



Project I: Assembler & Simple Verilog CPU (10%)

ENEE 646: Digital Computer Design, Fall 2008

Assigned: Tuesday, Sep 2; Due: Tuesday, Sep 16

Purpose

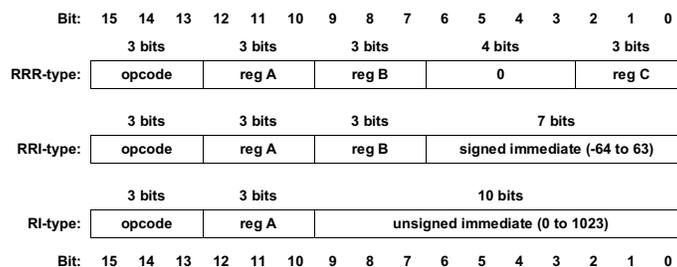
This assignment has two primary goals: to teach you the rudiments of the Verilog hardware description language and to show you how software assembles programs into machine-level code. The project has two parts. For the first part, you will write a C-language program that reads in an assembly-language program and produces the corresponding machine code. For the second part, you will write a Verilog-language behavioral simulator for arbitrary machine code. In writing an assembler, you will learn about file I/O and how CPUs interpret numbers as instructions. When writing your Verilog program, you will learn about 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.

RiSC-16 Instruction Set

Before we talk about the C-language assembler and Verilog-language CPU implementations, we have to talk about the instruction-set architecture (ISA) that you are working with. This section describes the ISA of the 16-bit Ridiculously Simple Computer (RiSC-16), a teaching ISA 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 and has a total of 8 16-bit instructions. The instructions are illustrated in the figure below.

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	3 bits			3 bits			3 bits			4 bits			3 bits			
ADD:	000			reg A			reg B			0			reg C			
	3 bits			3 bits			3 bits			7 bits						
ADDI:	001			reg A			reg B			signed immediate (-64 to 63)						
	3 bits			3 bits			3 bits			4 bits			3 bits			
NAND:	010			reg A			reg B			0			reg C			
	3 bits			3 bits			10 bits									
LUI:	011			reg A			immediate (0 to 0x3FF)									
	3 bits			3 bits			3 bits			7 bits						
SW:	100			reg A			reg B			signed immediate (-64 to 63)						
	3 bits			3 bits			3 bits			7 bits						
LW:	101			reg A			reg B			signed immediate (-64 to 63)						
	3 bits			3 bits			3 bits			7 bits						
BNE:	110			reg A			reg B			signed immediate (-64 to 63)						
	3 bits			3 bits			3 bits			7 bits						
JALR:	111			reg A			reg B			0						
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

There are 3 machine-code instruction formats: RRR-type, RRI-type, and RI-type.



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. 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	100	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	101	lw rA, rB, imm	Load value from memory into regA. Memory address is formed by adding imm with contents of regB.
bne	Branch if Not Equal RRI-type	110	bne rA, rB, imm	If the contents of regA and regB are not the same, branch to the address PC+I+imm, where PC is the address of the bne instruction.
jalr	Jump And Link Register RRI-type	111	jalr rA, rB	Branch to the address in regB. Store PC+I into regA, where PC is the address of the jalr instruction.

Note that the 8 basic instructions of the RiSC-16 architecture form a complete ISA that can perform arbitrary computation. For example:

- **Moving constant values into registers.** The number 0 can be moved into any register in one cycle (add rX r0 r0). Any number between -64 and 63 can be placed into a register in one operation using the ADDI instruction (addi rX r0 number). Moreover, any 16-bit number can be moved into a register in two operations (lui+addi).
- **Subtracting numbers.** Subtracting is simply adding the negative value. Any number can be made negative in two instructions by flipping its bits and adding 1. Bit-flipping can be done by NANDing the value with itself; adding 1 is done with the ADDI instruction. Therefore, subtraction is a three-instruction process. Note that without an extra register, it is a destructive process.
- **Multiplying numbers.** Multiplication is easily done by repeated addition, bit-testing, and left-shifting a bitmask by one bit (which is the same as an addition with itself).

RiSC Assembly Language and Assembler

The first half of this project is to write a program to take an assembly-language program and translate it into machine-level code. You will translate assembly language names for instructions, such as `bne`, into their numeric equivalent (e.g. 110), and you will translate symbolic names for addresses into numeric values. The final output produced by the assembler will be a series of 16-bit instructions, written as ASCII text, hexadecimal format, one line per instruction (i.e. each number separated by newlines).

