

Project 1: Simple Verilog Model (5%)

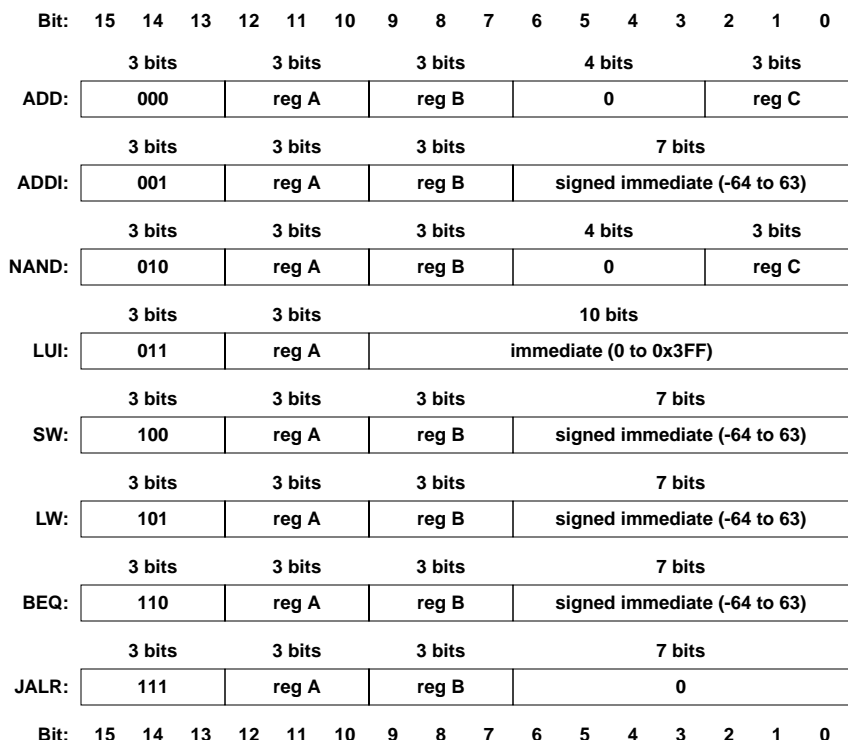
ENEE 446: Digital Computer Design

1. Purpose

The purpose of this assignment is to learn the rudiments of the Verilog hardware description language in the context of a processor design. Some of the the most important concepts you will learn are those of *non-blocking assignments* and *concurrency*. Non-blocking assignments are specific to the Verilog language; concurrency is a powerful concept that shows up at all levels of processor design. The processor model will be a simple sequential implementation—on every cycle, you will execute an instruction and update the program counter accordingly.

2. RiSC-16 Instruction Set

Before we talk about Verilog implementation, we have to talk about the instruction-set architecture (ISA). This section describes the ISA of the 16-bit Ridiculously Simple Computer (RiSC-16), a teaching ISA that is based on the Little Computer (LC-896) developed by Peter Chen at the University of Michigan. The RiSC-16 is an 8-register, 16-bit computer. All addresses are shortword-addresses (i.e. address 0 corresponds to the first two bytes of main memory, address 1 corresponds to the second two bytes of main memory, etc.). Like the MIPS instruction-set architecture, by hardware convention, register 0 will always contain the value 0. The machine enforces this: reads to register 0 always return 0, irrespective of what has been written there. The RiSC-16 is very simple, but it is general enough to solve complex problems. There are three machine-code instruction formats and a total of 8 instructions. They are illustrated in the figure below.



The following table describes the different instruction operations.

Mnemonic	Name and Format	Opcode (binary)	Assembly Format	Action
add	Add RRR-type	000	add rA, rB, rC	Add contents of regB with regC , store result in regA .
addi	Add Immediate RRI-type	001	addi rA, rB, imm	Add contents of regB with imm , store result in regA .
nand	Nand RRR-type	010	nand rA, rB, rC	Nand contents of regB with regC , store results in regA .
lui	Load Upper Immediate RI-type	011	lui rA, imm	Place the top 10 bits of the 16-bit imm into the top 10 bits of regA , setting the bottom 6 bits of regA to zero.
sw	Store Word RRI-type	101	sw rA, rB, imm	Store value from regA into memory. Memory address is formed by adding imm with contents of regB .
lw	Load Word RRI-type	100	lw rA, rB, imm	Load value from memory into regA . Memory address is formed by adding imm with contents of regB .
beq	Branch If Equal RRI-type	110	beq rA, rB, imm	If the contents of regA and regB are the same, branch to the address $PC+1+imm$, where PC is the address of the beq instruction.
jalr	Jump And Link Register RRI-type	111	jalr rA, rB	Branch to the address in regB . Store $PC+1$ into regA , where PC is the address of the jalr instruction.

