MIPS–Lite Verilog Model

EE282 Programming Assignment #1

Due: 18 October, 2001

1.0 Overview

The purpose of this assignment is to familiarize you with Verilog, the MIPS–Lite instruction set architecture, and the MIPS–Lite model. You will be given several test programs and a Verilog model of a MIPS–Lite processor, into which several bugs have been introduced. The first part of the assignment is to fix several (less than 10) bugs in the Verilog model so that the test programs run correctly. The second part of the assignment is to modify the Verilog model so that instructions execute only the stages that they require. Specifically, Branch and Jump instructions should complete in 3 cycles; and, ALU, Store and Jump&Link instructions should complete in 4 cycles.

In general, since NOPs can be implemented in a variety of ways on real machines. you should not explicitly check for NOPs and instead treat them as normal instructions. For example, a NOP is typically represented in MIPS–Lite as a `sll r0, r0, 0`. You should treat this as an ALU instruction which completes in 4 cycles. DO NOT remove the writeback stage for writes to `r0`.

Make sure to check the FAQs in the class homepage regularly for extra information. The assignment is to be done in groups of two or three people. If you have trouble getting the Verilog simulator to work, see one of the TAs.

2.0 What to turn in

Rather than physically hand in a hardcopy of your modified Verilog model, you will be asked to electronically submit your code and a README file. Exact details on how to electronically submit your assignment will be given to you later. The README file should contain a brief description of bug fixes and modifications that you made to the Verilog model.

3.0 Details

3.1 Setup

Before doing anything else, add `source /usr/class/ee282/setup` to your .login file. You need to either relogin or execute "source ~/.login" to activate changes made to .login file.

The files needed for this programming assignment are located in `/usr/class/ee282/proj1.`
In this directory there are two subdirectories: `testcode`, which contains sample test programs and `verilog`, which contains the MIPS–Lite verilog model.

Create your own directory and copy all of the files and the directory structure into it by typing

```
cp -r /usr/class/ee282/proj1 .
```

3.2Compiling Test Programs

Log onto one of the Sun machines. Go to testcode directory under proj1 that you have created. In this directory there are several sample test programs as well as two scripts which compile the test programs for you.

3.2.1 Compiling MIPS assembly test programs

To compile a MIPS assembly program like add.s, type `compile282 add.s` from the testcode directory. The compile282 script will generate the following files:

add.dis – the disassembled object code produced by the assembler
add.data – the user program data_segment used by the verilog model
add.text – the user program text_segment used by the verilog model

Finally, run "useprogram add" to copy the data and text files into the verilog directory.

Remember to put a `jr r31` instruction at the end of your assembly program so that the simulation will terminate correctly. When in doubt, follow the examples provided in the testcode directory.

It is currently not possible to compile "C" files using compile282. We will let you know if we are able to add this feature. However, the files obtained on compiling bubble.c (namely bubble.dis, bubble.data and bubble.text) are available in the testcode directory.

3.3 Running the verilog MIPS–Light model

Verilog is licensed to run on only some of the elaines in Sweet Hall. Since someone else will often be sitting in front of the machines running verilog, you may have to run verilog remotely. To do this type

```
xhost +
```

on your current machine, then rlogin to an elaine that runs verilog and in the terminal window type

```
setenv DISPLAY <yourmachine>:0.0
```

To run the verilog model, change to the verilog directory and type

```
verilog -f master
```

This will compile all of the verilog source files for the MIPS–Light model. There are also three command line arguments which allow you to use other verilog features:

+waves – use the graphical waves display
+regs – use the graphical register display
+output – dump the final contents of memory to a file called memory.core

You can use any combination of these options. Try using all three!

```
verilog -f master +waves +regs +output
```

Once verilog has started up, type `.<return> ;` this will run the most recently compiled program. By clicking on the buttons of the graphical register display, it is possible to step through the execution of the program. You can see what is happening during every cycle and look at the values in all the registers. To exit verilog, type `<Ctrl>D`

4.0 MIPS–Light Debugging Utilities

For this assignment we have added several debugging utilities to the verilog model. These are commands that can be typed at the verilog prompt to enable features or print information to the screen. The following utilities are provided.

