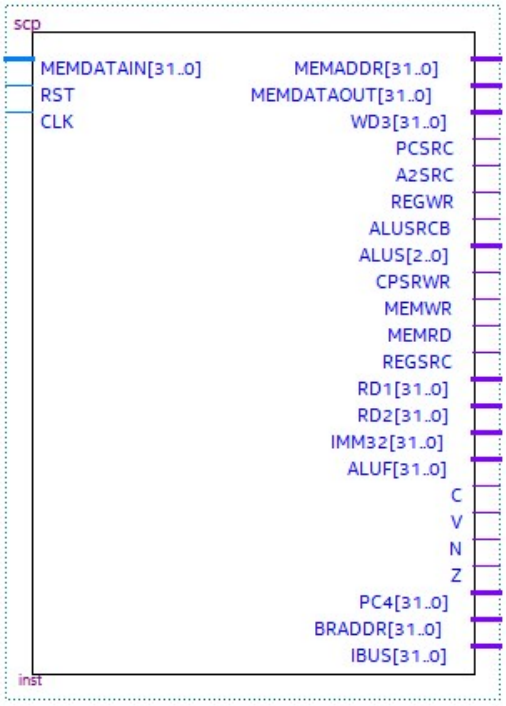


Figure 6: The Single-Cycle Processor Symbol

7. Convert all registers to **asynchronous *active-low* reset** and **rising-edge** sample. This change impacts the PC register, each register in the register file, and the current program status register. Synchronous reset depends on the clock edge. But, what happens to the computer if the clock fails? What happens immediately after power-up as the clock circuitry is stabilizing its outputs? Asynchronous reset allows the processor to reset to a known starting state if the clock fails. It also allows the processor to be held in reset by other system level components as the processor clock circuitry stabilizes after power-up. Figure 7 shows example of the required changes to **each** VHDL file that contains a clock. Signal names may differ and of course the reset and sample behaviors may vary across your files. Also remember that you are not adding any new inputs or outputs to the VHDL files. Thus, you do not need to update component symbols or rewire schematics. You are only changing the reset behavior of already existing components.

**PROGRAM COUNTER REGISTER CHANGE**

```
reg: process(LD, RST, CLK)
begin
  if RST='0' then Q <= X"00000000"; -- asynchronous
  elsif rising_edge(CLK) then -- clock edge
    if LD='0' then Q<=D; -- synchronous load
    end if ;
  end if;
end process reg;
```

**REGISTER FILE CHANGES**

```
reg0: process(rst,clk)
begin
  if RST='0' then R0 <=X"00000000";
  elsif rising_edge(clk) then
    if REGWR = '0' then
      if A3 = B"0000" then R0 <= WD3;
      end if;
    end if;
  end if;
end process;
```

Figure 7: Asynchronous Reset Logic Added to PC and R0 in the Register File as Examples

## SEVEN-SEGMENT OUTPUT DEVICE (SEG7DECODE.VHD, SEG7.BDF)

1. **Complete** the provided skeleton VHDL code for a seven-segment display decoder that generates six output busses. Each output bus carries the voltages to turn on the display LEDs of one seven-segment display. The decoder accepts a 32-bit input bus but only displays the lower six **hex** nibbles of the eight total nibbles because the DE10-Lite FPGA board has six seven-segment displays. The DE10-Lite has seven-segment LEDs that are active-low. Each output bus should correspond to its similarly named LED in Figure 8. The DP LED is bit 7 of the bus. **Use** the component shown in Figure 9 to guide the entity description. When finished, create the symbol file using File → Create/Update → Create Symbol File.


Figure 8: The DE10-Lite 7-segment LED arrangement

2. **Create** the SEG7 output device shown in Figure 9 as a new schematic block diagram file within your project. **Use** the seven-segment decoder and a 32-bit register with **active-low asynchronous reset**, **active-low synchronous load**, and **rising-edge** sample. You will need to write this VHDL REG32 component. It is identical to the PC register you have already created but should be a new component with the proper name. This data register will be memory-mapped to address 0x000000FC as shown in Table 1. The mapping will be done in a later component and does not impact this stage of the design. If desired, the green text label can be added to your schematic using the text icon from the schematic editor toolbar. **Name the schematic diagram SEG7** and then **create** the symbol file (File → Create/Update …) for use in other schematics. The design is shown in Figure 9.

