HW VII - (3 days)

Microprocessor Design: Shifter and Program Counter (PC)

The purpose of this assignment is to design a 16-bit shifter and a program counter (PC) for your microprocessor and to test your PC on an FPGA.

Shifter

The shifter is an essential element for many microprocessor operations. It may be used to align or scale data, manipulate bits and bytes, or in an automatic or program-controlled shift-and-add multiply function. In the baseline machine, you are required to implement only a Logical Shift, which shifts data in a register (dest) as specified by a signed count operand (2's complement). A positive count specifies a left shift; a negative count specifies a right shift. Considering the shift amounts and directions will be determined by the flow of the given instruction sequence when used within a CPU later, you must support a shift amount contained in a register or the immediate field of the instruction. All bits shifted out of the destination register are lost. All destination bits not mapped from the original operand are filled with zeros. Note that shift-register is not asked here. It is a shifter and it is a combinational circuit.

This part is for your information: Other common shift functions, which you may want to add to your processor, depending upon your application, are arithmetic shifts, arithmetic shift including the carry bit, byte swap, and rotate. In right arithmetic shifts, the high-order bits are filled with the original sign bit. To shift the contents of a register one bit per clock cycle, an ordinary shift register, which can be assembled from cascaded D-latches, could be used. However if there is a need to shift data by an arbitrary number of bit positions within one clock cycle, a dedicated, programmable shifter is required. Two frequently used approaches are (i) Barrel Shifters, and (ii) Logarithmic Shifters. While the barrel shifter implements the whole shifter as an array of pass transistors or multiplexers, the logarithmic shifter uses a staged approach. In general, barrel shifters are appropriate for small shift width and logarithmic shifters are more suitable for shifters of large width. Both types of shifter can be implemented in full CMOS transmission gates or NMOS pass gates in ASIC designs. NMOS pass gate designs in ASIC can benefit from level restorers to avoid problems associated with V_t drops. Shifters should be disabled in low-power designs when they are not in use.

Example: Arithmetic vs. logical shift

Suppose we chose the number 1101 for shifting.

– Logical shift (This is baseline operation for us)

- Logical shift right by 1: results in 0110 (Shift in 0s for logical right shifts)
- Logical shift right by 2: results in 0011 (Shift in 0s for logical right shifts)
- Logical shift left by 1: results in 1010 (Shift in 0s for left shifts)
- Logical shift left by 2: results in 0100 (Shift in 0s for left shifts)

– **Arithmetic shift** (This is not a baseline operation. You may add it if you want)

- Arithmetic left shifts same as logical
- Arithmetic right shift by 1: 1110 (Repeat sign of MSB for arithmetic right shifts)
- Arithmetic right shift by 2: 1111 (Repeat sign of MSB for arithmetic right shifts)

Program Counter (PC)

All instructions start by using the contents of the program counter as the address to fetch the next instruction. The program counter stores the current instruction address and calculates the address of the next instruction. Storing the current instruction address requires nothing more than a resettable 16-bit register. However, depending on the current instruction, we must also determine whether the next instruction is the instruction that follows sequentially or the instruction resulting from a jump or a branch. Except on Jumps and Branches, the next instruction follows sequentially from the current instruction (i.e., PC ← PC + 1). The following table lists the instructions from the baseline architecture which may force an address different from the next sequential address in the PC.

INSTRUCTION	NEXT PC VALUE
Bcond disp	PC ← PC + disp (sign ext.) or PC ← PC + 1
Jcond Rdest	PC ← Rdest or PC ← PC + 1
JAL Rlink, Rdest	Rlink ← PC + 1 PC ← Rdest

To avoid confusion concerning the value of the displacement amount for branches, consider the output listing below from the assembler. Note that the displacement for the *ble* (branch if less or equal) at 0x141 is 0x0F (e.g. 0x142 + 0xF = 0x151), not 0x10! Defining the displacement in this manner makes the design of your program counter simpler since the pc has been incremented to 0x142 by the time you are ready to calculate the new branch address.

```
0140  /  0020;  #0000000000100000  ( 99)      test5   or    r0  r0
0141  /  C70F;  #1100011100001111  (100)              ble   j1
0142  /  0434;  #0000010000110100  (101)              xor   r4  r4
                #0000000101010001  (103)              .orig 0x0151
0151  /  0020;  #0000000000100000  (104)      j1      or    r0  r0
0152  /  CC0F;  #1100110000001111  (105)              blt   j2
0153  /  0434;  #0000010000110100  (106)              xor   r4  r4
```

This part is for your information: A complication is presented if the processor is pipelined. One must decide how to handle changes in program flow in general, as the change in program flow happens. Meanwhile, unless we design the control to do differently, the processor would continue to fetch and decode subsequent instructions that follow the branch or jump. Several possible solutions are:

1. Always follow a branch or jump instruction with a NOP, or with other instructions that should be executed anyway. This is done by the compiler in many RISC processors. This is the simplest solution from a hardware point of view. Often the branch delay slots can be filled with useful instructions.
2. Lock the first stages of the pipeline so that they do not continue to fetch and decode instructions following a jump or branch. This is equivalent to forcing NOPs in the hardware.

