Mini-Term Project
Project Ideas[1] and Requirements

---

[1] These are just the ideas. It is not required to select projects provided below. However, any project of your choice (if it is not in the list) must be pre-approved by me (rc) and must be of adequate difficulty and/or scale.

Objectives

1. Demonstration of creativity and ingenuity in applying obtained knowledge for the solution of practical, "real-life" problems;
2. Independent research on design of logic systems;
3. Preparation of Project Documentation and Reports (that is, "Blueprints" or "Proofs");
4. Demonstration of ability to design a digital logic system, synthesize and implement it in hardware;
5. Knowledge of how to validate a behavioral system performance with simulation software (ModelSim or Xilinx ISE);
6. Demonstration of project performance on Digilent's Basys Spartan-3E FPGA Board[2]. Clever emulation of external input signals, control signals and/or disturbances is a plus but not required.

Equipment

- PC or compatible
- Digilent's Basys Spartan-3E FPGA Evaluation Board
- Signal Generators, Oscilloscopes and Digital Multimeters (if needed)

Software

- Xilinx ISE Design Software Suite 10.1
- ModelSim XE III modeling software
- Digilent's Adept ExPort Software

Parts

- Any part and components required to implement the project

[2] It is allowed to use Digilent's Nexys2 board if available and/or desired

Presentation Requirements

Project Presentation will be composed of two parts:

1. Project Demonstration[3] (***group***, in front of the class)
2. Project Defense (***individual***, in front of the instructor and the TA)

NOTE 1:

1. Note from the outset that the final write-up **is not** required for the Presentation[4] per se. However, the following items MUST be shown if requested:
2. Description of the System (that is, Specification) similar to the Lab Manual Specifications;
3. Logic Circuit Schematic (and electronic if needed) with parts used (to identify the hardware interfacing issues);

Expected Questions (during Project Defense)

- Design Choices both Hardware and Software (why this was picked over that, etc.)
- Problems encountered and how they were resolved (this is also design related)
- How can your system be improved and/or substantially modified

NOTE 2:

1. Project Demonstrations must be short and clear: the project's verbal description and a demo;
2. Project Defenses are individual, 5-10 min (not in groups)
3. Presentation quality will be critical in determining the final grade for the class

[3] Project Demonstrations must show **only** success. Translating: encountered problems and/or gross misfortunes are to be saved for the Project Defense and Final Report Conclusions.
[4] The final report **is required** and is due by the date that will be announced later

Report Format

The following information must be included to receive the highest grade

1. Detailed specification of the project[5]
   - Purpose
   - Flowchart if needed

2. System Design and Architecture[6]
   - Circuit schematic
   - State Diagrams
   - Timing diagrams
   - Flow-charts
   - etc., whatever is needed[7]

3. Detailed description of the problems and technical issues encountered especially the ones that resulted in system re-design and/or modifications

4. Conclusions[8]

   - whether the system works according to provided specifications,
   - exceptions (or special care)
   - ways to improve the system

Grading

For these projects, you will be graded on:

- your ***creativity*** in developing the project; (20%)
- circuit's behavioral functionality (testbenching); (20%)
- application functionality; (20%)
- circuit cleanliness (level of rat's nest); (20%)
- report writeup (20%)

[5] Assume this is a system-product order from a customer with a very little idea of electronics other than to Press this or that button (START/STOP?)
[6] This is a BLUEPRINT (or PROOF) of the technical system for your records, a guide during the R&D stage
[7] So that an independent reviewer can understand the structure and implementation procedures
[8] This is viewed by the customer before the final approval (or testing stage when spec's are modified)

Project 1. NETWORK ROUTER

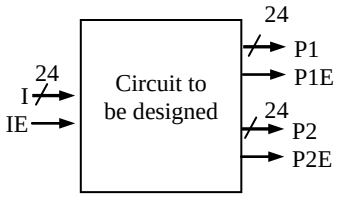
Challenge

Given a high-level state machine, implement the state machine as a datapath and a controller.

Background and Specification

The basic function of a network router chip is to receive an incoming packet, read the packet's destination address, and send that packet out over one of several ports based on the address. A packet consists of data to be sent over the network (e.g., data corresponding to part of an email message or a picture). A packet can also consist of an address, error-checking bits, and much more.

Figure P1-1. Block diagram of simplified network router system.



In this project, we'll consider a highly simplified router, whose block diagram is shown in **Figure P1-1**. The router has a 24-bit input I comprised of 3 bytes: byte I(7..0) is the packet's destination address, byte I(15..8) is the data, and byte I(23..16) is a checksum (to be explained momentarily). A 1-bit input IE pulses to 1 for one clock cycle when a valid packet arrives. The router has two 24-bit output ports, P1 and P2, each with corresponding enable ports P1E and P2E. A packet with an address less than 128 should be routed to P1, while a packet with an address of 128 or greater should be routed to P2. Routing the packet to a port requires writing the entire 24-bit packet to the port, and pulsing the corresponding enable port for one clock cycle.

The packet's sender computed the checksum byte as the sum of the other two bytes of the packet. For example, if a packet's destination address is 15 and the data is 12, the checksum should be 27 (in binary, of course). If the router detects that a packet's checksum is incorrect, then an error must have occurred during the packet's journey to the router; in which case the router should not route the packet.

Figure P1-2 provides a high-level state machine describing the router's behavior. The router waits in state WaitPkt for the arrival of a packet indicated by IE=1. The router then saves the packet in a register pkt (because the packet is only guaranteed to exist on the input I for one clock cycle), and computes the checksum in state Check1. In state Check2, the checksum has been stored in register cs, so that state ignores the packet if cs does not equal the packet's checksum byte (Note: a common error would be to place those transitions coming from state Check1, but those transitions would be reading the old value of cs, not the new value computing in that state and updated on the next clock edge. Another common error would be to read pkt in state Check1 when computing the checksum, but pkt also won't be updated until the next clock edge). If the checksum is OK, then the router proceeds to route in state Route. Transitions from that state take us to state Route1 if the address is less than 128, and to state Route2 otherwise. Those states write the packet to the appropriate port and set the appropriate enable output to 1. The router then goes back to waiting for another packet.

Figure P1-2: High-level state machine of a simplified network router system.

