

Lab Assignment #7

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 9-8 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:	<code>readmemh("file name", mem, start addr, end addr)</code>
Store the instruction file in BINARY format:	<code>readmemb("file name", mem, start addr, end addr)</code>

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.

Instruction	Description
JAL	Jump and Link
LUI	Load Upper Immediate
MULT	Multiply Word
MFHI	Move from special register HI
MFLO	Move from special register LO
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	MULT \$4, \$5 MFLO \$2 MFHI \$3	0	0	lower 16 bits of \$2
				0	1	upper 16 bits of \$2
				1	0	lower 16 bits of \$3
				1	1	upper 16 bits of \$3
0	0	1	ADD8 \$2, \$4, \$5	0	0	lower 16 bits of \$2
				0	1	upper 16 bits of \$2
0	1	0	LUI \$2, imm	0	0	lower 16 bits of \$2
				0	1	upper 16 bits of \$2
0	1	1	RBIT \$2, \$5	0	0	lower 16 bits of \$2
				0	1	upper 16 bits of \$2
1	0	0	REV \$2, \$4	0	0	lower 16 bits of \$2
				0	1	upper 16 bits of \$2
1	0	1	SADD \$2, \$5, \$5	0	0	lower 16 bits of \$2
				0	1	upper 16 bits of \$2
1	1	0	SSUB \$2, \$4, \$5	0	0	lower 16 bits of \$2
				0	1	upper 16 bits 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.

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 3)
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

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

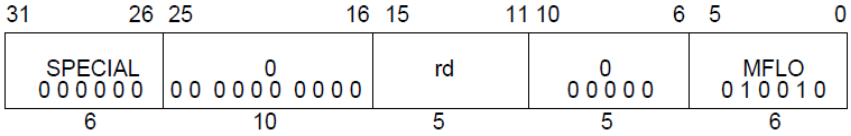
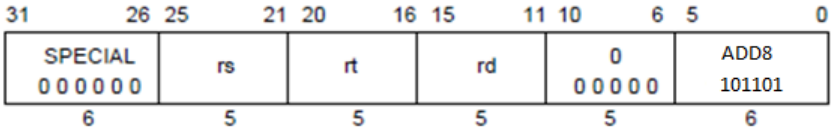
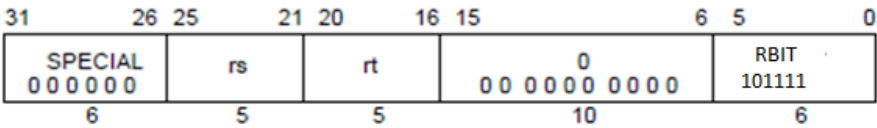
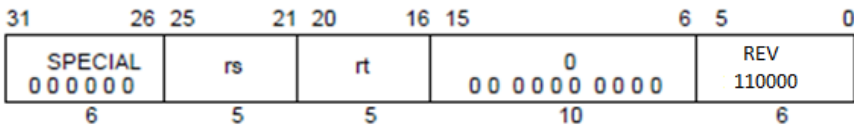
Checkout details

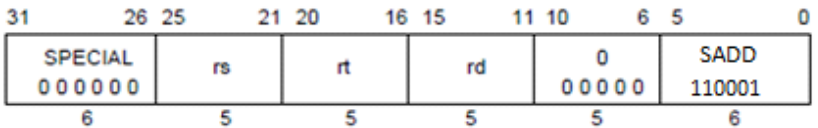
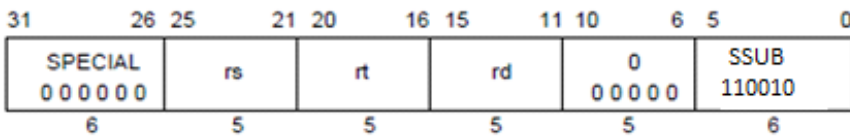
The following things will be checked during check-out:

- 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	Encoding	<div> <div>3126250</div> <div> <div>JAL 000011</div> <div>instr_index</div> </div> <div>626</div> </div>
	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	Encoding	<div> <div>312625212016150</div> <div> <div>LUI 001111</div> <div>0 00000</div> <div>rt</div> <div>immediate</div> </div> <div>65516</div> </div>
	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
MULT	Encoding	<div> <div>31262521201615650</div> <div> <div>SPECIAL 000000</div> <div>rs</div> <div>rt</div> <div>0 0000000000</div> <div>MULT 011000</div> </div> <div>655106</div> </div>
	Format	MULT rs, rt
	Description	The 32-bit word value in reg <i>rt</i> is multiplied by the 32-bit value in reg <i>rs</i> , treating both operands as signed values, to produce a 64-bit result. The low-order 32-bit word of the result is placed into special register <i>LO</i> , and the high-order 32-bit word is placed into special register <i>HI</i> .
	Operation	prod = rs[31:0] * rt[31:0] LO = prod[31:0] HI = prod[63:32]
MFHI	Encoding	<div> <div>31262516151110650</div> <div> <div>SPECIAL 000000</div> <div>0 0000000000</div> <div>rd</div> <div>0 00000</div> <div>MFHI 010000</div> </div> <div>610556</div> </div>
	Format	MFHI rd
	Description	The contents of special register HI are stored in the GPR rd

	Operation	$rd \leftarrow HI$
MFLO	Encoding	 <p>Diagram showing the MFLO instruction encoding. It is a 32-bit word divided into six fields: SPECIAL (6 bits, 000000), rs (10 bits, 0000000000), rd (5 bits), shamt (5 bits, 00000), and MFLO (6 bits, 010010). Bit positions 31, 26, 25, 16, 15, 11, 10, 6, 5, and 0 are indicated above the fields.</p>
	Format	MFLO rd
	Description	The contents of special register LO are stored in the GPR rd
	Operation	$Rd \leftarrow LO$
ADD8	Encoding	 <p>Diagram showing the ADD8 instruction encoding. It is a 32-bit word divided into six fields: SPECIAL (6 bits, 000000), rs (5 bits), rt (5 bits), rd (5 bits), shamt (5 bits, 00000), and ADD8 (6 bits, 101101). Bit positions 31, 26, 25, 21, 20, 16, 15, 11, 10, 6, 5, and 0 are indicated above the fields.</p>
	Format	ADD8 rd, rs, rt
	Description	This perform byte-wise addition as illustrated below
	Operation	$rd[31:24] = rs[31:24] + rt[31:24]$ $rd[23:16] = rs[23:16] + rt[23:16]$ $rd[15:8] = rs[15:8] + rt[15:8]$ $rd[7:0] = rs[7:0] + rt[7:0]$
RBIT	Encoding	 <p>Diagram showing the RBIT instruction encoding. It is a 32-bit word divided into five fields: SPECIAL (6 bits, 000000), rs (5 bits), rt (5 bits), shamt (10 bits, 0000000000), and RBIT (6 bits, 101111). Bit positions 31, 26, 25, 21, 20, 16, 15, 6, 5, and 0 are indicated above the fields.</p>
	Format	RBIT rs, rt
	Description	Reverse the bits in a word
	Operation	$\text{for}(i=0; i<32; i++)\{$ $rs[i] = rt[31-i]\}$
REV	Encoding	 <p>Diagram showing the REV instruction encoding. It is a 32-bit word divided into five fields: SPECIAL (6 bits, 000000), rs (5 bits), rt (5 bits), shamt (10 bits, 0000000000), and REV (6 bits, 110000). Bit positions 31, 26, 25, 21, 20, 16, 15, 6, 5, and 0 are indicated above the fields.</p>
	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]
SADD	Encoding	
	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	Encoding	
	Format	SSUB rd, rs, rt
	Description	Saturating Subtraction
	Operation	if $((rs - rt) < 0)$ then $rd = 0$; else $rd = rs - rt$;

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
    jal operation6
    j loop

operation0:
    mult $4,$5
    mflo $2
    mfhi $3
    jr $31

operation1:
    add8 $2, $4, $5
    jr $31

operation2:
    lui $2, 4096
    jr $31

operation3:
    rbit $2, $5
    jr $31

operation4:
    rev $2, $4
    jr $31

operation5:
    sadd $2, $5, $5
    jr $31

operation6:
    ssub $2, $4, $5
    jr $31
```