The format for a line of assembly code is:

```
label:<whitespace>opcode<whitespace>field0, field1, field2<whitespace># comments
```

The leftmost field on a line is the label field. Valid RiSC labels are any combination of letters and numbers followed by a colon. Labels make it much easier to write assembly language programs, since otherwise you would need to modify all address fields each time you added a line to your assembly-language program! The colon at the end of the label is not optional—a label without a colon is interpreted as an opcode. After the optional label is whitespace (space/s or tab/s). Then follows the opcode field, where the opcode can be any of the assembly-language instruction mnemonics listed in the above table. After more whitespace comes a series of fields separated by commas and possibly whitespace (the whitespace is not necessary; the commas are). For the purposes of this assignment, all fields are to be given as **decimal** numbers (this way, you do not have to deal with hexadecimal formats, etc.). The number of fields depends on the instruction. Some instructions will have three fields; others two fields; while some will have no fields beyond the opcode. 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. The following table describes the behavior of the instructions in C-like syntax.

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

In addition to RiSC-16 instructions, an assembly-language program may contain directives for the assembler. These are often called pseudo-instructions. The assembler directives we will use are `nop`, `halt`, and `.fill` (note the leading period for `.fill`, which simply signifies that this pseudo-instruction represents a data value, not an executable instruction).

Assembly-Code Format	Meaning
<code>nop</code>	do nothing
<code>halt</code>	stop machine & print state
<code>.fill immed</code>	initialized data with value <i>immed</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 & print current machine state” and is replaced by **jalr 0, 0** with a non-zero immediate having the value 113 decimal (chosen for historical reasons). Therefore the **halt** instruction is the hexadecimal number 0xE071.
- 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 (in decimal, hexadecimal, or octal) or a symbolic address (i.e. a label). For example, “.fill 32” puts the value 32 where the instruction would normally be stored; “.fill 0x10” also puts the value 32 (decimal) where the instruction would normally be stored in the assembler provided). 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.

When labels are used as part of instructions (e.g. the *immediate* values for **lw**, **sw**, **.fill**, **lui**, **bne**, and even **addi** instructions), they refer to the value of the label (the address at which the label occurs), and they are interpreted slightly differently depending on the instruction in which they are found:

- For **lw**, **sw**, **lui**, or **.fill** instructions, the assembler should compute the immediate value to be equal to the address of the label. Therefore if a label appears in an immediate field, it should be interpreted as the address at which the label is found. In the case of **lw** or **sw**, this could be used with a zero base register to refer to the label, or could be used with a non-zero base register to index into an array starting at the label.
- Labels are slightly different for **bne** instructions: the assembler should not use the label’s address directly but instead should determine the numeric immediate value needed to branch to that label. The **bne** description shows the program counter being updated by the immediate value ($PC \leftarrow \text{label}$), but the address represented by the label does **not** go directly into the instruction. The instruction format specifies that the immediate field *must contain a PC-relative value*. Therefore, if a **label** is used in the assembly program, the assembler must *calculate* the PC-relative value that will accomplish the same as a jump to the address of that label. This amounts to solving the following equation for *offset*: $\text{label} = PC + 1 + \text{offset}$.

The assembler makes two passes over the assembly-language program. In the first pass, it calculates the address for every symbolic label. You may assume that the first instruction is located at address 0. In the second pass, the assembler generates a machine-level instruction in ASCII hexadecimal for each line of assembly language. For example, the following is an assembly-language program that counts down from 5, stopping when it hits 0.

```

                lw    1,0,count    # load reg1 with 5 (uses symbolic address)
                lw    2,1,2        # load reg2 with -1 (uses numeric address)
start:         add    1,1,2        # decrement reg1 (could have been "addi 1, 1, -1")
                bne   0,1,start    # go back to the beginning of the loop unless reg1==0
done:         halt                    # end of program
                nop
count:        .fill 5
neg1:        .fill -1
startAddr:   .fill start           # will contain the address of start (2)

```

And here is the corresponding machine-level program (note the absence of “0x” characters):

```

a406
a882
0482
c0fe
e071
0000
0005
ffff
0002

```

Be sure you understand how the above assembly-language program got translated to this machine-code file.