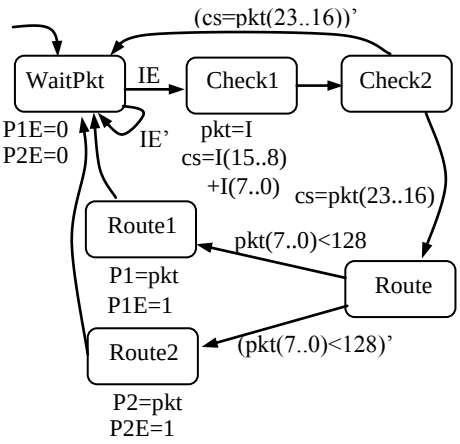


Figure represents the completion of the first step of the RTL design process. Simulate the high-level state machine at this time. Provide several packets, some with an address less than 128, some greater than or equal to 128, and watch for packets to appear on the appropriate output port. Include a packet with an incorrect checksum byte, and make sure the router does not route that packet. Question: is this router guaranteed to route every packet that arrives at its input, or might the router miss ("drop") some packets? If packets could get dropped, what is the situation that causes such drops, other than bad checksums?

The second step is to create a datapath. Thus, create a datapath able to implement the data operations and conditions in the high-level state machine of **Figure P1-2**. Notice that you'll need registers for pkt and cs, an adder, and a particular comparator, all of which you'll need to design. It's good design practice to also instantiate registers at the P1 and P2 outputs, perhaps called

P1reg and P2reg. Connect these components appropriately. Make sure to give unique names to any control signals on the datapath components.

Now, perform the third RTL design step of connecting the datapath to a controller. Be sure all inputs and outputs are connected. Next, perform the fourth RTL design step of deriving the FSM for the controller. Recall that the FSM should have the same state and transition structure of the original high-level state machine of **Figure P1-2**, but with all data operations and conditions replaced by Boolean operations and conditions that achieve those same data operations and conditions by controlling the datapath.

At this point, you should simulate your design again, which consists of a structural datapath, connected to a controller described behaviorally as an FSM. See above for simulation suggestions. Your simulation results should match those from the earlier high-level state machine simulation.

You should already know how to convert an FSM to a controller, so we will not design the internal structure of the controller in this activity.

Demonstration

Demostrate that the application performs according to specs (use emulation for incoming packets).

Project 2. STOPWATCH

Challenge

Given a high-level state machine, implement the state machine as a datapath and a controller.

Background and Specification

The watch displays the time in four decimal digits, and counts from 00.00 to 99.99 seconds and wraps around. It contains a synchronous clear signal, **RESET**, which re-sets the count to 00.00, and an enable/disable signal, **START/STOP**, which enables (starts) and suspends (stops) the counting. There is a switch control that turns the display ON/OFF such that in the OFF state the counting must be on-going and the count must be properly displayed once the switch is turned back ON.

This design is basically a BCD (binary-coded decimal) counter, which counts in BCD format. The project will involve the development of 4 counters one of which is a global mod-N counter[9] that generates a one-clock-cycle tick every 10 ms. The tick is than used to enable counting of the four digit BCD counter with the decimal point, dc, enabled always (unless in the display's OFF state).

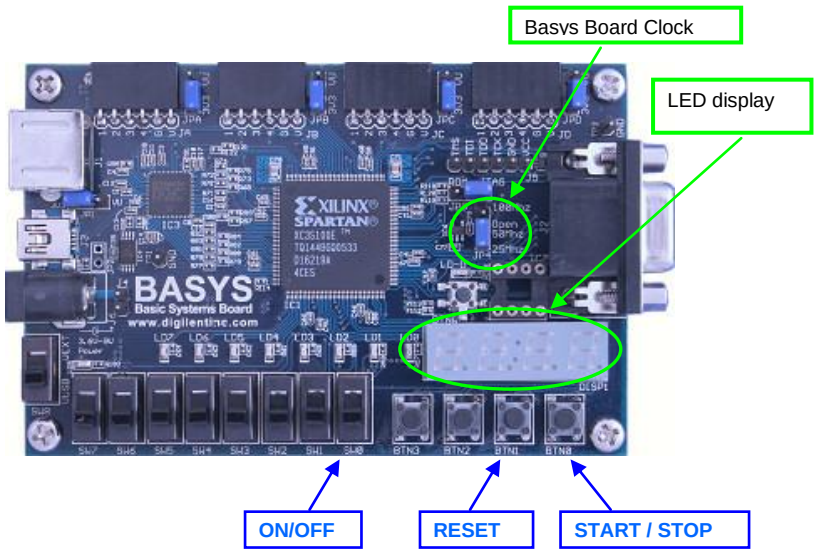


Figure P2-1. Stopwatch project Basys Board implementation

[9] The choice of N depends on the Basys board's clock frequency 25 MHz or 50 MHz whichever is selected with a jumper and must be correctly computed.

The next 3 counters are mod-10 counters with respect to the global mod-N counter developed earlier forming a cascading structure representing counts of 0.1, 1, 10 seconds respectively. The decade counters have an enable signal and generate a one-clock-cycle tick when they reach the count of 9. We can use these ticks to hook up the counters together. For example, the 10 second counter is enabled only when the enable tick of the mod-N (the first one) counter is asserted and 0.01, 0.1 and 1-second counters are 9. Observe that all system registers must be controlled by the same Basys board clock signal. **No asynchronous design is allowed**: Arithmetic type counters or T FF based counters (with EN) can be utilized as long as CLK inputs are used for CLK and not for previous (lower) clock stages. Design proper ENABLE inputs for counters if needed[10].

Demonstration

Demostrate that the application performs according to specs.

[10] This would lead to asynchronous design that must be avoided at all costs.

Project 3. PROGRAMMABLE SQUARE WAVE GENERATOR

Challenge

Given a high-level state machine, implement the state machine as a datapath and a controller.

Background and Specification

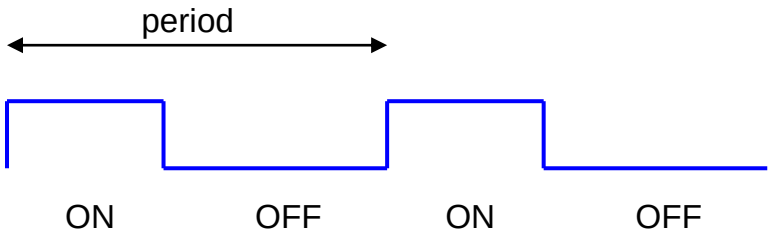


Figure P3-1. Square Wave Parameters

A programmable square wave generator is a circuit that can generate a square wave with variable ON (i.e., logic '1') and OFF (i.e., logic '0') intervals as shown in **Figure P3-1**. The duration of the intervals is specified by two 4-bit control signals, *m* and *n*, which are interpreted as unsigned integers. The ON and OFF intervals are *m**100 ns and *n**100 ns, respectively. Design must be completely synchronous. Square wave parameters are to be displayed in decimal form on the Basys board LED display.