This approach adds some complexity, while reducing code size somewhat, and reducing throughput somewhat.
3. For conditional branches, guess whether program flow will be changed or not, and continue fetching and decoding instructions. This requires a fast decode (to know whether the instruction is a conditional branch or jump), a separate arithmetic unit to calculate PC values, and the ability to squash instructions in the pipe if the guess was wrong (this is found out when the instruction gets to the second stage).

We will follow option 1 in our baseline machine.
You should be able to initialize the register to a known state, though this state may not necessarily be 0x0000, depending on your application. For example, sometimes processors have operating modes such that they come out of reset fetching from internal memory in one mode and from external memory in another mode.

PC Design:
From the table above, it is clear that the next PC value to be stored could be from either an incrementer (normal), the register file (Jump), or an adder (Branch). The PC unit block shown in your baseline architecture block diagram, is assumed to include an adder. Branch and Jump instructions could alternatively use the ALU to calculate the next address. It would also be possible to use the ALU to increment the PC in the normal case, too, thereby reducing chip area, but complicating control and routing, and lengthening the critical path. The method implied by the bus interconnections in the baseline architecture block diagram is shown below.

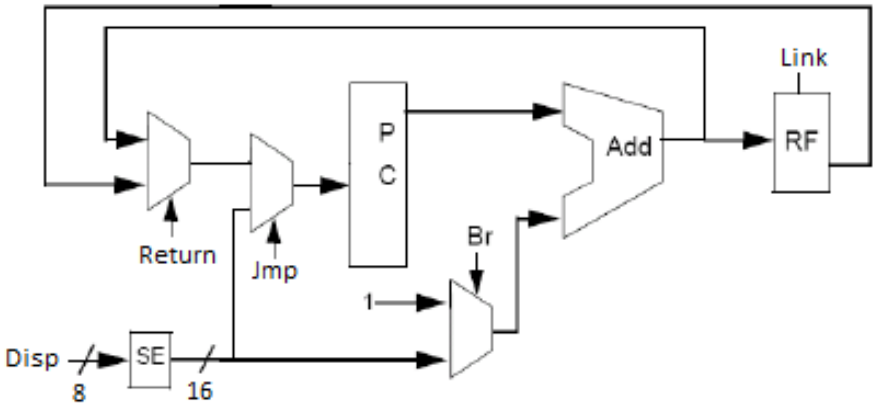


Figure 1. Program counter (PC) architecture.

Synchronous Counter Design:
You may design the PC/incrementer as a 16-bit **synchronous** up-counter with reset and parallel loading capability. In the common case, it will auto-increment every clock cycle. The parallel loading ability would be used for loading the branch/jump target calculated by the ALU. Counters are one of the simplest types of sequential circuits. A counter is usually constructed from two or more flip-flops which change state in a prescribed sequence when input pulses are received. Synchronous counters have an advantage over asynchronous counters in that the stages are clocked simultaneously and the outputs change in a synchronous manner. In the circuit below, the first stage operates as a simple divide by two stage, with Q' fed back to the D input. Subsequent stages drive Q or Q' back to D via a multiplexer. (The register bits hold their state

when Q is fed back and change state when Q' is fed back.) This change of state is enabled when all of the previous stage Q signals are true. In this example D flip-flops use differential clocks (clk and clk_bar). Multiplexers use differential selection inputs (S and S_bar). This example is for ASIC implementation. You are not supposed to use differential signals as shown below.