Name	Description
`'help`	Help. Prints this list of utility descriptions
`'por`	Power–On Reset
`'ss`	Toggles single step
`'it`	Toggles instruction stream trace
`'waves`	Start the waves display
`'regs`	Start the register display
`'output`	Dumps memory to memory.core
`'rfd`	Dumps the register file
`'break = #;`	Sets a breakpoint in the code
`'start = #;`	Sets the start address for 'dismem and 'dumpmem
`'num = #;`	Sets the number of words to dump for 'dismem and 'dumpmem
`'dismem`	Disassembles 'num' words starting at address 'start'
`'dumpmem`	Dumps 'num' words starting at address 'start'

When you want to rerun a program type `'por` at the verilog command line. This will reinitialize memory and restart the program.
To single step through cycles, type `'ss`. To turn single stepping off, type `'ss` again. `'it` toggles the instruction trace on and off similarly to the `'ss` command.
The `'waves`, `'regs`, and `'output` perform the same function as the `+waves`, `+regs`, and `+output` options except that they may be called after verilog is already running.
With the `'break` command you can set breakpoints in the code. That is if you want the program to stop when it reaches the instruction at 0x100, then type `'break = 'h100;` at the verilog prompt before running the program.
With the last four commands, you can also dump any portion of memory. Say for example you wanted to dump the array in bubble which starts at location 0x50. You would type the following sequence:

```
'start = 'h50;
'num = 6;
'dumpmem;
```

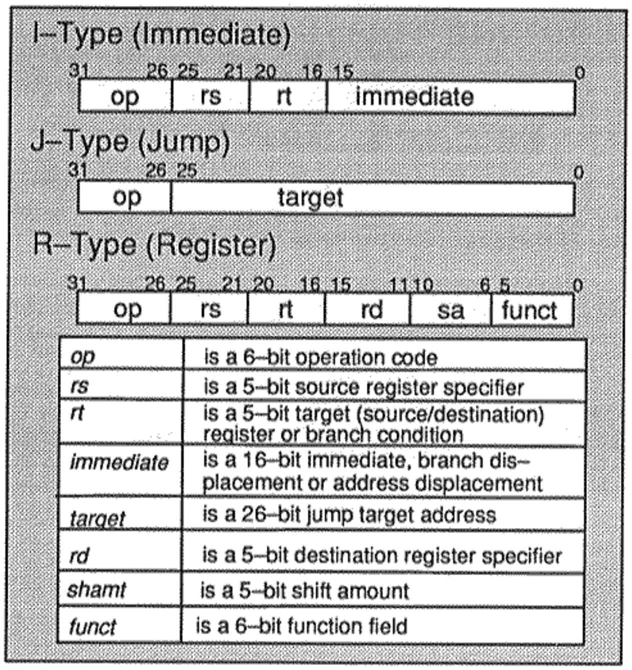
This will dump six words starting at location 0x50. If you use the `'dismem` routine, it will also disassemble memory.

If you ever forget one of the routines, simply type `'help` at the verilog command line.

5.0 MIPS–Light Instruction Set Summary

The MIPS–Light ISA is a stripped down version of the MIPS R2000 ISA which is very close to the DLX ISA that is described in the textbook. The main difference between MIPS–Light ISA and DLX ISA concerns the branch instructions. MIPS–Light allows branches that have equal and not–equal comparisons between any two registers. When one of the registers is R0, the branch instruction is equivalent to a BEQZ or a BNEZ instruction in DLX.

5.0.1 Instruction Formats



In addition to the standard R2000 formats shown above, the MIPS–lite instruction set also has an additional format for the BLTZ and BGEZ instructions.

Bit:	31	25	20	15
Field:	`REGIMM`	`rs`	`sub`	`offset`

5.0.2 Load and Store Instructions

Instruction	Format and Description (op | base | rt | offset)
Load Word	*LW rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Load contents of addressed word into register *rt*.
Store Word	*SW rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Store the contents of register *rt* at addressed location.

5.0.3 ALU Instructions

Instruction	Format and Description (op | rs | rt | immediate)
ADD Immediate	*ADDI rt,rs,immediate* Add 16-bit sign-extended *immediate* to register *rs* and place the 32-bit result in register *rt*. Trap on 2's-complement overflow.
ADD Immediate Unsigned	*ADDIU rt,rs,immediate* Add 16-bit sign-extended *immediate* to register *rs* and place the 32-bit result in register *rt*. Do not trap on overflow.
Set on Less Than Immediate	*SLTI rt,rs,immediate* Compare 16-bit sign-extended *immediate* with register *rs* as signed 32-bit integers. Result = 1 if *rs* is less than *immediate*; otherwise result = 0. Place result in register *rt*.
Set on Less Than Immediate Unsigned	*SLTIU rt,rs,immediate* Compare 16-bit sign-extended *immediate* with register *rs* as unsigned 32-bit integers. Result = 1 if *rs* is less than *immediate*; otherwise result = 0. Place result in register *rt*.
AND Immediate	*ANDI rt,rs,immediate* Zero-extend 16-bit *immediate*, AND with contents of register *rs* and place the result in register *rt*.
OR Immediate	*ORI rt,rs,immediate* Zero-extend 16-bit *immediate*, OR with contents of register *rs* and place the result in register *rt*.
Exclusive OR Immediate	*XORI rt,rs,immediate* Zero-extend 16-bit *immediate*, exclusive OR with contents of register *rs* and place the result in register *rt*.
Load Upper Immediate	*LUI rt,immediate* Shift 16-bit *immediate* left 16 bits. Set least significant 16 bits of word to zeros. Store the result in register *rt*.