Demonstration

Demostrate that the application performs according to specs. Use one of the Pmod output ports of the Basys board and an oscilloscope to verify the operation.

Project 4. PERIOD COUNTER

Challenge

Given a high-level state machine, implement the state machine as a datapath and a controller.

Background and Specification

A *period counter*[11] measures the period of a periodic input waveform. One way to construct the circuit is to count the number of clock cycles between two rising edges of the input signal.

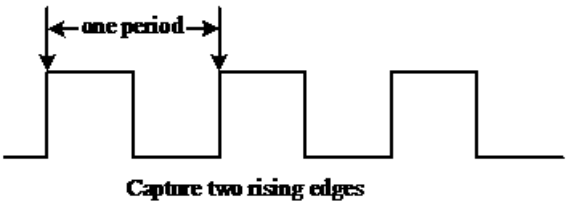


Figure P4-1. Period counter's principle of operation

Since the frequency of the system clock is known, the period of the input signal can be derived accordingly. For example, if the frequency of the system clock is f and the number of clock cycles between two rising edges is is N, the period of the input signal is $N * 1/f$.

An example design measures the period in milliseconds. The algorithm flowchart[12] of operation is shown in **Figure P4-2**. The period counter takes the measurement when START signal is asserted. Using a rising-edge detector circuit we can generate a one-clock-cycle tick, EDGE, to indicate the rising edge of the input waveform. After the START is asserted, the FSM moves to the WAIT state to wait for the first rising edge of the input. Than it moves on to the COUNT state until the next rising edge is detected. Two registers are used to keep track of the time. The *t* register counts for 50,000 clock cycles, from 0 to 49,999, and than wraps around. If the period of the system clock is 20 ns, the t register takes 1 ms

[11] another name is a "frequency counter", a misnomer actually
[12] Also called the ASM chart (algorithmic state machine), another equivalent representation of FSM.

to circulate through 50,000 cycles. The *p* register counts in terms of milliseconds. It is incremented once when the *t* register reaches 49,999.

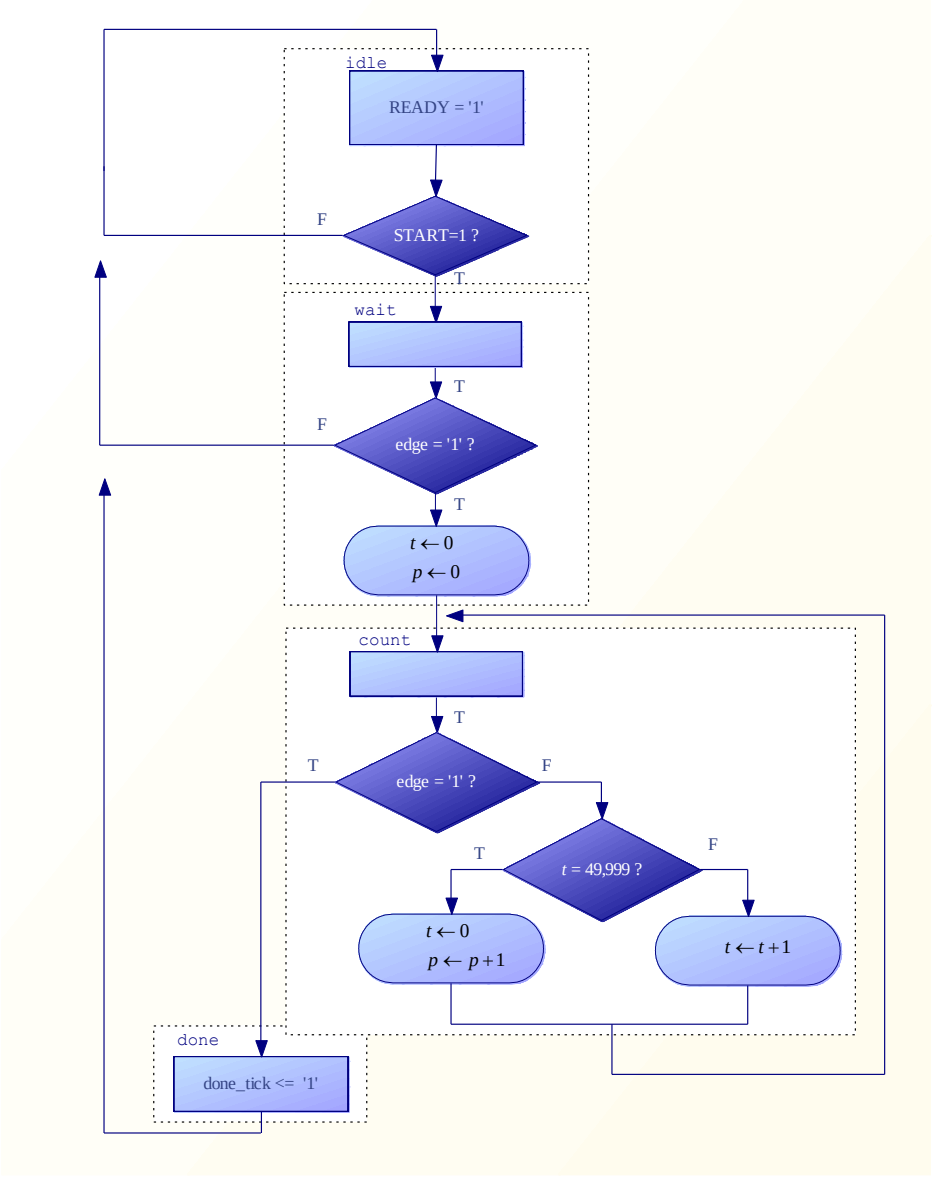


Figure P4-2. ASM chart of Period Counter operation

When the FSM exits the COUNT state, the period off the input waveform is stored in the *p* register and its unit is milliseconds. The FSM asserts the done_tick signal in the DONE state.

