Laboratory Manual:
Digital Design Methodologies

ECE 152B, Spring 2013

Department of Electrical and Computer Engineering
University of California, Santa Barbara

Instructor : Dr. Tim Cheng

Contents

Introduction

The ECE 152B Digital Lab is intended to complement the theory of Digital Design Methodology taught in the lectures, and to provide actual experience in building practical projects from off-the-shelf SSI TTL hardware and programmable array logic. The course consists of a set of two projects. You are expected to be familiar with logic design and switching theory (ECE 152A), and assembly language programming (ECE 15A&B).

Laboratory work is organized in groups of no more than three and is evaluated through demonstrations and written reports. Records for laboratory work are kept on an individual basis; hence all group members should make sure that their names appear on laboratory reports.

1.0 General Laboratory Instructions

- Lab work is conducted in the Digital Systems Laboratory, Harold Frank Hall (HFH) Room 1124. To access the lab, students should obtain a card key from the ECE Shop on the first floor of HFH. It is important to attend at least one scheduled lab session during the week so that important information and instructions about the labs can be obtained from a TA.
- Laboratory work will be conducted in groups of 2 students. At the beginning of the quarter each laboratory group will be assigned a locker containing a tool box, a UCSB/ECE DigiLab FPGA board and some other components. Groups are collectively responsible for issued equipment. Failure to hand-in equipment will be treated in the same manner as failure to return books to the library. Unreturned equipment will be billed to your BARC account.
- Teaching Assistants will be present in the laboratory during scheduled laboratory and office hours. Assistance at other times can be arranged with TAs on an individual basis.
- **Read experiment requirements and specifications carefully.** In your design, never change the specifications. If you have any doubts or questions, consult your TA.
- Complete your design, including logic diagrams, software listing and flowcharts before implementing it. The design must be approved by a TA and only then will you get parts for the experiment.
- Implement your design in parts (software as well as hardware), test each part separately, and then try the whole system. It will save you a lot of time in debugging.
- After an experiment is successfully completed, it must be demonstrated to one of the TAs and his/her signature obtained on a report cover sheet. These sheets are available from a TA. Please note that all group members must be present during the demonstration.

- Your laboratory work will be evaluated according to the following criteria:
    1. Demonstration
    2. Correct and complete documentation
    3. Neatness
    4. TA's opinion based on questions asked during demonstration
- Familiarize yourself with the Lab workbench and the following equipment on the bench:
    1. Oscilloscope
    2. Pulse Generator
    3. Logic Analyzer
    4. Power Supply
    5. PC
- Procedures for the individual experiments are described in this manual. You will be expected to read the instructions carefully, and understand the experiment before coming to the lab. If you have any questions or doubts, you should meet with your TA or instructor and sort them out.
- Each experiment will involve four parts --- Design, Implementation, Demonstration and Report.
    1. The design should be completed before the start of the lab, so that you can get on with the implementation quickly. This will generally involve chip-level design of a circuit from its high-level design description and helpful hints given in the hand-outs, and deciding on the *exact* procedure for the particular lab. You must write and debug the Verilog or VHDL of your design for simulation using ModelSim, demonstrate the simulation results to your TA, and answer any questions he/she may ask, before starting on your implementation.
    2. The major time spent in the lab should be on actual implementation, and subsequent debugging. This will probably be the most time-consuming part of the lab, and you might need to consult with your TA if you are having a particularly difficult problem.
    3. When your implementation is complete, you are expected to demonstrate the working set-up to your TA, and answer specific questions that he/she may have. These questions include those asked in the experiment description. Each member of the group should be able to answer questions regarding all aspects of the lab. The group members will be graded individually during the question phase of the demonstrations.
    4. Finally, you should prepare a *short* report, detailing your design, the circuit, and any software, and answer questions asked in the hand-out. Any specific, interesting observations should also be reported. The format of the report should be precise and to-the-point, and should avoid excessively verbose

descriptions. Use figures wherever applicable. *"A picture is worth a thousand words" - Confucius*. The report for a lab will be due before the start of the next lab.

- Your grade in a lab will depend upon all of the above, and the TA will not be expected to give you the exact break-up for the various parts. Besides, he/she may use his/her discretion to award different grades to different students in the same lab-group based on individual participation in and/or understanding of the lab.
- The lab will be worth 40% of the total grade for the course. The exact break-up for the individual labs and quizzes will be decided by the TA.

2.0 Guidelines for Lab Write-Ups

Your lab write-up should be a document that describes the performance of your lab. That is, what did it do, and more importantly, WHY. The lab write-ups are a group effort, meaning that all members of the group should contribute. Taking turns writing the different lab write-ups is frowned upon, and may result in a lower write-up grade. The write-ups are worth 50 percent of the lab grade, and are due during the week following the final demo date. Notice that if you demo late, the due date for the write up does NOT get moved back.