Running Your Assembler

You must write your program so that it is run as follows (assuming your program name is “assemble”).

```
assemble assembly-code-file machine-code-file
```

Note that the format for running the command must use command-line arguments for the file names (rather than standard input and standard output). The first argument is the file name where the assembly-language program is stored, and the second argument is the file name where the output (the machine-code) is written. Your program should only store the list of hexadecimal numbers in the machine-code file, one instruction per line—any other format will render your machine-code file ungradable. Each number can have ‘0x’ in front or not, as you wish. Any other output that you want the program to generate (e.g. debugging output) can be printed to *stdout* or *stderr*.

Error Checking

Your assembler should catch errors in the assembly language program, as well as errors that occur because the user ran your program incorrectly (e.g. with only 1 argument instead of 2 arguments). For example, it should detect the use of undefined labels, duplicate labels, missing arguments to opcodes (e.g. only giving two fields to *lw*), immediate values that are out of range, unrecognized opcodes, etc.

Code Fragment for Assembler

The focus of this class is machine organization, not C programming skills. To help you, here is a fragment of the C program for the assembler. This shows how to specify command-line arguments to the program (via *argc* and *argv*), how to parse the assembly-language file, etc. This fragment is provided strictly to help you, though it may take a bit for you to understand and use the file. You may also choose to not use this fragment.

```
/* Assembler code fragment for RiSC */

#include <stdio.h>
#include <string.h>

#define MAXLINELENGTH 1000

char * readAndParse(FILE *inFilePtr, char *lineString,
char **labelPtr, char **opcodePtr, char **arg0Ptr,
char **arg1Ptr, char **arg2Ptr)
{
    /* read and parse a line
    note that lineString must point to allocated memory, so that *labelPtr,
    *opcodePtr, and *argXPtr won't be pointing to readAndParse's memory
    note also that *labelPtr, *opcodePtr, and *argXPtr
    point to memory locations in lineString.
    When lineString changes, so will *labelPtr, *opcodePtr, and *argXPtr.
    function returns NULL if at end-of-file */

    char *statusString, *firsttoken;

    statusString = fgets(lineString, MAXLINELENGTH, inFilePtr);

    if (statusString != NULL) {
        firsttoken = strtok(lineString, " \t\n");
        if (firsttoken == NULL || firsttoken[0] == '#') {
            return readAndParse(inFilePtr, lineString, labelPtr,
            opcodePtr, arg0Ptr, arg1Ptr, arg2Ptr);
        } else if (firsttoken[strlen(firsttoken) - 1] == ':') {
            *labelPtr = firsttoken;
            *opcodePtr = strtok(NULL, " \n");
            firsttoken[strlen(firsttoken) - 1] = '\0';
        } else {
            *labelPtr = NULL;
            *opcodePtr = firsttoken;
        }
    }
}
```

```

        *arg0Ptr = strtok(NULL, " , \t\n");
        *arg1Ptr = strtok(NULL, " , \t\n");
        *arg2Ptr = strtok(NULL, " , \t\n");
    }
    return(statusString);
}

int isNumber(char *string) {
    /* return 1 if string is a number */
    int i;
    return( (sscanf(string, "%d", &i) == 1) );
}

main(int argc, char *argv[])
{
    char *inFileString, *outFileString;
    FILE *inFilePtr, *outFilePtr;
    char *label, *opcode, *arg0, *arg1, *arg2;
    char lineString[MAXLINELENGTH+1];

    if (argc != 3) {
        printf("error: usage: %s <assembly-code-file>
               <machine-code-file>\n", argv[0]);
        exit(1);
    }
    inFileString = argv[1];
    outFileString = argv[2];
    inFilePtr = fopen(inFileString, "r");
    if (inFilePtr == NULL) {
        printf("error in opening %s\n", inFileString);
        exit(1);
    }
    outFilePtr = fopen(outFileString, "w");
    if (outFilePtr == NULL) {
        printf("error in opening %s\n", outFileString);
        exit(1);
    }

    /* here is an example for how to use readAndParse to read a line from inFilePtr */
    if (readAndParse(inFilePtr, lineString, &label, &opcode,
                    &arg0, &arg1, &arg2) == NULL) {
        /* reached end of file */
    } else {
        /* label is either NULL or it points to
           a null-terminated string in lineString
           that has the label. If label is NULL,
           that means the label field didn't exist.
           Same for opcode and argX. */
    }

    /* this is how to rewind file ptr to start reading from beginning of file */
    rewind(inFilePtr);

    /* after a readAndParse, you may want to do the following to test each opcode */
    if (!strcmp(opcode, "add")) {
        /* do whatever you need to do for opcode "add" */
    }
}

```

C Programming Tips

Here are a few programming tips for writing C programs to manipulate bits and then print them out:

1. To indicate a hexadecimal constant in C, precede the number by 0x. For example, 27 decimal is 0x1b in hexadecimal.
2. The value of the expression $(a \gg b)$ is the number a shifted right by b bits. Neither a nor b are changed. E.g. $(25 \gg 2)$ is 6. Note that 25 is 11001 in binary, and 6 is 110 in binary.
3. The value of the expression $(a \ll b)$ is the number a shifted left by b bits. Neither a nor b are changed. E.g. $(25 \ll 2)$ is 100. Note that 25 is 11001 in binary, and 100 is 1100100 in binary.
4. $\sim a$ is the bit-wise complement of a (a is not changed); if $a = 100101$, $\sim a = 011010$.

5. The value of the expression $(a \& b)$ is a logical AND on each bit of a and b (i.e. bit 15 of a ANDed with bit 15 of b , bit 14 of a ANDed with bit 14 of b , etc.). E.g. $(25 \& 11)$ is 9, since:

```

  11001 (binary)
& 01011 (binary)
-----
= 01001 (binary),

```

which is 9 decimal.

6. The value of the expression $(a | b)$ is a logical OR on each bit of a and b (i.e. bit 15 of a ORed with bit 15 of b , bit 14 of a ORed with bit 14 of b , etc.). E.g. $(25 | 11)$ is 27, since:

```

  11001 (binary)
| 01011 (binary)
-----
= 11011 (binary),

```

which is 27 decimal.

Use these operations to create and manipulate machine-code. E.g. to look at bit 3 of the variable a , you might do: $(a >> 3) \& 0x1$. To look at bits 15-13 of a 16-bit word (for instance, the opcode of each instruction), you could do: $(a >> 13) \& 0x7$. To put a 6 into bits 5-3 and a 3 into bits 2-1, you could do the following: $(6 << 3) | (3 << 1)$. If you're not sure what an operation is doing, print some intermediate results to help you debug.

Beware, however, that `printf` expects you to tell it when printing out non-4byte-word data sizes.

Printing 16-bit numbers from within C-language programs is done by attaching an 'h' to the print-format string ('h' stands for half-word). For example, the following code prints out short integers correctly.

```

int num = 0x983475;    /* larger than a 16-bit quantity */
short hword;

hword = num & 0xffff;

printf("short int: 0x%04hx, %hd \n", hword, hword);

```

The corresponding output:

```

short int: 0x3475, 13429

```

The first number printed is the value of `hword` as a hexadecimal number (the '04' tells the `printf` function to pad the number on the left with zeroes if necessary, up to a total length of 4 digits). The second number printed is the value of `hword` as a decimal number. If you leave off the 'h,' or instead use 'l,' the value printed might not reflect the actual value of the number (heavily dependent on the compiler).

Because the immediate value is a 7-bit 2's complement number, it only holds values ranging from -64 to 63. For symbolic addresses, your assembler will compute the immediate value so that the instruction refers to the correct label. Remember to check the resulting immediate value. Since Sun workstations are 64-bit machines, you'll have to chop off all but the lowest 7 bits for negative numbers.

Verilog Implementation

The second half of the project is to create in Verilog a CPU model of the RiSC-16 instruction set. As mentioned, the model is to be single-cycle sequential (non-pipelined) execution. This means that during every cycle, the CPU will execute a single instruction and will not move to the next instruction until the present instruction has been completed and the program counter redirected to a new instruction (the next instruction). This is the simplest form of processor model, so it should require very little code to implement. My solution, which is not particularly efficient, adds a mere 50 lines to

the skeleton code shown below. You should be able to develop your processor model well within a week.

Clearly, the main point of this exercise is not to investigate advanced architecture concepts but to teach you the rudiments of the Verilog modeling language. Future projects will explore more advanced architecture concepts like pipelines and precise interrupts.

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 BNE         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, to be sign-extended)
`define INSTRUCTION_LI     9:0  // large immediate (10-bit, to be 0-extended)
`define INSTRUCTION_SB     6    // immediate's sign bit

`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

    always @(posedge clk) begin
        // put your code here
    end
endmodule
```

The file 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 ];
```

would yield the opcode of the instruction contained in instr.

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 1000 cycles into the future.

Running Your CPU

First, “tap cds-v” 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. This is useful for long-running benchmarks.

Submitting Your Project

Submit your files as attachments to an email message. When you submit your code to me, all I want is the *assembler.c* file and *RiSC.v* file (or whatever names you have given them). I do not need anything else. For example, I do not need your *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.