Demonstration

Demostrate that the application performs according to specs. Use signal generator and one of the Pmod inputs for the incoming “unknown” signal.

Project 5. TRAFFIC SIGNAL CONTROL

Challenge

Given a high-level state machine, implement the state machine as a datapath and a controller.

Background and Specification

Figure P5-1. Traffic Signals

Assume a traffic signal control system which controls the sequencing of the lights at the intersection of a busy main street and a lightly used side street. The timing requirements are illustrated in the **Figure P5-2**.

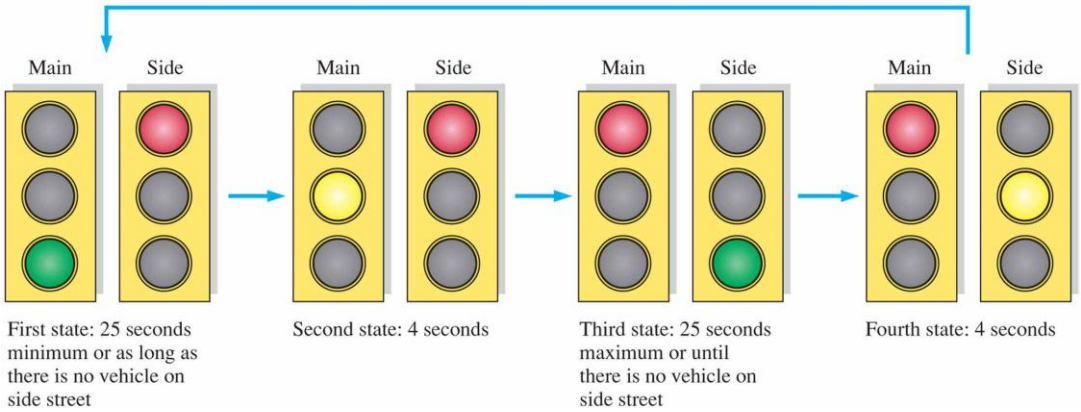


Figure P5-2. Timing requirements for for the traffic signals

From the timing requirements, a finite state machine[13] can be developed as shown in **Figure P5-3** where the variables are:

- Vs – a vehicle is present on the side street;
- TL – The 25 s timer (long timer) is ON;
- TS – The 4 s timer (short timer) is ON

Observe that the system contains two timer that must be properly triggered. Block diagram of such a system is shown in **Figure P5-4**.

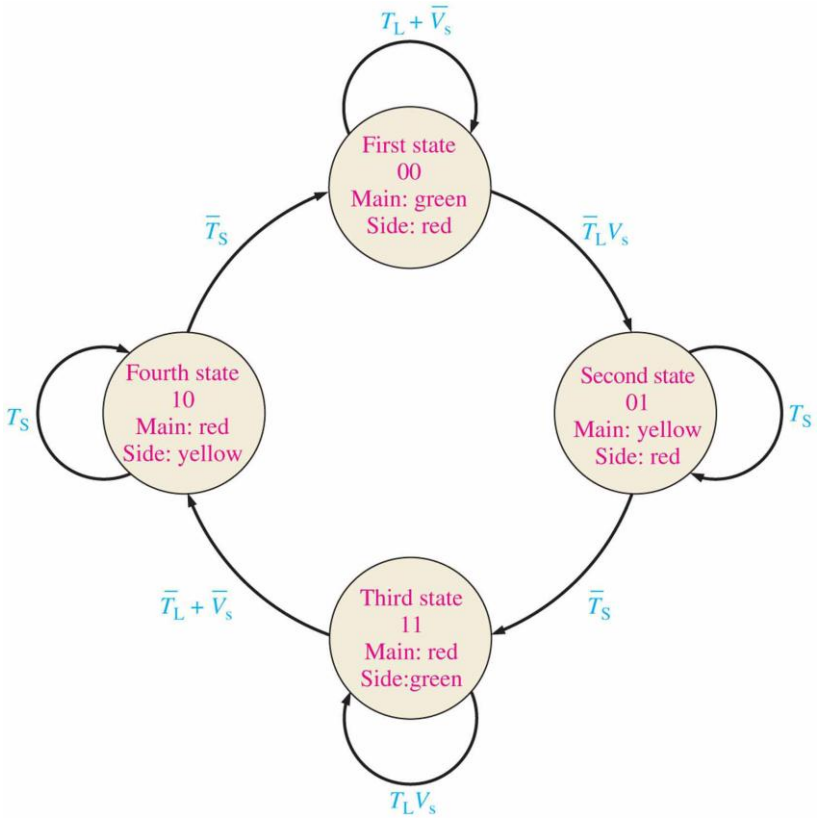


Figure P5-3. FSM of the traffic signals system

[13] The formulas must be properly derived in the Project Report.