Instruction	Format and Description (op | rs | rt | rd | sa | function)
Shift Left Logical	*SLL rd,rt,sa* Shift the contents of register *rt* left by *sa* bits, inserting zeros into the low order bits. Place the 32-bit result in register *rd*.
Shift Right Logical	*SRL rd,rt,sa* Shift the contents of register *rt* right by *sa* bits, inserting zeros into the high order bits. Place the 32-bit result in register *rd*.
Shift Right Arithmetic	*SRA rd,rt,sa* Shift the contents of register *rt* right by *sa* bits, sign-extending the high order bits. Place the 32-bit result in register *rd*.
Shift Left Logical Variable	*SLLV rd,rt,rs* Shift the contents of register *rt* left. The low order 5 bits of register *rs* specify the number of bits to shift left; insert zeros into the low order bits of *rt* and place the 32-bit result in register *rd*.
Shift Right Logical Variable	*SRLV rd,rt,rs* Shift the contents of register *rt* right. The low order 5 bits of register *rs* specify the number of bits to shift right; insert zeros into the high order bits of *rt* and place the 32-bit result in register *rd*.
Shift Right Arithmetic Variable	*SRAV rd,rt,rs* Shift the contents of register *rt* right. The low order 5 bits of register *rs* specify the number of bits to shift right; sign-extend the high order bits of *rt* and place the 32-bit result in register *rd*.
Subtract Unsigned	*SUBU rd,rs,rt* Subtract contents of registers *rt* from *rs* and place the 32-bit result in register *rd*. Do not trap on overflow.
Set on Less Than	*SLT rd,rs,rt* Compare contents of register *rt* to register *rs* as signed 32-bit integers. Result = 1 if *rs* is less than *rt*; otherwise result = 0.
Set on Less Than Unsigned	*SLTU rd,rs,rt* Compare contents of register *rt* to register *rs* as unsigned 32-bit integers. Result = 1 if *rs* is less than *rt*; otherwise result = 0.
AND	*AND rd,rs,rt* Bitwise AND the contents of registers *rs* and *rt*, and place the result in register *rd*.
OR	*OR rd,rs,rt* Bitwise OR the contents of registers *rs* and *rt*, and place the result in register *rd*.
Exclusive OR	*XOR rd,rs,rt* Bitwise exclusive OR the contents of registers *rs* and *rt*, and place the result in register *rd*.
NOR	*NOR rd,rs,rt* Bitwise NOR the contents of registers *rs* and *rt*, and place the result in register *rd*.

5.0.4 Jump and Branch Instructions

Note: correct format is *JALR rd, rs.*

Instruction	Format and Description	Encoding
Branch on Equal	*BEQ rs,rt,offset* Branch to target address if register *rs* is equal to register *rt*.	op, rs, rt, offset
Branch on Not Equal	*BNE rs,rt,offset* Branch to target address if register *rs* is not equal to register *rt*.	
Branch on Less than or Equal Zero	*BLEZ rs,offset* Branch to target address if register *rs* is less than or equal to zero.	
Branch on Greater Than Zero	*BGTZ rs,offset* Branch to target address if register *rs* is greater than zero.	
Branch on Less Than Zero	*BLTZ rs,offset* Branch to target address if register *rs* is less than zero.	REGIMM, rs, sub, offset
Branch on Greater than or Equal Zero	*BGEZ rs,offset* Branch to target address if register *rs* is greater than or equal to zero.	
Branch on Less Than Zero And Link	*BLTZAL rs,offset* Place address of instruction following the delay slot in register *r31* (Link register). Branch to target address if register *rs* is less than zero.	
Branch on Greater than or Equal Zero And Link	*BGEZAL rs,offset* Place address of instruction following the delay slot in register *r31* (Link register). Branch to target address if register *rs* is greater than or equal to zero.	

Jump	Shift the 26-bit *target* address left two bits, combine with high order four bits of the PC, and jump to the address with a 1-instruction delay.
Jump And Link	*JAL target* Shift the 26-bit *target* address left two bits, combine with high order four bits of the PC, and jump to the address with a 1-instruction delay. Place the address of the instruction following the delay slot in *r31* (*Link* register).

Instruction	Format and Description	Encoding
		op, rs, rt, rd, sa, function
Jump Register	*JR rs* Jump to the address contained in register *rs*, with a 1-instruction delay.	
Jump And Link Register	*JALR rs, rd* Jump to the address contained in register *rs*, with a 1-instruction delay. Place the address of the instruction following the delay slot in register *rd*.	