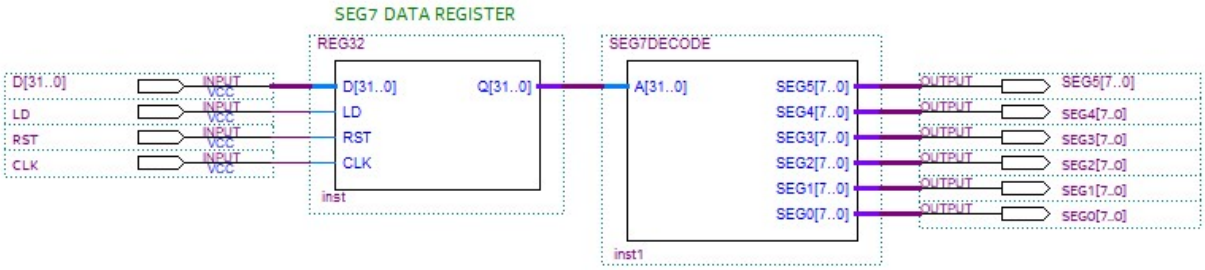


Figure 9: The SEG7 Output Device with Decoder and Data Register

## LED OUTPUT DEVICE (LED.BDF)

1. **Create** the LED output device as a schematic block diagram shown in Figure 10 using the 32-bit register component. This data register will be memory-mapped to address 0x000000F8 as shown in Table 1. The mapping will be done in a later component and does not impact this state of the design. **Name the schematic diagram LED** and then **create** the symbol file (File → Create/Update …) for use in other schematics. The design is shown in Figure 10. Only the lower ten bits of the 32-bit number stored to the LED data register are used because the DE10-Lite only contains ten LEDs.

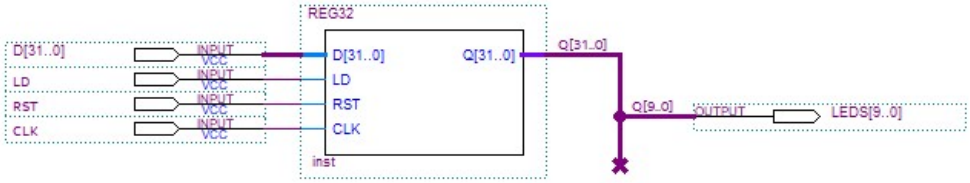


Figure 10: The LED Output Device

## ADDRESS DECODER (ADDRESSDECODER.VHD)

**Complete** the skeleton VHDL code for the system architecture address decoder and then **use** File → Create/Update → Create Symbol Files to make its schematic block diagram symbol. This component requires the processor address bus, the processor MEMRD control signal, and the processor MEMWR control signal as inputs. It outputs individual load signals for memory and the I/O device registers. It also controls the system-level data input multiplexer. The behavior is summarized in Table 2. **Examine ADDR row one**. If the memory address is in the range of the data memory and the MEMRD and MEMWR control signals show the processor is executing STR then the address decoder commands the data memory to store data by asserting LD2. **Examine ADDR row five.** If the memory address is the address of the seven-segment display data register, and the processor is executing STR then the address decoder commands the data register to store data by asserting LD0. The only time DATAS is not a don't care is when MEMRD shows the processor executing a LDR instruction. During LDR, the address decoder routes either the memory or the DE10 slider values as the processor data input.

*Table 2: The Address Decoder Truth Table*

| INPUTS | | | | OUTPUTS | | | |
|---|---|---|---|---|---|---|---|
| | | | | DMEM | LED | SEG7 | |
| ADDR | MEMRD | MEMWR | ACTION | LD2 | LD1 | LD0 | DATAS |
| 0x00000000 – 0x0000001F | 1 | 0 | STR TO MEM | 0 | 1 | 1 | - |
| 0x00000000 – 0x0000001F | 0 | 1 | LDR FROM MEM | 1 | 1 | 1 | 1 |
| 0x000000F4 | 0 | 1 | LDR FROM SLIDERS | 1 | 1 | 1 | 0 |
| 0x000000F8 | 1 | 0 | STR TO LED | 1 | 0 | 1 | - |
| 0x000000FC | 1 | 0 | STR TO SEG7 | 1 | 1 | 0 | - |
| INVALID ACCESS | 1 | 1 | NOT ALLOWED | 1 | 1 | 1 | - |