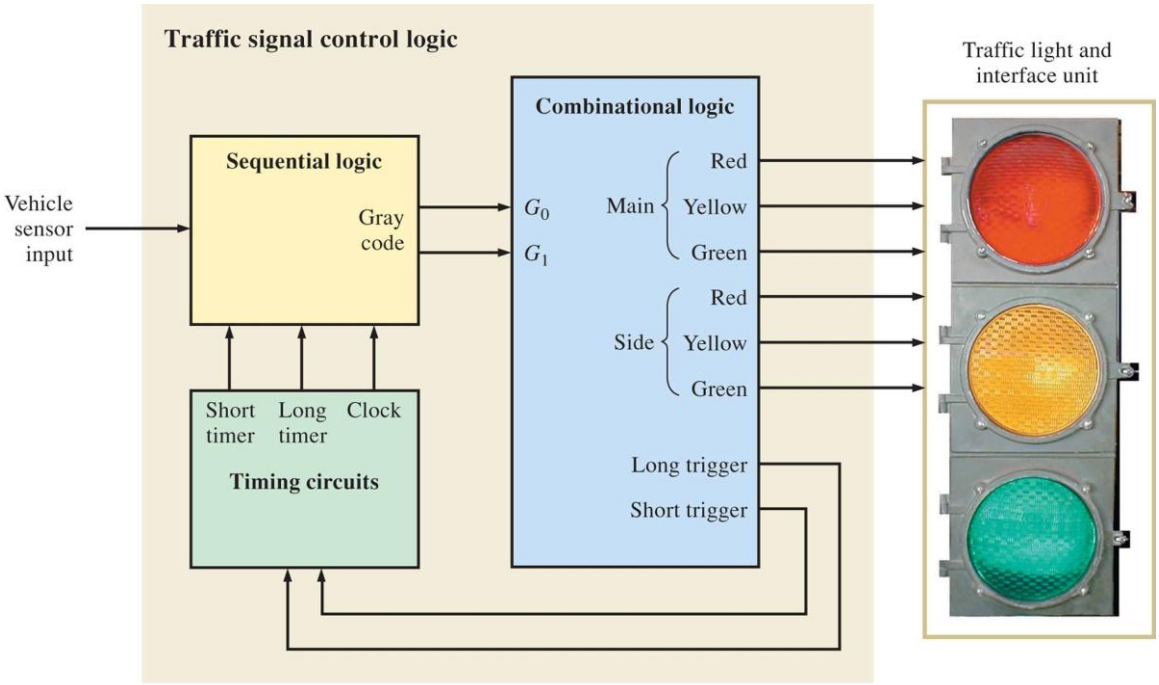


Figure P5-4. Block diagram of the traffic signal control system

Demonstration

Demostrate that the application performs according to specs. Use

- LEDs on the Digilent's Basys board of different color to emulate traffic lights, and
- Switches (or Buttons) to emulate incoming traffic

Project 6. AUTOMOBILE TURN LIGHTS CONTROL

Challenge

Given a high-level state machine, implement the state machine as a datapath and a controller.

Background and Specification

Figure P6-1. Audi R8 "quattro" Coupe 2009 Turn Lights

Audi has developed the turn lights as shown in **Figure P6-1**. The project assignment is to develop a cheaper version of the turn lights which contains 4 LEDs for each turn signal. It functions similarly to the turn lights described in Wakerly's text for the Ford Thunderbird (T-bird) automobile. For example, when the LEFT turn signal is ON, the LEFT turn signal is indicated by sequentially turning ON the LEDs running to the left (1 LED ON, 2 LEDs ON, …, 4 LEDs ON) and similarly for the RIGHT turn light. Implement the Finite State Diagram similar to the one shown in text and duplicated here for convenience in **Figure P6-2, Figure P6-3**.