The following is a guideline of items to be included in the write-up. Note, this is **NOT** all-inclusive. If you have comments about certain portions of the lab, include them. Similarly, if you have unexpected results, or if you think you have a good way of solving a problem, document it. If any portion of your hardware/software did not work at the demonstration, include a reasonable explanation of why in your write-up.

10 percent of your write-up grade will be based on the quality of your writing style. We will be looking for grammar, spelling, and sentence construction. Grades will be based on the following scale: 0-lousy; 3-acceptable; 7-good; and 10-outstanding. Notice that the write-ups will sometimes be long, so plan ahead.

1. Title page. Which lab is being written-up, the date, the lab group members, etc.
2. Abstract. This should be a brief (1 paragraph) summary of the experiment and the major results. Think of it as intended for someone who won't have time to read the report but who wants to know what it's about.
3. Hardware description. What we're looking for here is a description of the FUNCTION of each of the individual blocks that made up your circuit. Also include some discussion about how and **WHY** certain portions of the circuit work (OPERATION). This section should not go to the same level of detail as the part specification sheets. However, any peculiarities of a component that drove some aspect of the design should be discussed.

4. Schematic diagram. Provide a schematic diagram for your circuit. Label all chips and buses, but it is not necessary to include every wire. Rather, if there is a group of wires leading from one chip to another (known as a bus), you may draw one line, label it as denoting multiple wires, and name it (i.e. address bus). Schematics should be drawn in a simple manner so that they are readable, but they must include all information necessary for someone else to reconstruct the system.
5. Verilog or VHDL descriptions of your design, along with the testbenches and simluation results. Discuss your experience in the simulation process and hardware bugs found, and in turn modifications made to your design, in this process.
6. Discussion. This section should contain brief answers to the questions asked in the experiment description.
7. Conclusion. Include in this any problems you encountered with the lab, and comment upon any unusual or unexpected results from your lab. Try to explain your results.
8. Grade sheet. As the last sheet of your report, attach the grade sheet that the TA will give you after the demonstration.

3.0 Deadlines

Experiment #1:

- Demonstration - Week 3 of the quarter in your lab session
- Report - Wednesday of week 4 of the quarter

Experiment #2:

- Design approval and your debugging plan - Week 5 of the quarter in your lab session
- Simulation of entire design completed- Week 7 of the quarter in your lab session
- Demonstration - 9th week of the quarter in your lab session
- Report - Wednesday of Week 10 of the quarter

Experiment 1: ALU and Pipelining

4.0 Introduction

This lab is intended to familiarize you with two important aspects of microprocessor design before you begin the actual construction of BURP (Basic Undergraduate RISC Processor). In the first part, you will build a data path that exercises the 74LS181 ALU. This will involve programming an EEPROM to supply a sequence of operations to the ALU, then verifying that the correct result was written to the appropriate location in a RAM. In the second part, you will pipeline the data path to improve its performance. This will involve choosing the best place to insert registers that will divide the data path into stages. A discussion of pipelining follows this assignment.

5.0 Getting to Know the ALU

5.1 Hardware Description

The first step is to construct the data path shown in Figure 1. The data path consists of an EEPROM that contains the program to be executed. The EEPROM is addressed by the program counter (PC) which should be a 4-bit counter with asynchronous clear. The 8-bit output of the EEPROM has three functions. Four bits will be data provided to the A input of the ALU. Two bits will control the function of the ALU. Note that will only allow you to perform 4 of the possible 16 functions in each mode. The last two bits will select the destination register location in the CY7C130 SRAM. The output of the ALU is connected to both the CY7C130 SRAM and a discrete register. The discrete register serves as an accumulator and will always contain the result of the last operation. The accumulator also provides the data to the B input of the ALU. As a result all operations are accumulator-based operations because the accumulator is always one of the two operands of the operation. The outputs of the CY7C130 SRAM will be connected to buffered LEDs so that you can see what is in each of the register locations.

5.1.1 ALU

The 74181 ALU will be used to provide the functionality in the data path you will be implementing. Data sheets on the ALU are included at the end of this manual. Use these data sheets to help you connect and operate the ALU properly. The ‘181 is a 4-bit ALU that can be cascaded with other ‘181s to create a unit that can operate on wider operands. A single ‘181 takes two 4-bit operands, designated by A and B, and provides a single 4-bit result, designated by F. The function of the ALU is selected by the pins S3, S2, S1, and S0, in addition to mode pin (M) that loosely selects between arithmetic and logic functions. Data to the A inputs of the ALU will come from the EEPROM (where the program is stored). Data to the B inputs of the ALU will come from the accumulator, which contains the result of the last operation. The output of the ALU will go to the input of the accumulator and to the

data inputs of the register file. The ALU also has a carry-in and a carry-out. It will be necessary to store the carry-out signal and provide a signal to the carry-in input. The value of the carry-in signal will depend on the operation being performed. Appendix II to this experiment discusses the carry signal in more depth.

