Lab Assignment 7 –
Option 2

Guideline
This lab can be done with a partner. Required starter code is provided on Canvas.

Objective
1. Become familiar with the MIPS ISA
2. Synthesize and implement a basic MIPS processor on the Basys3 board
3. Learn how to use Verilog Text-IO to initialize a memory image for simulation
4. Extend the MIPS ISA by adding ARM-like instructions

Reading
Please read chapter 9 of your textbook *Digital Systems Design Using Verilog* for background on the MIPS ISA and the basic MIPS implementation.

Summary of tasks
You will use a model of a MIPS processor that handles a subset of the MIPS instructions. On Canvas, go to the starter files page to get the Verilog description of the model. This model is provided for you in the book. In Part A you will write a testbench for this model that uses the Verilog text-IO package to initialize the instruction memory. Once the processor is verified in simulation using this testbench, you will synthesize and implement it on the board and run a simple MIPS program to light up some of the board LEDs. Next you will augment the MIPS ISA by adding ARM-like instructions in Part B.

Part A:

Description
The starter file gives you the Verilog MIPS model for you. There are two main differences with the MIPS presented in the textbook, however: 1) The MIPS in this lab is word addressable while the textbook version is byte addressable. 2) The memory only has 128 memory locations. Therefore, your program counter and address lines will be 7-bits instead of 32-bits. You will also be given code for the MIPS memory and register file. You are also provided a template testbench. This code is appended at the end of the lab description. Here are your tasks:

Simulation

1. Take the complete MIPS model presented in Figure 9-10 and compile it in Modelsim. You will need to include the MIPS processor, memory, and register file as supporting modules (either as separate entities in the same file, or as separate entities in separate files).

2. Write a testbench to test your MIPS processor. You will use the skeleton testbench provided in the starter file. Utilize your knowledge of delay and 'display' statements to test the functionality of every instruction in your design. The idea here is to get you familiar with using the Verilog text-IO. Your testbench will use text-IO to initialize the memory with a set of instructions that you will provide in a text file (called the instruction text file hereafter). After initializing the memory with your instructions, the testbench will run the processor for as many cycles as you need to see your program working. You will need to hard-code test instructions to your instruction text file in hex or binary.

Synthesis

3. Modify the above model by adding code to interface it to the input switches and LEDs on the Basys3 board. Your interface must be able to halt operation of the MIPS processor and display the lower 8 bits of register $1 on eight LEDs. Your interface must also divide the prototyping board's internal clock to provide the model with a slow clock.
    a. You should add a new 'Halt' port to the MIPS top level. When Halt is high your processor should complete the current instruction and not proceed to the next instruction. When halt goes back to zero you should keep executing the normal flow of instructions starting from the next instruction.
    b. Map switches SW0 and SW1 to Reset and Halt, respectively.
    c. Make necessary changes to your register file so that you can use register $1 as a top-level output and map it to LEDs [7:0].
    d. The slow clock can be used to execute instructions in a manner that makes your outputs visible (when you implement this on the board). You may choose the frequency of the slow clock. Using a 100 MHz clock will make any program outputs on the LEDs a blur.

4. Write a program in MIPS assembly to create a rotating light on the LED outputs. The light rotates from one LED to the next. This rotation should not stop. So, if a number denotes a specific LED being lit, the result should be like 0,1,2,3,4,5,6,7,0,1,2,3,… In other words, the LED rotation should happen indefinitely. Make sure not to blank out the LEDs after LED 7 is lit, but go directly to light LED 0.

5. Translate this program to machine code and put it in your instruction text file. Run the testbench and analyze the processor outputs using the Modelsim Waveform Viewer. Verify that the correct values are being written to register $1 and that they are showing up on your LED outputs. Also verify that reset and halt are working.

6. Synthesize your modified MIPS model and implement it on the Basys3 board.
    a. You will not need to synthesize the testbench.
    b. The memory should be initialized with your instructions by using Verilog text-IO. Simply call your 'readmemh' function in an initial block and Xilinx will correctly instantiate block RAM and initialize them with these values for you.
    c. Correctly map the switches and LEDs.