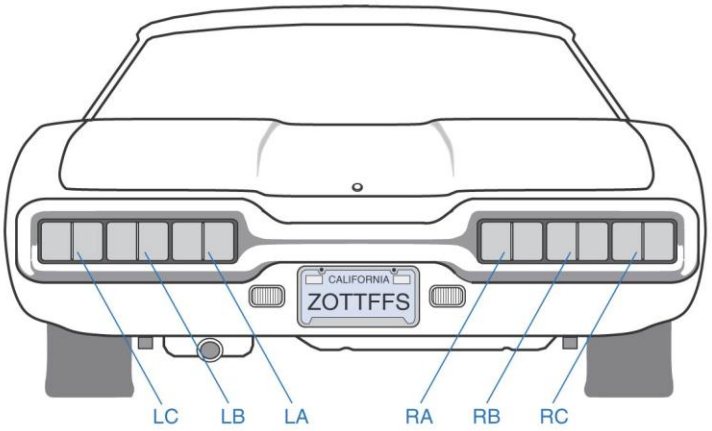


Figure P6-2. T-bird light naming

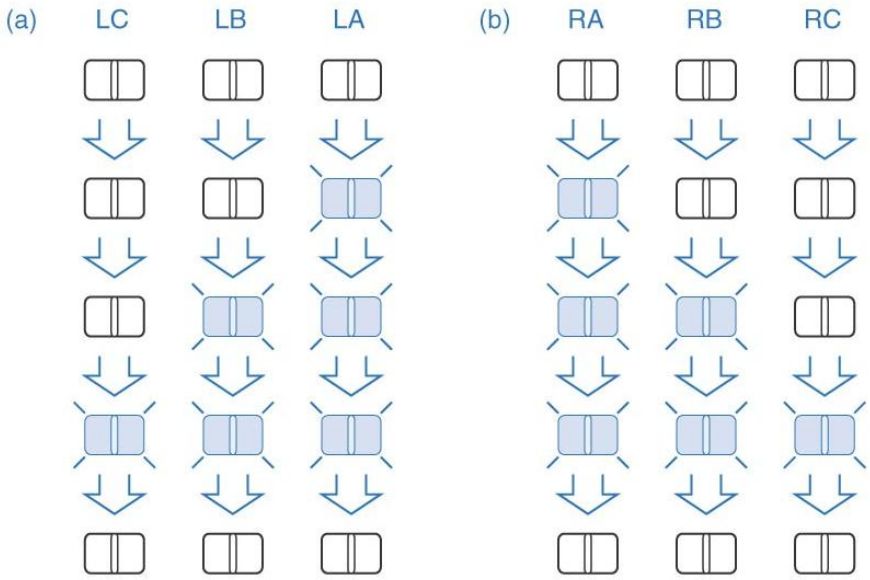


Figure P6-3. T-bird light turn sequence

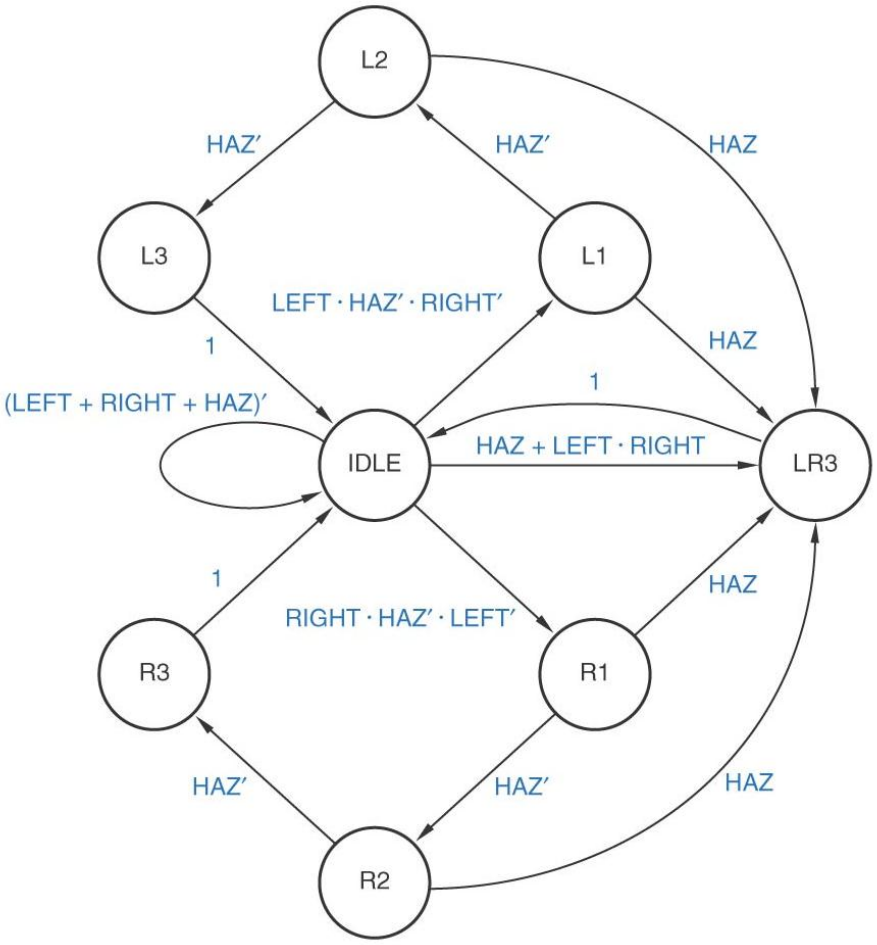


Figure P6-4. T-bird light turn light FSM

Demonstration

Demostrate that the application performs according to specs. Use

- LEDs on the Digilent's Basys board to emulate traffic lights (4 LEDs for LEFT, 4 LEDs for RIGHT, etc), and
- Switches (or Buttons) to emulate turn signal switching

Project 7. SIMPLE PROCESSOR

Challenge

Implement a simple CPU as a datapath and a controller[14].

Background and Specification

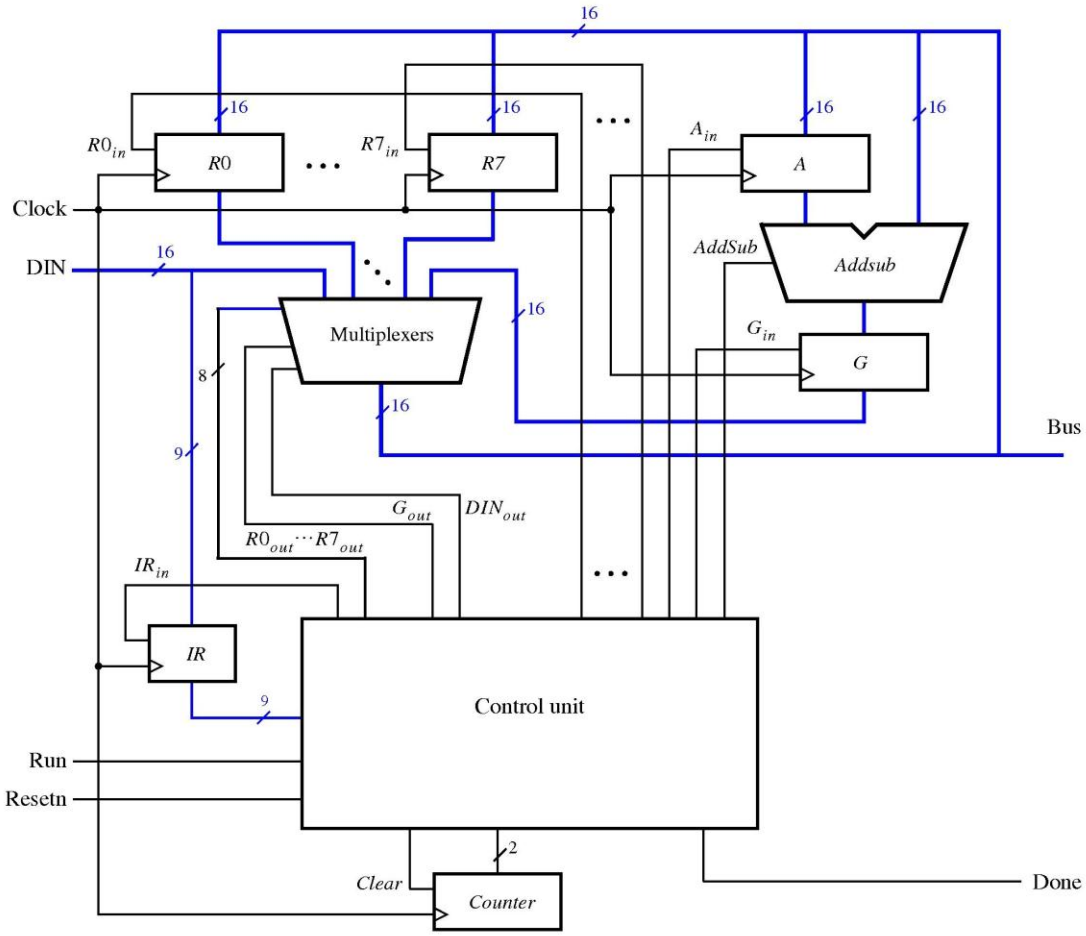


Figure P7-1. Block diagram of a simple 16-bit processor

Figure P7-1 shows a digital system that contains a number of 16-bit registers, a multiplexer, an adder/subtracter unit, a counter, and a control unit. Data is input to this system via the 16-bit *DIN* input. This data can be loaded through the 16-bit

[14] Beware, this project is ***quite challenging*** but doable. To further simplify the project develop a 4-bit processor by redesigning the block diagram of **Figure P7-1**.

wide multiplexer into the various registers, such as *R*0*, . . . , R*7 and *A*. The multiplexer also allows data to be transferred from one register to another. The multiplexer's output wires are called a *bus* in the figure because this term is often used for wiring that allows data to be transferred from one location in a system to another. Addition or subtraction is performed by using the multiplexer to first place one 16-bit number onto the bus wires and loading this number into register *A*. Once this is done, a second 16-bit number is placed onto the bus, the adder/subtracter unit performs the required operation, and the result is loaded into register *G*. The data in *G* can then be transferred to one of the other registers as required.