The operations to be implemented are move (F=A), inversion of the accumulator (F=$\overline{B}$), add (F=A plus B plus carry), and subtract (F=A minus B). The add and subtract operations should be done in twos' complement form.

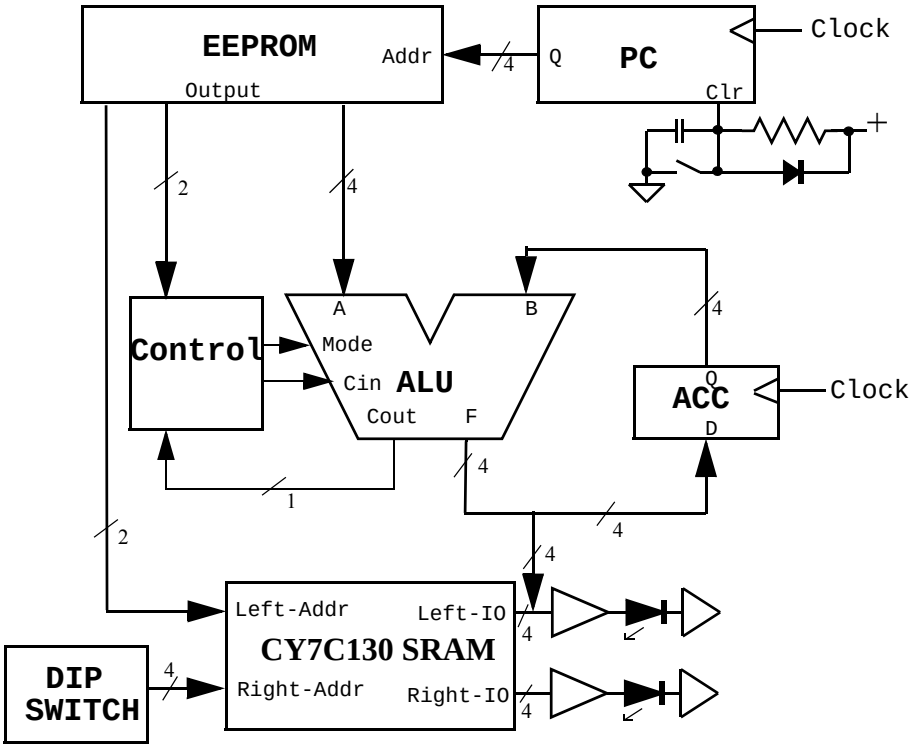


Figure 1: Non-Pipelined Data Path

5.1.2 CY7C130 or IDT 7130SA Dual Port SRAM

The CY7C130 is a 1K by 8-bit words dual port SRAM. IDT 7130SA is a drop-in compatible SRAM with CY7C130. So you can use either part. Either SRAM has 1000 memory locations that are each 8-bit wide. Two ports *LEFT* and *RIGHT* ports) are provided permitting independant access (read/write) to any location in memory. Each port has its own control pins; chip enable $\overline{CE}$, write enable $R/\overline{W}$, and output enable $\overline{OE}$.

* To write Data into CY7C130: ($\overline{CE}$) and ($R/\overline{W}$) need to be low with ($\overline{OE}$) set to high.
* To read Data from CY7C130: ($\overline{CE}$) and ($\overline{OE}$) need to be low with ($R/\overline{W}$) set to high.

Each port has two flag signals $\overline{BUSY}$ and $\overline{INT}$ which are not required in this experiment. Hence, they do not need to be connected.

Use the *LEFT* port $A_{0L\text{-}9L}$, $I/O_{0L\text{-}7L}$, $\overline{CE}_L$, $\overline{OE}_L$, and $R/\overline{W}_L$ for writing data into the SRAM. The two least significant bits (LSBs) of the *LEFT* address lines ($A_{0L\text{-}1L}$) should be provided by two of the EEPROM data lines. Since there are 10 address lines, the remaining 8 address lines should be grounded. There are 8 availabe data lines from the SRAM, but only 4 data lines are required. Hence, unused data lines need not be connected.

The *RIGHT* port can be configured to read the content of the SRAM for debugging and analysis purposes. A dip switch can be connected to the *RIGHT* address lines ($A_{0R\text{-}9R}$), allowing a specific RAM location to be observed while the system is running.