## RESET METASTABILITY SYNCHRONIZER (SYNCHRONIZER.BDF)

All register components were converted to asynchronous reset earlier in the laboratory so that the computer can be reset even if the clock has stopped due to a failure, power-on stabilization, or for testing. Unfortunately, asynchronous signals, such as that from a reset pushbutton, can violate the setup and hold times of registers because they occur too close to the edge of the register sample clock. This leads to a condition within the D flip-flop circuit known as ***metastability***. During metastability, the flip-flop output did not have enough time to sample the input voltage before being commanded to stop by the clock edge. As a result, the output may be the incorrect value.

In this laboratory, a pushbutton from the DE10-Lite board provides the system reset signal. The DE10-Lite pushbuttons provide logic-0 when pushed. The computer will reset when the user pushes this button because the registers were converted to asynchronous active-low reset. But, when the user releases the button, the logic-1 may occur too close to the clock edge and cause metastable behavior. A synchronizer can help realign the pushbutton release so that it does not violate the setup and hold times. The circuit is shown in Figure 11.

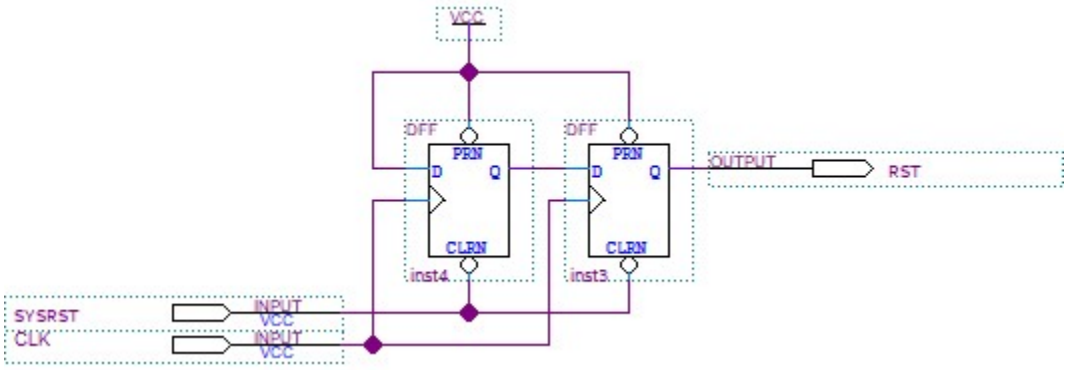


Figure 11: The Reset Pushbutton Synchronizer

The DFF components in Figure 11 have asynchronous reset behavior. Thus, a SYSRST pushbutton press will result in a near instantaneous logic-0 hitting the RST signal of the processor and I/O device registers. When the button is released, the first DFF may be left in a metastable state but the second should correctly sample the logic-0 cleared in the first flip-flop. The hope is that on the next clock edge, the first DFF will correctly sample the logic-1 provided by VCC. The second DFF will sample whatever metastable value existed in the first DFF. This could have been a logic-1 or logic-0. In the best case, it was already a logic-1 and the reset ends throughout the system. In the worst case, the metastable state was logic-0 and a third clock cycle is required to propagate the VCC logic-1 out into the system and bring the registers out of reset. **Build** this synchronizer circuit as a new schematic block diagram within your project and then **use** File → Create/Update → Create Symbol Files to make its schematic block diagram symbol.

## INPUT SLIDER DEVICE

**Input** is provided to the DE10-Lite Single Cycle Computer using the DE10-Lite slider switches. The switch values are sampled on each clock period and stored in a 10-bit register. **Complete** the skeleton VHDL code for a 10-bit register with **active-low asynchronous reset and active-low synchronous load**.

## UPDATING THE IROM

**Replace** the machine code in your IROM component with the new machine code **provided** as part of the lab assignment. This machine code is the result of assembling the DE10ROM1 TEST PROGRAM.