The system can perform different operations in each clock cycle, as governed by the *control unit*. This unit determines when particular data is placed onto the bus wires and it controls which of the registers is to be loaded with this data. For example, if the control unit asserts the signals $R0_{out}$ and A_{in}, then the multiplexer will place the contents of register *R*0 onto the bus and this data will be loaded by the next active clock edge into register *A*. A system like this is often called a *processor*. It executes operations specified in the form of instructions. Table 1 lists the instructions that the processor has to support for this exercise. The left column shows the name of an instruction and its operand. The meaning of the syntax RX ← [RY] is that the contents of register RY are loaded into register RX. The **mv** (move) instruction allows data to be copied from one register to another. For the **mvi** (move immediate) instruction the expression RX ← D indicates that the 16-bit constant D is loaded into register RX.

Table P7-1. Instructions performed in the processor

Operation	Function performed
mv *Rx,Ry*	*Rx* ← [*Ry*]
mvi *Rx,#D*	*Rx* ← *D*
add *Rx,Ry*	*Rx* ← [*Rx*] + [*Ry*]
sub *Rx,Ry*	*Rx* ← [*Rx*] − [*Ry*]

Each instruction can be encoded and stored in the *IR* register using the 9-bit format IIIXXXYYY, where III represents the instruction, XXX gives the RX register, and YYY gives the RY register. Although only two bits are needed to encode our four instructions, we are using three bits because other instructions will be added to the processor in later parts of this exercise. Hence *IR* has to be

connected to nine bits of the 16-bit *DIN* input, as indicated in **Figure P7-1**. For the **mvi** instruction the YYY field has no meaning, and the immediate data #D has to be supplied on the 16-bit *DIN* input after the **mvi** instruction word is stored into *IR*. Some instructions, such as an addition or subtraction, take more than one clock cycle to complete, because multiple transfers have to be performed across the bus. The control unit uses the two-bit counter shown in **Figure P7-1** to enable it to "step through" such instructions. The processor starts executing the instruction on the *DIN* input when the *Run* signal is asserted and the processor asserts the *Done* output when the instruction is finished. **Table P7-2** indicates the control signals that can be asserted in each time step to implement the instructions in Table 1. Note that the only control signal asserted in time step 0 is IR_{in}, so this time step is not shown in the table.

Table P7-2. Control signals asserted in each instruction/time step

	T_1	T_2	T_3
(**mv**): I_0	*RYout*, *RXin*, *Done*		
(**mvi**): I_1	*DINout*, *RXin*, *Done*		
(**add**): I_2	*RXout*, *Ain*	*RYout*, *Gin*	*Gout*, *RXin*, *Done*
(**sub**): I_3	*RXout*, *Ain*	*RYout*, *Gin*, *AddSub*	*Gout*, *RXin*, *Done*

Demonstration

Design a 4-bit processor of the type shown in **Figure P7-1**. Cleverly use switches, buttons and LEDs to demonstrate the performance.

Project 8. ARITHMETIC LOGIC UNIT (ALU)

Challenge

Implement the ALU datapath.

Background and Specification

The ALU is a combinational circuit that performs a set of basic arithmetic and logic operations. It has a number of selection lines used to determine the operation to be performed. The selection lines are decoded within the ALU, so m selection lines can specify up to 2^m distinct operations. **Figure P8-1** shows the symbol for a typical n-bit ALU. The n data inputs from A are combined with the n data inputs from B to generate the result of an operation at the G outputs. The mode-select input S_2 distinguishes between arithmetic and logic operations. The two Operation select inputs S_1 and S_0 and the Carry input C_{in} specify the eight arithmetic operations with S_2 and 0. Operand select input S_0 and C_{in} specify the four logic operations with S_2 and 1.

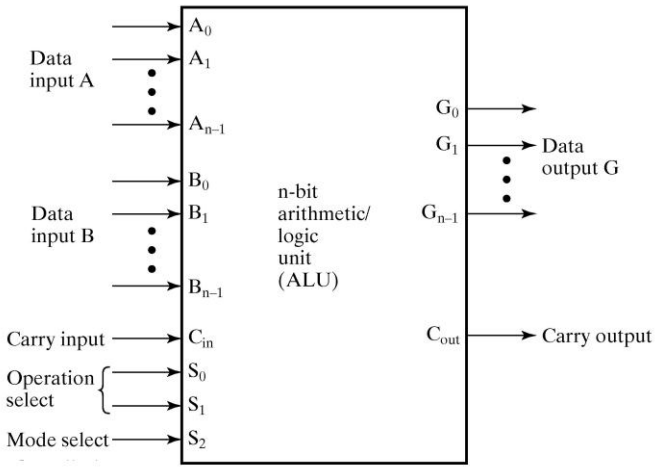


Figure P8-1. ALU schematic symbol

Demonstration

Design an ALU performing operations of your choice using extenders and demonstrate the performance on the Digilent's Basys board.

Project 9. LASER DISTANCE MEASURER[15]

Challenge

Given a high-level state machine, implement the state machine as a datapath and a controller.