5.1.3 2816 EEPROM

The 2816 EEPROMs are Electronic Erasable Programmable Read Only Memories with 16 Kbits of storage. Since each word is 8-bits wide (byte), the 2816 stores 2 KBytes. The EEPROM has 11 address lines and 8 data lines. The address lines should be connected to the output of the Program Counter (PC). In this lab, the PC only has 4 output lines, so the upper 7 address lines of the EEPROM should be grounded. The output enable pin controls whether the outputs are high-impedance or not. This pin should be grounded so that the outputs are always enabled. Consult the data sheet for help in connecting the rest of the pins. The EEPROM will store the program that controls the data path. The program will consist of single-word instructions so that each instruction can complete in one clock cycle. As a result, 8-bits must control all the functions of the data path. The lower four bits will be the operand for the A input of the ALU. The next two bits will select one of four operations of the ALU. The most significant two bits of the instruction word will be the destination address. They will designate to which of the lower four RAM locations in the CY7C130/IDT7130SA the result of the ALU should be written.

The EEPROMs are programmed by using the universal programmer connected to one of the bench computers. Before you program the EEPROM you need to list the 16 instructions that will constitute your program. These instructions should be listed in hexadecimal notation. Then, in the EEPROM programming software, you can enter your program data into locations 00h to 0Fh of the memory and program it into the EEPROM. The EEPROM programmer is located by one of the bench computers. The TA will demonstrate programming of the EEPROM during the first of each of the scheduled lab sessions.

5.1.4 Accumulator

The accumulator should be a 4-bit register. Recall that a register is simply an array of edge-triggered flip-flops. The purpose of the accumulator is to store the result of the last operation. A suitable chip to serve as the accumulator is the ‘374. This chip has eight flip-flops so some of the extras could be used for the controller.

5.1.5 Controller

The controller performs two main functions. First, it decodes the two bits coming from the EEPROM that represent the operation to be performed and provides the appropriate mode and select signals to the ALU. Second, it determines the proper value for the carry-in signal to the ALU. This signal will depend on the carry-out from the last operation and on what the current operation is doing. The controller should be implemented by the UCSB/ECE DigiLab FPGA Board. You should write the VHDL or Verilog code for the controller and simulate the code using ModelSim to verify its correctness before implementing the controller in the UCSB/ECE DigiLab FPGA Board.

5.2 Operation of Non-Pipelined Data Path

The objective of the first part of this experiment is to exercise the ALU and understand its operation in a simple, non-pipelined data path. In preparation for the second part of this assignment, you will record the maximum frequency at which you can operate this data path.

You must devise a 16 line program that will exercise each of the four implemented operations of the ALU (see EEPROM). It is up to you to decide when and what instructions to perform, but you should use each of the operations with various operands. However, a good tip would be to make your program deterministic. That is, make sure the program will do the same thing no matter what the state of the system is when it is turned on for the first time. A good way to do this is to make the first instruction be the MOVE instruction. This way you will know exactly what is in the accumulator by the second instruction and you will know that there was no carry so the carry flag will be zero. Another tip is to make one of the bits of the result (from the ALU) and one of the bits of the instruction word (from the EEPROM) toggle every instruction. This will produce two square waves (one for each bit) when you run the system and will allow you to easily measure the propagation delays of the various components of the data path. Also, you should make one of the destination registers hold a constant value that is written to it repeatedly. Then, by watching this value you can determine if the machine is working properly. If you increase the clock frequency too much, the value will probably change.

In this non-pipelined version, the system should be clocked such that an entire operation, including display of the output, completes before the beginning of the next clock cycle. One particularly tricky part of this project is controlling the $R/\overline{W}$ and $\overline{OE}$ pins of the RAM. An easy solution is to connect them straight to the clock. This will cause data to be written into the RAM during the low phase of the clock cycle. Note, however, that the address to the RAM must be valid before the falling edge of the clock.

Once you have written your program and burned it into the EEPROM, single step your data path while checking the memory locations to see that everything works as expected. Then determine the maximum clock frequency at which your system can operate correctly. Additionally, determine the propagation delays of each of the components in the data

path (program counter, EEPROM, ALU, and CY7C130), compare them to the values from the data sheets, and use them to justify the overall frequency you measured.

6.0 Pipelining the Data Path

Following this assignment is a discussion on pipelining. Before attempting the next part of this assignment, you should have a clear understanding of the purpose and methodology of pipelining.