3. RiSC-16 Assembly Language and Assembler

The RiSC-16 assembler is called “a” and comes as a SPARC executable. Also included is the assembler source code should you wish to recompile for some other architecture (e.g. x86). Valid RiSC labels are any combination of letters and numbers followed by a colon. The colon at the end is not optional—a label without a colon is interpreted as an opcode. After the optional label is the opcode field. All register-value fields are given as **decimal** numbers, optionally preceded by the letter ‘r’ ... as in r0, r1, r2, etc. Immediate-value fields are given in either decimal, octal, or hexadecimal form. Octal numbers are preceded by the character ‘0’ (zero). For example, 032 is interpreted as the octal number ‘oh-three-two’ which corresponds to the decimal number 26. Hexadecimal numbers are preceded by ‘0x’ (oh-x). For example, 0x12 corresponds to the decimal number 18. For those of you who know the C programming language, you should be perfectly at home. The following table describes the instructions.

	Assembly-Code Format	Meaning
add	regA, regB, regC	$R[\text{regA}] \leftarrow R[\text{regB}] + R[\text{regC}]$
addi	regA, regB, immed	$R[\text{regA}] \leftarrow R[\text{regB}] + \text{immed}$
nand	regA, regB, regC	$R[\text{regA}] \leftarrow \sim(R[\text{regB}] \& R[\text{regC}])$
lui	regA, immed	$R[\text{regA}] \leftarrow \text{immed} \& 0\text{xffc0}$
sw	regA, regB, immed	$R[\text{regA}] \rightarrow \text{Mem}[R[\text{regB}] + \text{immed}]$
lw	regA, regB, immed	$R[\text{regA}] \leftarrow \text{Mem}[R[\text{regB}] + \text{immed}]$
beq	regA, regB, immed	<pre> if (R[regA] == R[regB]) { PC <- PC + 1 + immed (if label, PC <- label) } </pre>
jalr	regA, regB	$PC \leftarrow R[\text{regB}], R[\text{regA}] \leftarrow PC + 1$

Anything after a pound sign (`#`) is considered a *comment* and is ignored. The comment field ends at the end of the line. Comments are vital to creating understandable assembly-language programs, because the instructions themselves are rather cryptic.

In addition to RiSC-16 instructions, an assembly-language program may contain directives for the assembler. These are often called *pseudo-instructions*. The six assembler directives we will use are **nop**, **halt**, **lli**, **movi**, **.fill**, and **.space** (note the leading periods for **.fill** and **.space**, which simply signifies that these represent data values, not executable instructions).

Assembly-Code Format		Meaning
<code>nop</code>		do nothing
<code>halt</code>		stop machine & print state
<code>lli</code>	<code>regA, immed</code>	$R[\text{regA}] \leftarrow R[\text{regA}] + (\text{imm} \& 0x3f)$
<code>movi</code>	<code>regA, immed</code>	$R[\text{regA}] \leftarrow \text{imm}$
<code>.fill</code>	<code>imm</code>	initialized data with value <i>imm</i>
<code>.space</code>	<code>imm</code>	zero-filled data array of size <i>imm</i>

The following paragraphs describe these pseudo-instructions in more detail:

- The **nop** pseudo-instruction means “do not do anything this cycle” and is replaced by the instruction **add 0,0,0** (which clearly does nothing).
- The **halt** pseudo-instruction means “stop executing instructions and print current machine state” and is replaced by **jlr 0, 0** with a non-zero immediate field. This is described in more detail in the documents *The Pipelined RiSC-16* and *An Out-of-Order RiSC-16*, in which HALT is a subset of syscall instructions for the purposes of handling interrupts and exceptions: any JLR instruction with a non-zero immediate value uses that immediate as a syscall opcode. This allows such instructions as syscall, halt, return-from-exception, etc.
- The **lli** pseudo-instruction (*load-lower-immediate*) means “OR the bottom six bits of this number into the indicated register” and is replaced by **addi X,X,imm6**, where **X** is the register specified, and **imm6** is equal to **imm & 0x3f**. This instruction can be used in conjunction with **lui**: the **lui** first moves the top ten bits of a given number (or address, if a label is specified) into the register, setting the bottom six bits to zero; the **lli** moves the bottom six bits in. The six-bit number is guaranteed to be interpreted as positive and thus avoids sign-extension; therefore, the resulting **addi** is essentially a concatenation of the two bitfields.
- The **movi** pseudo-instruction is just shorthand for the **lui+lli** combination. Note, however, that the **movi** instruction *looks* like it only represents a single instruction, whereas in fact it represents two. This can throw off your counting if you are expecting a certain distance between instructions. Thus, it is always a good idea to use labels wherever possible.
- The **.fill** directive tells the assembler to put a number into the place where the instruction would normally be stored. The **.fill** directive uses one field, which can be either a numeric value or a symbolic address. For example, “**.fill 32**” puts the value 32 where the instruction would normally be stored. Using **.fill** with a symbolic address will store the address of the label. In the example below, the line “**.fill start**” will store the value 2, because the label “**start**” refers to address 2.
- The **.space** directive takes one integer **n** as an argument and is replaced by **n** copies of “.fill 0” in the code; i.e., it results in the creation of **n** 16-bit words all initialized to zero.