Useful Information

1. To read instructions from a file you can use the following options:

    Store the instruction file in **HEX** format: readmemh("file name", mem, start addr, end addr)
    Store the instruction file in **BINARY** format: readmemb("file name", mem, start addr, end addr)

    The text file is in strictly **HEX** or **BINARY** format with a value on each line. You do not have to put "0x" or anything else to denote a HEX value.
    Example hex input file with three entries:
    01
    02
    03

2. Since the memory size is very large and you will have a small number of instructions, you can use a for loop in the initial block to fill the rest of the unused locations to zero. You may also get a warning that all outputs are unconnected, but you can ignore it.

Part B:

Description

In this part of the lab, you will extend the basic MIPS processor you implemented in Part A so that it can execute new instructions. The following table contains a summary of the new instructions you will be required to execute. Details of each instruction and their encodings appear towards the end of this document.

Instructions	Description
JAL	Jump and Link
LUI	Load Upper Immediate
ADD8	Byte-wise addition
RBIT	Reverse bits in a word
REV	Reverse bytes in a word
SADD	Saturating ADD
SSUB	Saturating Subtract

You will be modifying your processor to execute these instructions and testing your modifications using a test program that is provided (the assembly language version of the test program is available at the end of this lab's description and also on Canvas). The test program uses three switches and two buttons from the board to perform certain operations and show certain results on the 7 segment display. The following table summarizes the functions and display modes of the test program.

SW2	SW1	SW0	Task	BTNL	BTNR	Value to Display on 7 segment
0	0	0	ADD8 $2, $4, $5	0	0	Lower 16 bit of $2
				0	1	Upper 16 bit of $2
0	0	1	LUI $2, imm	0	0	Lower 16 bit of $2
				0	1	Upper 16 bit of $2
0	1	0	RBIT $2, $5	0	0	Lower 16 bit of $2
				0	1	Upper 16 bit of $2
0	1	1	REV $2, $4	0	0	Lower 16 bit of $2
				0	1	Upper 16 bit of $2
1	0	0	SADD $2, $5, $5	0	0	Lower 16 bit of $2
				0	1	Upper 16 bit of $2
1	0	1	SSUB $2, $4, $5	0	0	Lower 16 bit of $2
				0	1	Upper 16 bit of $2

Summary of test Program

Three input switches from the board will be used to load a value into register $1. The assembly program will be running in a loop and will be constantly looking at the value in $1. When the value in $1 changes, the program will jump to a subroutine that performs the Task indicated in Table 2. The program will use JAL to jump to subroutines, so you should make sure this instructions works perfectly. The subroutines use $4 and $5 which will be loaded with some constant values. At the end of the subroutine, the program will continue looping, waiting for a change in $1. While the program is looping, you should be able to press BTNL and BTNR in the appropriate combinations to display the value in the result register $2 or $3 on the 7 segment display. The constants that are loaded into $4 and $5 for the computations will be

changed during checkouts to make sure your implementation works. This test program is given to you in assembly in this document after the starter code.

Extra Credit I: Matrix Multiplication Accelerator (30pt)

In this section, you will implement a 3x3 floating point matrix multiplication accelerator for MIPS using the hardware designed in Lab 6 part 1 with additional hardware. The additional hardware includes: 3 buffers, each with 9 locations that can store 1 byte of data to hold matrix A, B, and their product C, and 1 ready bit that is set when the matrix multiplication is done computing (this happens after 7 cycles). 3 instructions should be added to the MIPS ISA: MMLT3 (matrix multiply), STRLD (streaming load), and STRST (streaming store). Consider the two matrices, A and B are stored consecutively in memory in a row-major fashion, meaning the consecutive elements of a row reside next to each other. The numbers are assumed to be in a floating point format with 1 sign bit, 3 exponent bits in biased format, and a 4-bit fraction with an implied 1. A matrix multiplication should be done in the following 3 steps:

First, STRLD is used to load an 18 byte data stream (9 byte from both matrix) from memory specified by $27 into BUF1 & BUF2, 1 contains matrix A and the other contains matrix B. Since $27 is dedicated for address for STRLD and therefore is implicit in the ISA, no operand is needed.