The next step of the assignment is to insert pipeline registers into the data path to divide it into stages. The data path should be divided into two stages in an effort to increase the clock frequency. You need to decide on the most advantageous place to insert these registers. Note that you should not have to modify your program at all to work with the new pipelined data path. After pipelining your data path find the new maximum clock frequency. Also find the propagation delay through each stage. Remember to account for the delay of the newly added pipeline registers.

7.0 Report and Demonstration

Be prepared to demonstrate the proper functioning of your program on the pipelined version of your system to your TA. Also be prepared to answer any questions related to the operation and architecture of the components used in this assignment (74181, IDT7130SA/CY7C130, EEPROMs, registers, counters, the controller, etc.) as well as design considerations for the data path and pipelining.

Your lab report should contain the following:

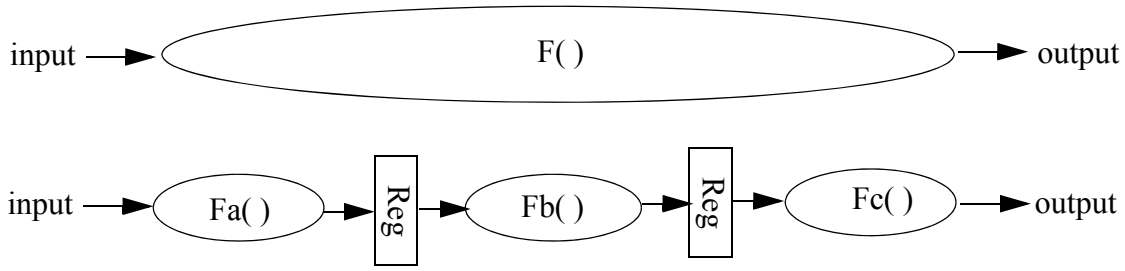
1. All standard report items (Description, schematic ... see the Introduction)
2. A listing of your program in register transfer notation accompanied by the codes used.
3. Discuss the location of the pipeline registers and why you placed them where you did.
4. A comparison of the pipelined and non-pipelined versions of the system. Which is faster, which has more throughput, etc.

Appendix I: Pipelining Concepts

What is pipelining?

Simply put, pipelining is the process of breaking up a time-critical data path into a series of smaller data paths by placing registers between sections. Assuming there are no dependencies between the stages of the pipeline, the registers effectively isolate sections of the data path. This allows succeeding operations to begin once information has been latched by the following stage. The clock period is determined by the segment with the longest, or *critical*, delay.

FIGURE 1. Introduction of registers



There are a number of factors that determine the degree of improvement for a pipelined system. The placement of the breakpoints in the data path will have critical implications for the speed of the design. This is because the cycle time is based upon the critical delay of each of the data path segments. Therefore, the designer must partition the data path into segments with as close to equal delays as possible.

The process of selecting breakpoints to evenly divide the delays of the data path is the difficult task of the designer. Several conditions must be met. First, registers can only be inserted between physical components or, in other words, only where their placement is physically possible. Second, there can be no dependencies between any of the stages. A dependency occurs whenever two or more stages try to share a common resource at the same time or if the input to a stage is dependent upon a result which has not propagated through the pipeline. This type of behavior is commonly experienced when a unit tries to read and write to memory through a common port or has a conditional jump which is dependent upon the result of a previous instruction. In either case, a pipeline stall may be introduced to allow the later part of the pipeline to complete and yet halt the stage with the conflict and all elements before it. This should not be a problem for your experiment.

The benefit of pipelining is that several instructions are being worked on at the same time. Like a car on an assembly line, each instruction goes through several stages. In an unpipelined machine, each stage does its job and then may have to wait a long time for the next instruction to come along. Not pipelining is like waiting for a car to be painted, polished, and out the door before starting to weld the frame for the next car. In a pipelined machine, each stage does its job on one instruction, passes the results on, and then starts to work on

the next instruction, just like on an assembly line. To prevent a pile-up of work, the line can only go as fast as the slowest worker (or stage).

In a pipelined machine, one instruction will take approximately as long to finish as the same instruction in an unpipelined machine. However, because several instructions are being worked on at the same time, the second instruction will be finished much sooner than the second instruction in an unpipelined machine, the third instruction much, much sooner, and so on. The benefits of a pipelined machine build up as you perform more and more instructions.

Finally, it should be noted that introducing pipelining does have some negative impact. There is a small delay required for the latching of the intermediate results (between stages) and any extra control structure resulting from handling pipeline stalls may introduce an additional delay. This means that individual instructions will actually take slightly longer in the pipelined machine. Therefore the designer should integrate these penalties into the computations of the segment timing delay when considering possible breakpoints.