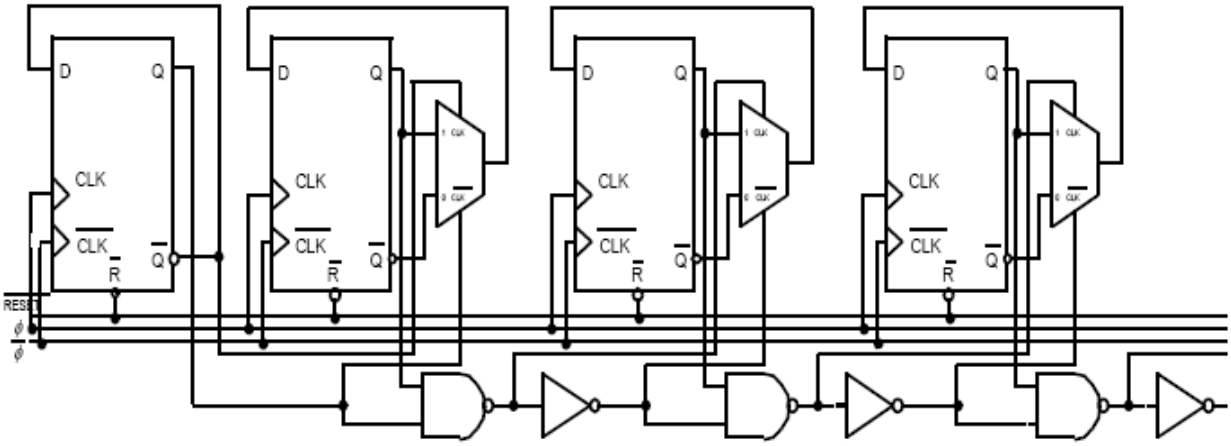


Figure 2. Synchronous counter.

Your PC must properly execute the instructions in the baseline architecture. Be sure that you can easily reset your PC, or set it to some predetermined state.
You can save routing work and chip area by putting the PC on the datapath. However, this is not necessary for this class.

FPGA Implementation of the PC

The purpose of this part of the assignment is to synthesize the PC you designed and to actually implement it on an FPGA. In this assignment you are supposed to show that you can reset the PC, do up-counting and also do parallel loading. You are going to use 8 slide switches available on the FPGA kit to enter 8-bit data for parallel loading. You will use the 5 push-buttons to reset, to start/stop counting and to do parallel loading the upper and lower bytes of the 16-bit PC content, which should be displayed on the four 7-segment displays of the Nexys3 board. Note that the 7-segment display inputs are shared among the four displays, so you need to multiplex those pins to turn on each display for a certain amount of time and move on to the next display. If each display is refreshed at a 60 Hz rate, human eye will perceive as if all four displays are illuminated simultaneously. To achieve this rate, allow 4 ms for illuminating each display, so that the entire refresh cycle will be 16 ms.