In general, acceptable RiSC assembly code is one-instruction-per-line. It **is** okay to have a line that is blank, whether it is commented out (i.e., the line begins with a pound sign) or not (i.e., just a blank line). However, a label **cannot** appear on a line by itself; it must be followed by a valid instruction on the same line (a **.fill** directive or **halt/nop/etc** counts as an instruction).

4. Verilog Implementation

You have been given a skeleton Verilog file that looks like this:

```
//
// RiSC-16 skeleton
//
`define ADD          3'd0
`define ADDI        3'd1
`define NAND        3'd2
`define LUI         3'd3
`define SW          3'd4
`define LW          3'd5
`define BEQ         3'd6
`define JALR        3'd7
`define EXTEND      3'd7

`define INSTRUCTION_OP 15:13 // opcode
`define INSTRUCTION_RA 12:10 // rA
`define INSTRUCTION_RB 9:7 // rB
`define INSTRUCTION_RC 2:0 // rC
`define INSTRUCTION_IM 6:0 // immediate (7-bit)
`define INSTRUCTION_LI 9:0 // large immediate (10-bit, 0-extended)
`define INSTRUCTION_SB 6 // immediate's sign bit

`define FORW_BRANCH 1'b0
`define BACK_BRANCH 1'b1

`define ZERO 16'd0
`define HALTINSTRUCTION { `EXTEND, 3'd0, 3'd0, 3'd7, 4'd1 }

module RiSC (clk);
    input        clk;

    reg [15:0] rf[0:7];
    reg [15:0] pc;
    reg [15:0] m[0:65535];

    initial begin
        pc = 0;
        rf[0] = `ZERO;
        rf[1] = `ZERO;
        rf[2] = `ZERO;
        rf[3] = `ZERO;
        rf[4] = `ZERO;
        rf[5] = `ZERO;
        rf[6] = `ZERO;
        rf[7] = `ZERO;
        for (j=0; j<65536; j=j+1)
            m[j] = 0;
    end

    always @(negedge clk) begin
        rf[0] <= `ZERO;
    end

endmodule
```

It contains a number of definitions that will be helpful. For instance, the top group of definitions are the various instruction opcodes. The second group are fields of the instruction, such that the following statement:

```
instr[ `INSTRUCTION_OP ];
```

yields the opcode of the instruction.

The HALTINSTRUCTION definition allows you to decide when to halt; when you encounter an instruction that matches this value, you can either \$stop (which exits to the simulator debugger level) or \$finish (which exits to the UNIX shell).

The RiSC module contains the definition of the CPU core. This is the module that you will implement. So far it contains only the registers, program counter, and memory. These are all initialized to hold the 16-bit value zero by the “initial” block. This block executes before all others at the time of the simulator start-up. The “always” block sets register zero to the value 0 on the negative edge of the clock. All processor activity should be driven on the positive edge of the clock, so that this statement will undo any changes to register zero before the next clock.

Finally, the input to the RiSC module is the clock signal, which indicates that the module is not free-standing. It must be instantiated elsewhere to run. That is the function of test modules. You have also been given a file called “test.v” which instantiates the RiSC ... it looks like this:

```

module top ();
    reg    clk;
    RiSC   cpu(clk);
    integer j;

    initial begin
        #1 $readmemh("init.dat", cpu.m);
        #1000 $stop;
    end

    always begin
        #5 clk = 0;
        #5 clk = 1;

        $display("Register Contents:");
        $display(" r0 - %h", cpu.rf[0]);
        $display(" r1 - %h", cpu.rf[1]);
        $display(" r2 - %h", cpu.rf[2]);
        $display(" r3 - %h", cpu.rf[3]);
        $display(" r4 - %h", cpu.rf[4]);
        $display(" r5 - %h", cpu.rf[5]);
        $display(" r6 - %h", cpu.rf[6]);
        $display(" r7 - %h", cpu.rf[7]);
    end

endmodule

```

The module instantiates a copy of the RiSC module and feeds it a clock signal. It also prints out some of the RiSC’s internal state—note the naming convention used to get at the RiSC’s internal variables. Just as in the RiSC module, the “initial” block executes before all else, where it initializes the RiSC’s memory. The \$readmemh call tells the simulator to overwrite the memory system with the contents of the file “init.dat” ... this is done at time t=1, which should occur after the RiSC has set all its internal memory locations to the value 0. Then the initial block tells itself to halt execution 100 cycles into the future.

5. Running Your Project

First, tap verilog to get access to the simulators. Then you can invoke your code this way:

```
verilog test.v RiSC.v
```

or

```
ncverilog test.v RiSC.v
```

The ncverilog simulator has a longer start-up time but executes much faster once it gets going.

When you submit your code to me, all I want is the RiSC.v code ... i.e. everything but the test.v file (I will use my own). Do not change any of the variable names within the RiSC.v file, for hopefully obvious reasons.