Appendix II: Carry Operations

The Carry Bit

The implementation of the carry-in and carry-out control circuitry has caused quite a bit of confusion in the past. Most of the problems revolve around two aspects of the 74181 ALU: the use of ones' complement subtraction and the use of negative logic for the carry bit. This section attempts to explain both the problems and the requirements for the lab.

Ones' Complement Subtraction

The 74181 ALU performs ones' complement subtraction, not twos' complement subtraction as required by both experiment specifications. Remember that the ones' complement of a number is computed by simply inverting each bit in a binary number. Therefore, ones' complement subtraction works like this:

7 - 5 = 0111 - 0101 = 0111 + 1010 = 0001 plus a carry = 2

The carry means that the answer is positive and must be incremented by one, so the final answer is 2. Here's another example:

5 - 7 = 0101 - 0111 = 0101 + 1000 = 1101 with no carry = -2

No carry means that the answer is negative and every bit must be inverted.

In contrast, a twos' complement of a number is computed by inverting each bit and then adding one to the result. Here are both examples in twos' complement:

7 - 5 = 0111 - 0101 = 0111 + (1010 + 1) = 0111 + 1011 = 0010 with a carry = 2

5 - 7 = 0101 - 0111 = 0101 + (1000 + 1) = 0101 + 1001 = 1110 with no carry = -2

Note that the interpretation of the number does not depend on the presence or absence of the carry bit. The four bit result is the answer in twos' complement form.

To get the ALU to do twos' complement subtraction, you have to manually add in an extra one with every subtraction. The best way to do this is set the carry bit before you do a subtraction. This should be hardwired into your implementation. Then, in effect, you are doing the following operation:

7 - 5 + carry = 0111 - 0101 + 1 = (0111 + 1010) + 1 = 0111 + (1010 + 1)

This is twos' complement subtraction.

Negative Logic

To add an extra layer of confusion, the 74181 ALU uses inverted logic for the carry bit. The specification lists two tables of functions. One is for negative logic data, i.e. a one is represented by zero volts and a zero is represented by 5 volts, and a positive logic carry bit, i.e. a carry is represented by 5 volts and no carry is represented by zero volts. The other table is for positive logic data and a negative logic carry bit.

For most people, positive logic seems more intuitive, so generally a positive logic data set-up is chosen. In addition, most of the available combinational logic (AND gates, OR gates, etc.) are labeled by their positive logic functions. If this representation is chosen for the data, then the carry bit will be in negative logic, i.e. a carry will be represented by zero volts and no carry will be represented by 5 volts. The same logic is used at both the carry-out output pin and the carry-in input pin.

The best way to avoid confusion is to map out the functionality of your carry circuitry by deciding when you want the carry SET, when you want the carry CLEAR, and when you just want to store the incoming carry. Only when you actually design the circuit that implements your desired function should you consider whether a CLEAR is logical one or a logical zero and chose your gates appropriately.

Summary

This discussion has the following implications for your experiment:

1. A carry must be forced before each subtraction. For each addition, use whatever carry came out of the last operation. If you are doing a move or inversion, the carry bit should be cleared for the next operation.

2. Keep in mind that the logic for the carry bit will be the complement of the logic for your data when you design the controller logic.

Experiment 2: Designing BURP

8.0 Description

In this experiment the 74LS181 ALU and IDT7130SA (or CY7C130) SRAM are used to implement a simple RISC processor. BURP stands for Basic Undergraduate RISC Processor. Figure 1 shows a block diagram of the system. The two blobs in the block diagram are what you must design to make a functioning processor. The Interconnect blob refers to the busses required for communication between the ALU, the register file, and the controller. The arrows from the Interconnect to the ALU and register file are two way since data must be written to and read from both devices. All such data connections are four bits wide. The connection from the control unit to the interconnect is one way to show that the controller is used to determine how the SRAM and ALU are connected at any one time.The interconnect busses can be implemented using multiplexers or tri-state buffers.