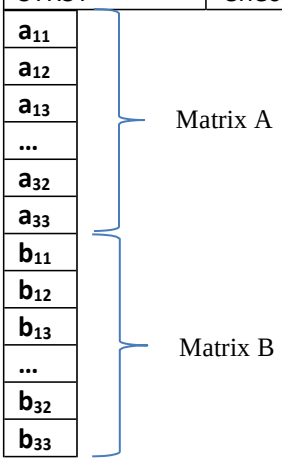
Next, call MMLT3. When a program calls MMLT3, the first thing is to **clear the ready bit** that is (might have been) set by accelerator last time when it was used. Then the accelerator will start computing using the elements pre-loaded into the buffer and store the result into BUF3.

Finally, use STRST to store the 9 byte result from BUF3 into memory specified in $28. When STRST is called, the processor should **wait for the ready bit to be set**. Once the ready bit is set, denoting the multiplication is done, data can be moved from BUF3 into memory.

Note that all 3 instructions have no operands, $27, $28 and associated hardware buffer are implicit. Note: This assumption was made to simplify the task for you. It is possible to create more general instructions where the size of the matrix, the registers used to hold the starting addresses of the buffers, etc are programmable.

Table 3:

STRLD	Load 18 byte of data from mem($27) into BUF1, 2
MMLT3	Clear ready bit, and BUF3 <- BUF1 X BUF2
STRST	Check ready bit, then BUF3 -> mem($28)



Snapshot of the memory containing matrix A & B

Update: please assume 18 bytes of data are stored in 18 memory locations at bit [7:0].

Extra Credit II: Performance Comparison (30pt)

In part 2, you will compare the performance of running matrix multiplication using hardware accelerator in extra credit part I and using software. You need to add following instructions to the CPU design: MULTF8 and ADDF8. MULTF8 performs 8-bit floating point multiplication and ADDF8 performs 8 bit floating point addition.. The numbers are assumed to be in a floating point format with 1 sign bit, 3 exponent bits in biased format, and a 4-bit fraction with an implied 1. Next, you will write a software program that only uses CPU to perform 3x3 floating point matrix multiplication.

Afterwards, please compare the performance of these two methods of doing matrix multiplication by counting how many cycles are used in each method.

One way of measuring performance is to create a cycle counter. This Cycle counter increments at every clock cycle of the cpu. Note that this is the CPU cycle and is likely to be different from the clock cycle used in your floating point multiplier implementation. In this part, you will implement a dedicated **performance counter** that counts cycles taken to do matrix multiplication. The performance counter starts counting when MIPS starts running. At the end of the matrix multiplication, you will use the performance counter to calculate how many cycles are dedicated to matrix multiplication and store this value into the memory location after the last element of the result matrix.

Table 4:

MULTF8 $d, $s1, $s2	Multiply 2 8–bit floating point numbers from $s1 and $s2 and store the result in $d.
ADDF8 $d, $s1, $s2	Add 2 8–bit floating point numbers from $s1 and $s2 and store the result in $d.

Your tasks

1. Modify your processor model from Part A – extend its functionality so that it can execute the instructions summarized in Table 1 (and detailed in Table 5)
2. In order to run the test program, you have to interface three board switches to one of the registers in the register file. Modify your processor such that SW2, SW1, and SW0 map to the LSB three bits of register $1. You are not restricted to the MIPS interfaces when doing this. Register $1 will be used to branch to various sub-routines that will test the new instructions.
3. In order to view the results from the test program, you need to interface two registers to the 7 segment display. As shown in Table 2, register $2 is used for output in most cases except for the HI part of the multiply result. You should write some code that takes BTNL and BTNR as inputs and then displays the upper or lower bytes of $2 or $3 on the 7-segment display as required.
4. Once you have made the necessary modification to your MIPS modules, you will translate the provided assembly language test program to machine code.
5. Initialize your memory using the machine code
6. Synthesize the design and implement it on the board

Extra Credit Tasks

1. Modify your processor model from Part B– extend its functionality so that it can support the matrix multiplication Accelerator in Table 3 and detailed in Table 4.
2. In order to run the test program, you will need three switches implemented in CPU to display the multiplication result. Modify your processor such that when SW2-0 is “110”, you use accelerator for matrix multiplication, and when SW2-0 is “111” you use cpu for matrix multiplication. If you only did extra credit I, you don’t need to implement the case when SW2-0 is “111”.
3. In order to view the results from the test program, you need to interface to the 7 segment display(all four display). You should write a state machine that display elements of resulted matrix sequentially using BTNL and at the end, display cycle count (If you only have extra credit I, you don’t need to display cycle count).
4. Once you have made the necessary modification to your MIPS modules, you will translate the provided assembly language test program to machine code.
5. Initialize your memory using the machine code
6. Synthesize the design and implement it on the board

Useful Information

This part of the lab has only an implementation requirement. However, simulation is recommended to debug your design. If you are unable to implement, be ready with simulation waveforms/do-file/testbench for partial credit.