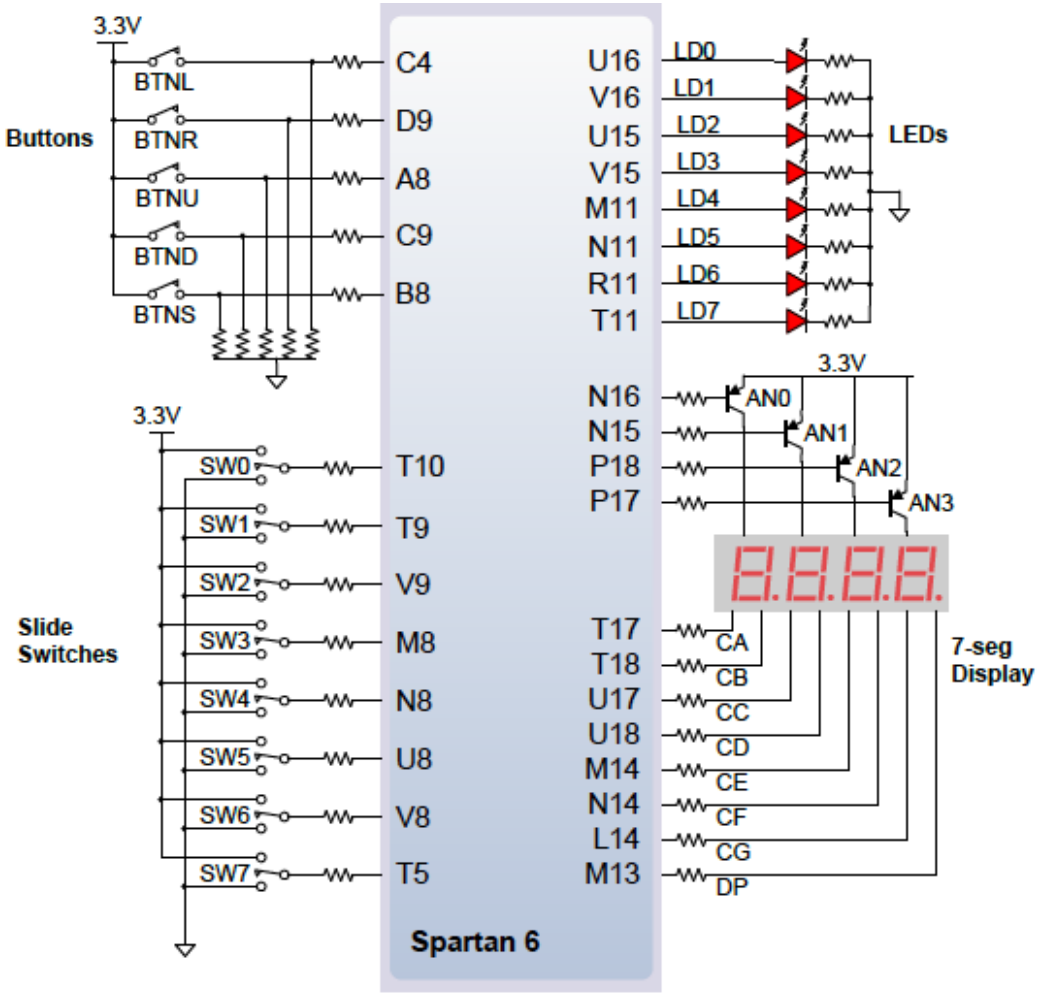


Figure 3. Basic I/O interface of the Nexys3 board.

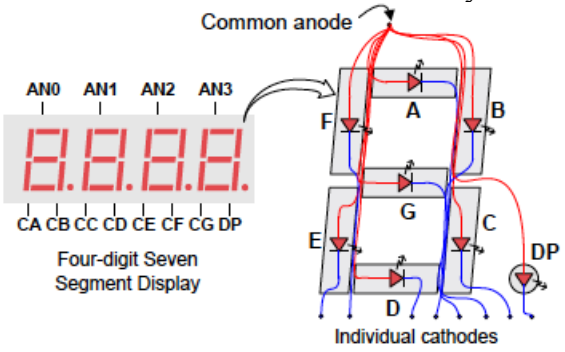


Figure 4. Driving signals for the 7-segment LED displays in the Nexys3 board.

Reading the Switches:

The FPGA is connected to five push buttons on the Nexys3 board. Each push button applies a high level to its FPGA pin when pressed and a resistor pulls the pin to a low level when the push button is released. You will use the outputs of these buttons as reset, count, stop counting and parallel load commands. If you press BTNU (upper button), your system should start counting up once every second (1 Hz). If you press BTNL (lower button), it should stop counting. If you

press BTNL and BTNR, it should do parallel loading for the upper byte (8 bits on the left) and the lower byte (8 bits on the right) of the PC respectively. Make sure that the 8 slide switches are in the correct position before pressing the button for upper or lower byte of the counter. BTNS (central button) resets the counter. However, you should condition the output of the push buttons before you use them as inputs to your system since pressing the button once actually generates many glitches. You can use a shift-register to detect if the button is pressed and then generate only one clean input signal.

The five push buttons are connected to C4, D9, A8, C9, and B8 pins of the FPGA. The module of your design should have the five push button signals as input ports. After you synthesize this design, before you transfer the code to the FPGA itself, you should map these input signals to the C4, … B8 pins of the FPGA.

In order to use the push buttons as inputs, they must be debounced so that a single push of the button results in a clean signal. One way to debounce a switch signal is to shift the bouncing signal down to a shift register. Then the outputs of all the flip-flops can be anded to generate a clean signal. The frequency of the global clock that the FPGA uses, which is generated by the Nexys3 board is 100 MHz. Assuming the bounce interval of a switch has been experimentally determined to be about 3 ms, we should design a clock divider circuit using an N-bit counter that will divide the 100 MHz clock into a clock rate suitable for switch debouncing (approximately 2^{19} times slower). Note that an n-bit counter divides the frequency of a clock signal by 2^n times. Design a switch debouncing circuit in Verilog using a 19-bit shift register and an AND gate. The output of this circuit should be input to your PC.