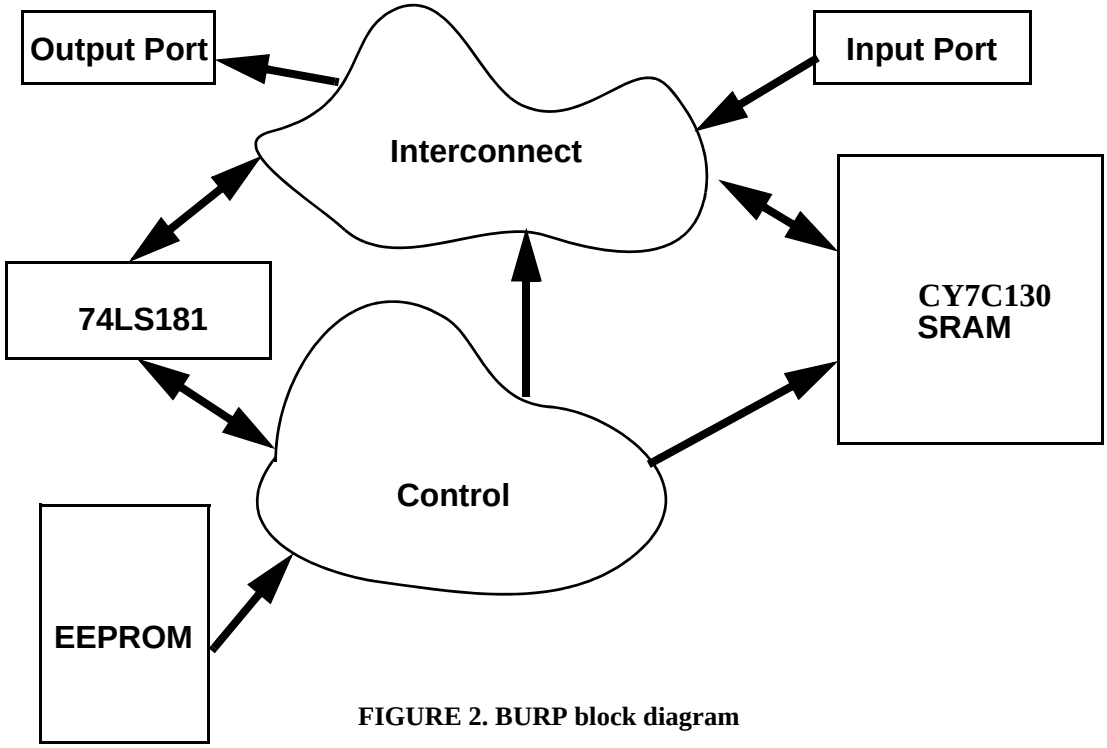


FIGURE 2. BURP block diagram

8.1 Control Unit

The control unit is the heart of the processor. It reads instructions from an external ROM where the program is stored, then sequences the appropriate control signals to the ALU, interconnect, and SRAM to execute the instruction. Figure 2 shows a block diagram of the control unit. Contained within the control unit are the Instruction Register (IR) and Program Counter (PC). Your controller must also maintain a carry flag, which may be encoded as a state in the controller. The IR can be implemented with either a 74373 or 74374,

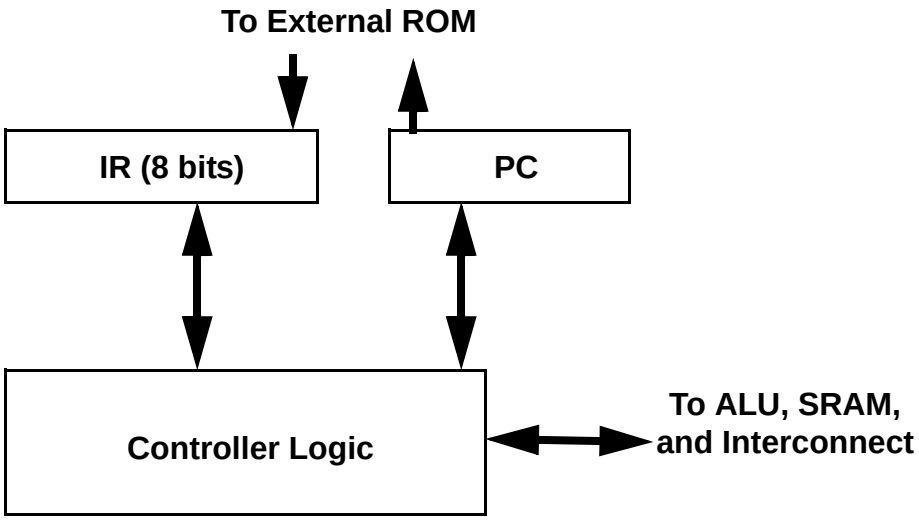


FIGURE 3. Controller Block Diagram

depending on whether your implementation requires a level sensitive or edge triggered latch. The PC counter can implemented using counters and should be capable of addressing at least 8 bits of the ROM address space.

The controller logic is to be implemented using a FPGA. Please refer to the website of the UCSB/ECE DigiLab FPGA Board (http://vader.ece.ucsb.edu/digilab-fpga/) for more information concerning programming and using the UCSB/ECE DigiLab FPGA Board.

The controller can be implemented in several ways. For example, a counter implementation, like the one discussed in class, could be used with the FPGA implementing the decoding and branching logic. Alternatively, the FPGA can be used to implement either a Moore or Mealy state machine. Both approaches will work and have different strengths and weaknesses. The design and simulation environment for the FPGA is Modelsim.