**DE10ROM1 TEST PROGRAM**

```
main:   MOV R4,#4        ; R4 = memory address
        MOV R12,#0       ; temp = 0
        STR R12,[R4]     ; MEM[4] = 0 : init memory
        MOV R12,#SLIDE   ; address of sliders (.equ SLIDE,0x000000F4)
        LDR R8,[R12]     ; i = n
        MOV R9,#0        ; sum = 0
        CMP R8,#0        ; i=0?
        BEQ print        ; if yes branch to print
loop:   ADD R9,R9,R8     ; sum = sum + i
        SUB R8,R8,#1     ; i = i - 1
        CMP R8,#0
        BNE loop
if:     MOV R10,#0       ; creating FFFFFFE0
        SUB R10,R10,#32  ; 0 - 32 = -32 = FFFFFFE0
        AND R10,R9,R10
        CMP R10,#0
        BEQ else         ; if (R9 > 32) MEM[4] = 1
        MOV R10,#1       ; set the 1
else:   STR R10,[R4]     ; memory[4] = either 1 or 0
print:  MOV R12,#SEG7    ; seg7 data reg address (.equ SEG7,0x000000fc)
        STR R9,[R12]     ; seg7 = sum
        MOV 12,#LED      ; LED reg address (.equ LED,0x000000F8)
        LDR R3,[R4]      ; get stored memory value back
        STR R3,[R12]     ; leds = mem[4] : is it >32? LED0 on
done:   B done
```

## DE10-LITE COMPUTER SYSTEM

1. **Implement** the CE1921 computer from Figure 3 as a schematic block diagram file called ***system.bdf***.
2. **Set** the ***system.bdf*** file as the top-level entity by right-clicking ***system.bdf*** in project navigator.
3. **Synthesize** the design and correct any errors using the third triangle icon in the toolbar – the one with a small and gate below it. **Move** to step four after you corrected all errors.
4. **Assign** DE10-Lite chip pins to the system signals shown in the table. All other signals can be unassigned.

| SIGNAL NAME | DE10-LITE ASSIGNMENT | USER MANUAL PAGE |
|---|---|---|
| CLK | MAX10_CLK1_50 | 24 |
| SYSRST | KEY0 | 25 |
| SLIDERS[9..0] | SLIDERS[9..0] | 26 |
| LEDS[9..0] | LEDR[9..0] | 27 |
| SEG0[7..0] | HEX0[7..0] | 28 |
| SEG1[7..0] | HEX1[7..0] | 28 |
| SEG2[7..0] | HEX2[7..0] | 29 |
| SEG3[7..0] | HEX3[7..0] | 29 |
| SEG4[7..0] | HEX4[7..0] | 29 |
| SEG5[7..0] | HEX5[7..0] | 29 |

5. **Create** a new Design Constraints file. This file contains specifications that we want the design automation tool to attempt to achieve. In the laboratory, the only constraint is a design that can clock using the 50MHz DE10-Lite oscillator. This means that the final circuit must propagate voltages across the logic cells and meet the setup and hold times of registers within a 20ns period. **Use** File → New → Synopsis Design Constraints File. **Add** this statement (use copy-paste) as the only line in the file and accept the default file name on save.

   ```
   create_clock -period 20 [get_ports {clk}]
   ```

6. **Build** the design using Processing → Start Compilation. This may take up to six or seven minutes to complete. The FPGA compiler is attempting to place and route hundreds of system signals.
7. **Program** the DE10-Lite board and test your design. **Try** setting the sliders to 10. The result should be hexadecimal 37 on the seven-segment displays and a one on the LEDs. **Try** setting the sliders to fifteen. The result should be hexadecimal 78 on the seven-segment displays and a one on the LED. **Try** setting the sliders to 3. The result should be 6 on the seven-segment displays and a zero on the LEDs.

## DUE DATE

Your work must be demonstrated to the instructor no later than that last day of class. **No demonstrations will be accepted after that date**. **Submit a project archive** file to the instructor using the preferred solution method. Quartus project archive files are made using Project → Create Archive option. The file will reside in your project folder and will have the file extension .QAR.