Submission details

Submit the following things on Canvas:

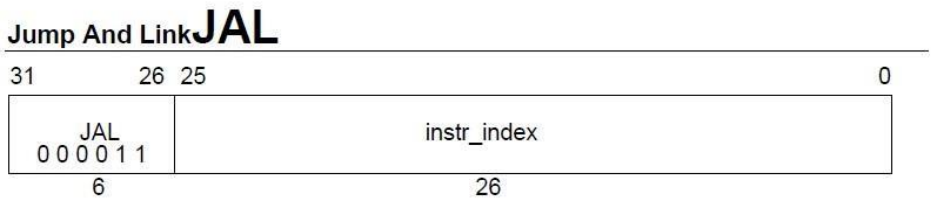
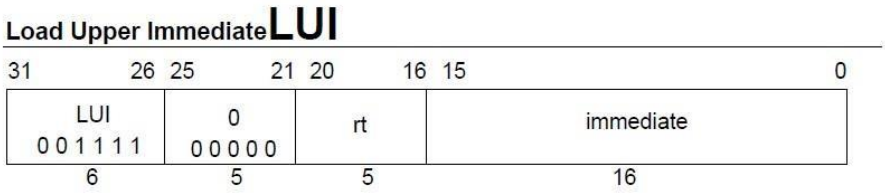
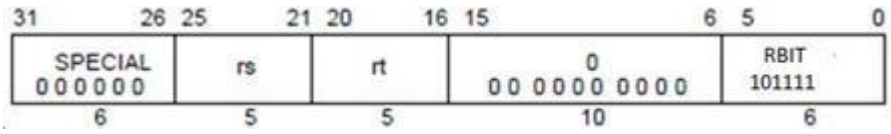
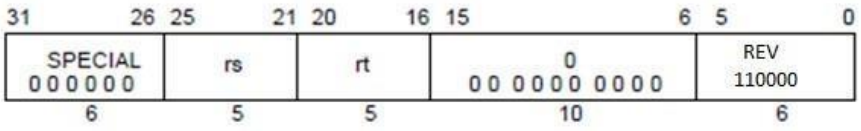
- All Verilog code (modified MIPS and testbenches)
- MIPS assembly program
- Instruction text file containing the machine code of your program
- All Bit file and XDC files

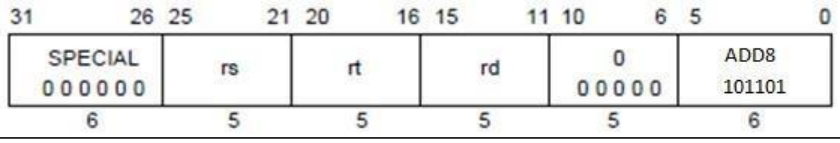
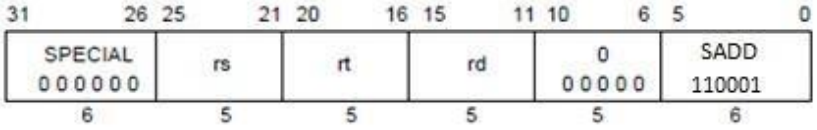
Checkout details

The following things will be checked during check-out: Your modifications to the code and the testbench. Correct functionality of the LED program on the board.

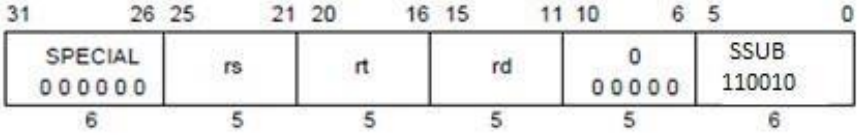
Run the test program that you have loaded into your memory and check that all the switches and buttons give the expected result on the 7-segment display for part B.