8.2 Instruction Set

Table 1 lists the instruction set which your controller must be able to interpret. Each instruction must be encoded in exactly 8 bits. The implementation should use pipelining where appropriate.

Table 1:

Mnemonic	RTL	Update Carry?
JC <address>	address + PC -> PC	no
JMP <imm> RA	(imm << 4) + RA -> PC	no
MOV R1, R2	R1 -> R2	no
MVI RA, <imm>	imm -> RA	no

Table 1:

Mnemonic	RTL	Update Carry?
INC R	R + 1 -> R	yes
ADD R1, R2	R1 + R2 + CY -> R1	yes
SUB R1, R2	R1 - R2 + CY -> R1	yes
AND R1, R2	R1 & R2 -> R1	no
OR R1, R2	R1 | R2 -> R1	no
SC	1 -> CY	yes
CC	0 -> CY	yes
PUSH R	R -> Stack	no
POP R	Stack -> R	no
IN R	IN -> R	no
OUT R	R -> OUT	no
NOP		no

In the specifications, R, R1 and R2 do not refer to specific registers but to any of four independently addressible registers (RA, RB, RC and RD) in your design. All these registers are maintained in the SRAM.

The first instruction in the table is JC, or “jump on carry”. The operand to this instruction is an address that can access four bits of the ROM space. This 4-bit address will be an immediate value contained within your instruction. This implies that you will only be able to branch at most 15 instructions ahead of the branch instruction. The carry flag should not be updated. The second instruction in the table is JMP or “jump immediate”. Its destination address is composed of two parts. The upper bits of the address are specified as an operand of the JMP instruction. The lower four bits of the destination address are located in register RA. Therefore, this JMP should be able to address eight bits of the ROM space.

The MOV instruction transfers data between two registers R1 and R2. MVI places a 4-bit value into register RA.

The ALU instructions (INC, ADD, SUB, AND, OR) are a subset of the 74LS181 capabilities. Registers are specified in the operands, and the RTL explains the actions required. Note that only 4 registers (RA, RB, RC, RD) are required. This allows you to encode instructions like ADD R1, R2 with 8 bits. SC and CC set and clear the carry flag respectively. Unlike the first experiment, SUB should not force a carry. Instead, the programmer should include a SC command prior to the start of a subtraction to get a twos’ complement result.

The PUSH function puts the contents of register R onto the top of a stack. The stack should be implemented in SRAM. The POP function moves the contents of the top of the

stack into register R. Your implementation should specifically address overflow (pushing to a full stack) and underflow (popping an empty stack) issues.

The IN instruction reads a value from a set of DIP switches and places it into register R. OUT writes the contents of register R to a latch which should be instrumented with LED's so that its value may be observed. NOP is the "no operation" instruction and, as such, does nothing.

Observe that BURP has no external memory; all program values are maintained within the 16 entry, 4-bit per entry SRAM. Again, in the specifications, R, R1 and R2 do not refer to specific registers but to any of four independently addressible registers (RA, RB, RC and RD) in your design. All these registers are maintained in the SRAM.

8.3 Simulation of the entire BURP

By end of Week 6 of the quarter, you must simulate your entire Lab 2 BURP design with Modelsim and demonstrate that your simulation results are correct before you start wiring your BURP.

You can use Verilog for simulation. The Verilog models of the components used for building BURP are available on the class website. You can easily add components, if necessary, to your VHDL/Verilog.

By building these models and simulating your complete BURP design, it should save your debugging time as you will benefit from being able to simulate and debug your design rapidly before implementing it on a bread board.

Also note that the simulator will simulate an external ROM for you! Please refer to the online simulator help for more information.

9.0 Demonstration and report

As you implement your design, you should develop a program which completely tests each instruction. Before the demonstration, you will be provided with a test program as a benchmark for your project. If your system runs correctly, no problems should be encountered. You may choose to present additional programs which demonstrate any special features of your design. Besides a working test program, grading criteria will include design, efficiency, performance, and degree of pipelining.

Your report should discuss the following areas:

1. Clocking structure of the design
2. Interconnect busses and muxes
3. Controller design details
4. Degree of parallelism

5. How fast does your implementation run?

Include a detailed schematic for your circuit. Provide all design entry information for the Xilinx FPGA.

Provide a listing of your test program. Be sure to explain what specific sections of the program are designed to test.

Provide a table which lists for each instruction:

1. The opcode
2. execution phases

Another table will list for each control signal in the design:

1. The signal name
2. Destination of the signal
3. Source of the signal
4. Which instructions (or class of instructions) either require or generate the signal.