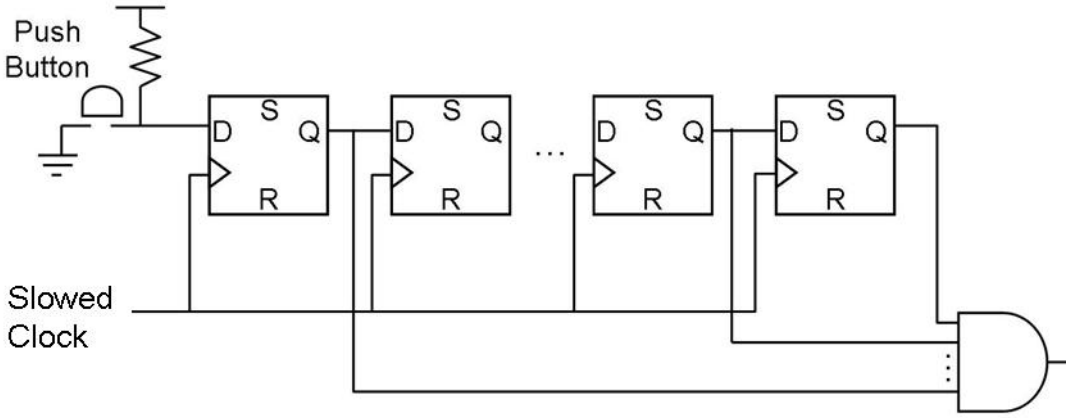
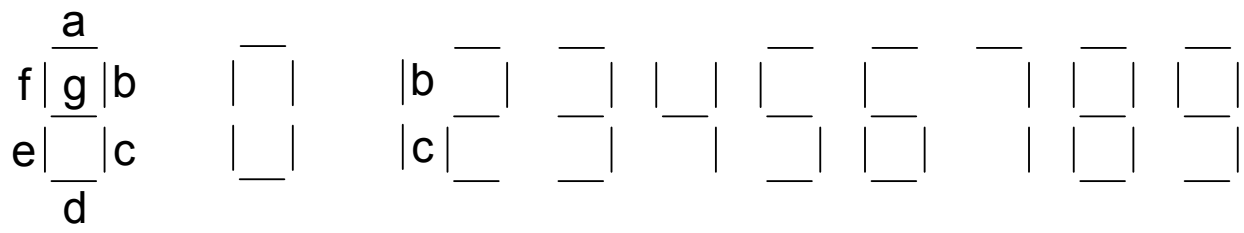


Figure 5. Debounce circuit

Also eight slide switches are attached to the FPGA. When closed (ON), each switch pulls the connected pin of the FPGA to ground. The pin is pulled high through a resistor when the switch is open (OFF). These eight switches will be used as the 8-bit input to be parallel loaded to your PC. You do not need to implement debouncer circuits for them. Refer to the Nexys3 board manual for finding the FPGA pins corresponding to each switch.

Writing the PC content to the LED Display:

We want to display the contents of the PC on the four 7-segment LED displays of the Nexys3 board. In order to do that you should write a Verilog code that does binary to 7-segment decoding. Here *D* should be a 4-bit binary input that generates a 7-bit *S* signal as the output port of your module. In this design you represent 1010 as *a*, 1011 as *b*, 1100 as *c* …etc. Furthermore, in order to visually see the counting of your PC, its clock frequency should be 1 Hz. However, the clock that the FPGA uses is 100 MHz. In your design you should implement a circuit to slow down 100 MHz clock signal to 1 Hz and this should be the clock signal of your PC.




```
module binary_to_seven_segment(input [3:0] D, output [6:0] S);
…..
```

Pin assignments

Refer to the Nexys3 Board Reference Manual for pin locations and Nexys3 Board Tutorial for preparing the .ucf file.

Assignment:

We will meet on April 27th at 12pm to grade the FPGA part of your assignment. So, do not wait until the last minute to test your design. It is recommended that you go to the laboratory before April 27th, to make sure that your synthesized design will work that day.

Shifter

- Run Isim on the 16-bit shifter and verify that it functions as expected by running a few test sequences (shift left and right by up to 15 bits).
- Use Xilinx ISE to synthesize the shifter based on Xilinx Spartan 6 family. If you write dataflow or behavioral style Verilog code, specify the components created on the schematic result of your synthesis. Explain why your code is synthesized like that. You may want to experiment on your code and see the difference in synthesis when you modify your code.

Program Counter (PC)

- Isim simulation result for the PC showing all the different possibilities for next address calculation implemented in your processor.
- Use Xilinx ISE to synthesize the PC based on Xilinx Spartan 6 family. If you write dataflow or behavioral style Verilog code, specify the components created on the schematic result of your synthesis. Explain why your code is synthesized like that. You may want to experiment on your code and see the difference in synthesis when you modify your code.

Implementation of PC on FPGA

- Design the push button debouncer circuit in Verilog.
- Design the PC_test module in Verilog that will be implemented on the FPGA kit.
- Get a synthesized Verilog file from Xilinx ISE. Make sure that this file will work in the Xilinx ISE CAD tool of the FPGA kit installed on the PCs in the Electrical Measurement and Networks Laboratory. You are strongly advised to experiment in the lab with your code before.
- Meet at the laboratory. Load your Verilog code to the computer at the laboratory. Run your code on the Xilinx ISE software. Load your design to the Nexys3 board.
- See if the LEDs read what you expect every time you press the buttons.
- Return your Verilog code for this part.