Background and Specification

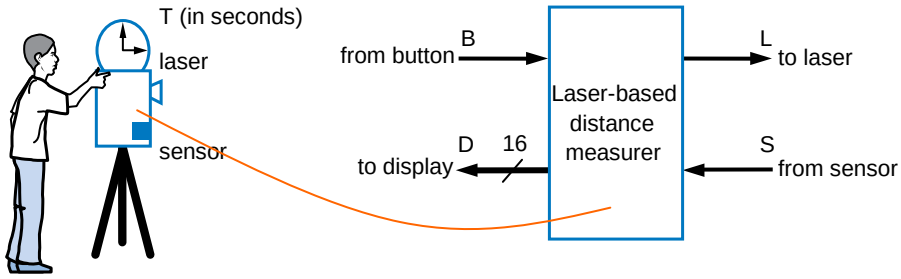


Figure P9-1. Laser Distance Measurer

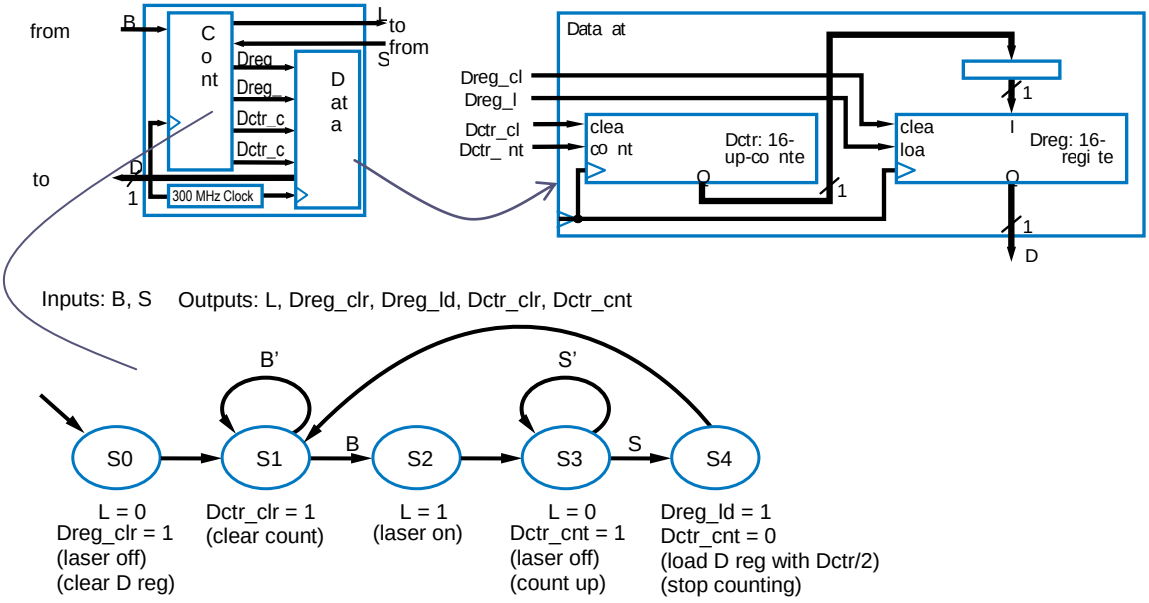


Figure P9-2. Laser Distance Measurer FSM and implementation Block Diagram

[15] For more information see **Lecture 18** and **Lecture 16 Supplement** (for the divider logic)

Inputs/outputs

- *B*: bit input, from button to begin measurement
- *L*: bit output, activates laser
- *S*: bit input, senses laser reflection
- *D*: 16-bit output, displays computed distance

Demonstration

Design an ALU performing operations of your choice using extenders and demonstrate the performance on the Digilent's Basys board.

Project 10. SECURITY SYSTEM

Challenge

Given a high-level state machine, implement the state machine as a datapath and a controller.

Background and Specification

A security system provides coded access to a secured area. A block diagram for the security system is shown in **Figure L10-1**.

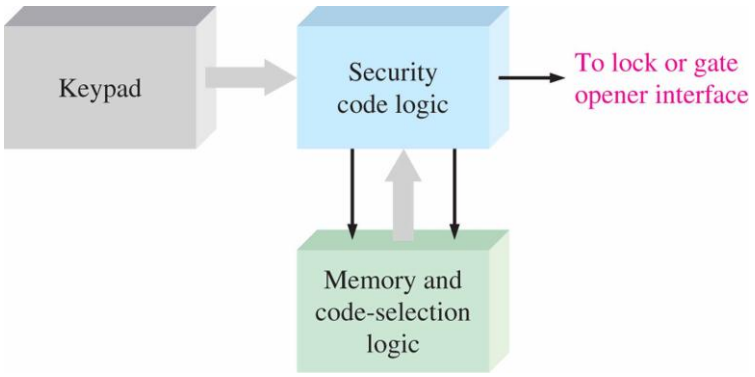


Figure P10-1. Block diagram of the security system

A system consists of the security code logic, the memory and code selection logic, and the keypad. A 4-digit entry code is set into the memory with user accessible switches. If the entered code agrees with the one stored in memory the access mechanism allows the door (or gate) to be opened. Otherwise it indicates a wrong code entry.

A sample block diagram of the implementation is shown in **Figure P10-2**. While the actual implementation would require the usage of non-volatile memory, a prototype can utilize File Register as memory for access code storage.

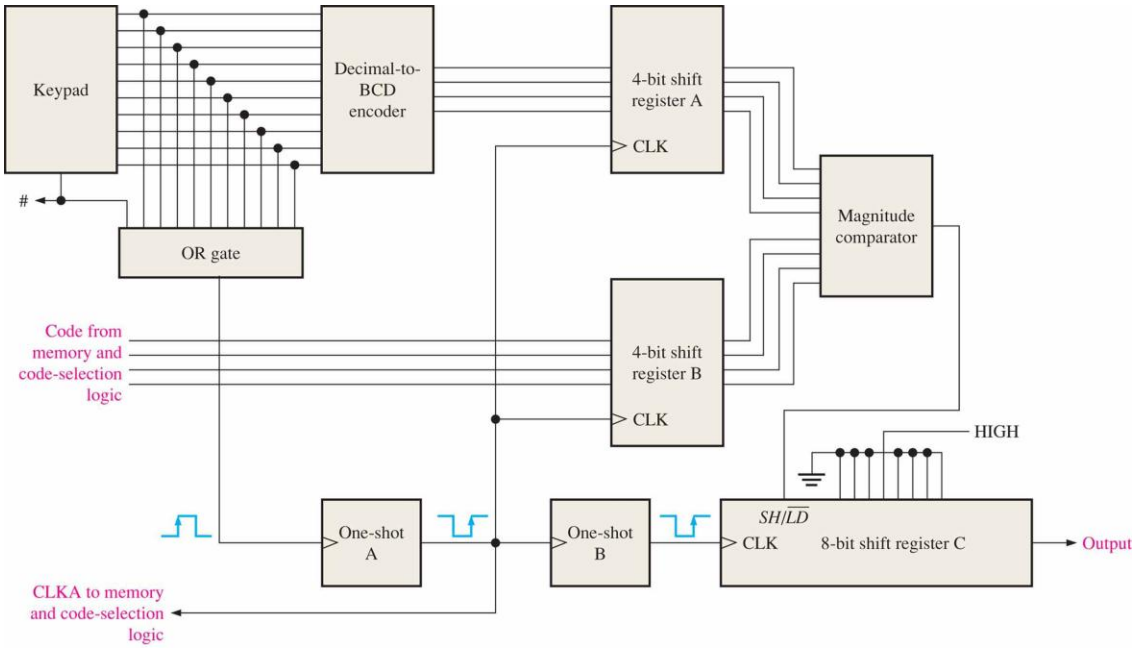


Figure P10-2. Block diagram of the security code logic

Demonstration

Design a security system[16] that can be re-programmed with different access codes and performs according to the specs on the Digilent's Basys board.

[16] As mentioned in the beginning the variations of the system are welcome.