Details of New Instructions

JAL	
Format	JAL Target
Description	JAL is used for procedure calls. JAL Target puts the return address (PC+1) in the register $31 and then goes to Target for the next instruction.
Operation	$31 = PC + 1; New_PC = Target; Note: The MIPS in this lab is word addressable
LUI	
Format	LUI $t, imm
Description	The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.
Operation	$rt = imm << 16;
RBIT	
Format	RBIT rs, rt
Description	Reverse the bits in a word
Operation	for(i=0;i<32;i++){ rs[i]=rt[31-i]}
REV	
Format	REV rs, rt
Description	Reverse the bytes in a word
Operation	rs[31:24]=rt[7:0] rs[23:16]=rt[15:8] rs[15:8]=rt[23:16] rs[7:0]=rt[31:24]

ADD8	
Format	ADD8 rd, rs, rt
Description	This perform byte-wise addition as illustrated below
Operation	rd[31:24] = rs[31:24]+rt[31:24] rd[23:16] = rs[23:16]+rt[23:16] rd[15:8] = rs[15:8]+ rt[15:8] rd[7:0] = rs[7:0]+ rt[7:0]
SADD	
Format	SADD rd, rs, rt
Description	Saturating addition

Operation	If ((rs + rt) > 2^32 – 1),
	then rd = 2^32 – 1;
	else rd = rs + rt;

SSUB	
Format	SSUB rd, rs, rt
Description	Saturating Subtraction Operation

Operation	If ((rs - rt) <0),
	then rd = 0;
	else rd = rs - rt;

Extra Credit
Instructions

MMLT3	31:26: 000000 25:6: 00000000000000000000 5:0: 001010
Format	MMLT3
Description	3x3 Matrix Multiplication
Operation	Clear ready bit BUF3 <- BUF1 X BUF2
STRLD	31:26: 100010 25:0: 00000000000000000000000000
Format	STRLD
Description	Streaming load.
Operation	BUF1 <- mem($27) to mem($27+8) Note: 9 byte data to each BUF BUF2 <- mem($27+9) to mem($27+17)
STRST	31:26: 100001 25:0: 00000000000000000000000000
Format	STRST
Description	Streaming store.
Operation	while(!ready) {}; BUF3 -> mem($28) to mem($28+8)
MULTF8	31:26: 000000 25:21 $s1 20:16:$s2 15:11: $d 10:6: 00000 5:0: 110011
Format	MULTF8 $d, $s1, $s2
Description	Byte-wise floating point multiplication.
Operation	rd[31:24] = rs[31:24]*rt[31:24] rd[23:16] = rs[23:16]*rt[23:16] rd[15:8] = rs[15:8]*rt[15:8] rd[7:0] = rs[7:0]*rt[7:0]
ADDF8	31:26: 000000 25:21: $s1 20:16: $s2 15:11: $d 10:6: 00000 5:0: 010010
Format	ADDF8 $d, $s1, $s2
Description	Byte-wise floating point addition.
Operation	rd[31:24] = rs[31:24]+rt[31:24] rd[23:16] = rs[23:16]+rt[23:16] rd[15:8] = rs[15:8]+rt[15:8] rd[7:0] = rs[7:0]+rt[7:0]

Test Program

```
start:
      addi $6, $1, 0
      andi $8, $8, 0
      lui  $4, 28672
      lui  $5, 32767
      ori  $8, $8, 11

loop:


      beq  $6, $1, loop
      addi $6, $1, 0
      sll  $7, $1, 1
      add  $7, $8, $7
      jr   $7
      j    loop

call_table:
      jal operation0
      j loop
      jal operation1
      j loop
      jal operation2
      j loop
      jal operation3
      j loop
      jal operation4
      j loop
      jal operation5
      j loop

operation0:
      add8 $2, $4, $5
      jr $31

operation1:
      lui $2, 4096
      jr $31

operation2:
      rbit $2, $5
      jr $31

operation3:
      rev $2, $4
      jr $31

operation4:
      sadd $2, $5, $5
      jr $31

operation5:
      ssub $2, $4, $5
      jr $31
```