Project 1: Single-Cycle RiSC-16 (10%)

ENEE ENEE 350: Computer Organization, Fall 2009
Assigned: Tuesday, Sep 1; Due: Tuesday, Sep 15

Purpose

This project is intended to help you understand the instructions of a very simple assembly language and how to assemble programs into machine-level code.

This project has three parts. In the first part, you will write a program to take an assembly-language program and produce the corresponding machine code. In the second part, you will write a behavioral simulator for arbitrary machine code. In the third part, you will demonstrate your assembler and simulator by writing a short assembly-language program to multiply two numbers.

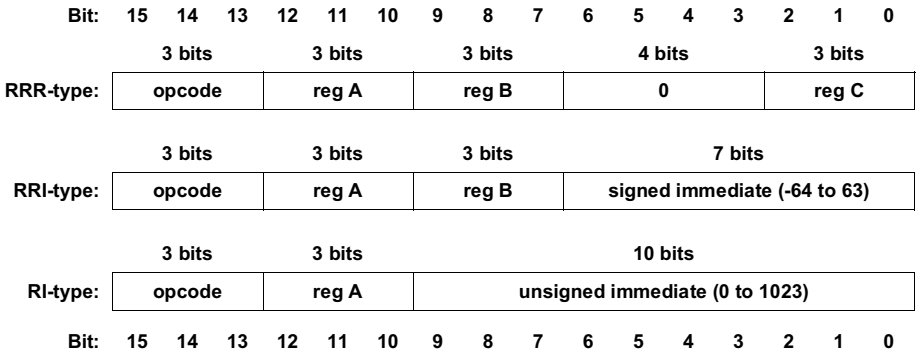
RiSC-16 Instruction-Set Architecture

For the first several projects, you will be gradually "building" the RiSC-16 (Ridiculously Simple Computer). The RiSC-16 is very simple, but it is general enough to solve complex problems. For this project, you will only need to know the architecture's instruction set and instruction format.

The RiSC-16 is an 8-register, 16-bit computer. All addresses are shortword addresses (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 the value 0, irrespective of what has been written there previously. There are three machine-code instruction formats and a total of 8 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

These eight instructions are classified into 3 machine-code instruction formats: RRR-type, RRI-type, and RI-type.



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+1+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+1 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 (3%)

The first part 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
`add regA, regB, regC`	R[regA] <- R[regB] + R[regC]
`addi regA, regB, immed`	R[regA] <- R[regB] + immed
`nand regA, regB, regC`	R[regA] <- ~(R[regB] & R[regC])
`lui regA, immed`	R[regA] <- immed & 0xffc0
`sw regA, regB, immed`	R[regA] -> Mem[R[regB] + immed]
`lw regA, regB, immed`	R[regA] <- Mem[R[regB] + immed]

`bne regA, regB, immed`	if (R[regA] != R[regB]) {
	PC <- PC + 1 + immed (if immed is numeric)
	(if immed is label, PC <- label)
	}

`jalr regA, regB`	PC <- R[regB], R[regA] <- 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
`nop`	do nothing
`halt`	stop machine & print state
`.fill immed`	initialized data with value *immed*

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 <- 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:* label = PC + 1 + 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*.

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" */
        }
}
```

Behavioral Simulator (4%)

The second part of this assignment is to write a program that can simulate any legal RiSC machine-code program. The input for this part will be the machine-code file that you created with your assembler. With a program name of "simulate" and a machine-code file of "code", your program should be run as follows:

```
simulate code > output
```

This directs all printfs to the file "output".

The simulator should begin by initializing all memory, registers, and the program counter to 0. The simulator will then simulate the program until the program executes a **halt**.

After every instruction executed, the simulation should print the current state of the machine (program counter, registers, memory). Print the memory contents for memory locations defined in the machine-code file (addresses 0-9 in the Section 4 example). Print the final state of the machine at the end of the simulation, too. You must use the format given in the **printState** function that is part of the project distribution ... your program will be graded based on its output, and if the output does not conform to the expected format, your program output will be considered incorrect.

Simulator Hints

Since this is a 16-bit simulator, it is much easier if every number in your simulator is of type *short*. You do not *have* to do this; it will simply save some debugging headache. Remember that whenever a variable is of type *short*, you must read into it (using scanf) or print it out (via printf) using a C printf format of '%hx' or '%hd' otherwise the printf/scanf function might read/write several extra bytes from/to surrounding data items. That would be bad.

Be careful how you handle the immediate values for **addi**, **lw**, **sw**, and **beq**. Remember that they are 2's complement 7-bit numbers, so when you need to use immediate values in computations, you will need to convert a 7-bit negative value to a 16-bit or 32-bit negative number on the Sun workstation by sign extending it. To do this, use the following function (assuming you decide to use 16-bit short integers throughout; if you use 32-bit integers, replace each *short* with *int* in the following code).

```
short convertNum(short num)
{
        /* convert a 7-bit number into a 16-bit Sun number */
        if (num & 0x40) {
                num -= 128;
        }
        return(num);
}
```

An example run of the simulator is included in the project distribution.

Code Fragment for Simulator

Here is some C code that may help you write the simulator. Again, take this merely as a hint. You may have to re-code this to make it do exactly what you want, but this should help you get started.

```
/* instruction-level simulator for RiSC */

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

#define MAXLINELENGTH 1000

#define MEMSIZE 0x7fff        /* maximum number of memory words */
short Memory[ MEMSIZE ];

main(int argc, char *argv[])
{
        FILE *filePtr;
        short address;
        char line[MAXLINELENGTH];

        if (argc != 2) {
                printf("error: usage: %s <machine-code file>\n", argv[0]);
                exit(1);
        }

        filePtr = fopen(argv[1], "r");

        if (filePtr == NULL) {
                printf("error: can't open file %s", argv[1]);
                perror("fopen");
                exit(1);
        }

        /* read in the entire machine-code file into memory */
        for (   address = 0;
                fgets(line, MAXLINELENGTH, filePtr) != NULL;
                address++) {

                if (sscanf(line, "%hx", &Memory[address]) != 1) {
                        printf("error in reading address %d\n", address);
                        exit(1);
                }

                printf("DEBUG: memory[%hd]=%04hx\n", address, Memory[address]);
        }
}
```

Assembly-Language Multiplication (3%)

Your simulator should work on any legal machine-code file, and your assembler should work on any legal assembly-language program. To show us how well your programs work, write an assembly-language program to multiply two numbers. Input the numbers by reading memory locations called "mcand" and "mplier". The result should be stored in register 1 when the program halts. You may assume that the two input numbers are at most 7 bits wide and that both are positive; this ensures that their (positive) result fits into a 16-bit RiSC-16 word. The algorithm in section 4.6 of the textbook shows how to multiply. Remember that shifting left by one bit is the same as adding the number to itself. Given the RiSC-16 instruction set, it is easiest to modify the algorithm so that you avoid the right shift. Submit a version of the program that computes (123 * 119).

You must use some kind of loop and shift algorithm to perform the multiplication—do not submit programs that compute the product via successive additions of one of the products (e.g. multiplying 5 * 6 by iteratively adding 5 six times) or perform each of the separate shift/add iterations with 8 different sections of code.

Make sure your simulator and assembler work for all instructions, even if your demonstration program doesn't contain any of a certain type of instruction.

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 >> b) is the number *a* shifted right by *b* bits. Neither *a* nor *b* are changed. E.g. (25 >> 2) is 6. Note that 25 is 11001 in binary, and 6 is 110 in binary.
3. The value of the expression (a << b) is the number *a* shifted left by *b* bits. Neither *a* nor *b* are changed. E.g. (25 << 2) is 100. Note that 25 is 11001 in binary, and 100 is 1100100 in binary.
4. *~a* is the bit-wise complement of *a* (*a* is not changed); if *a* = 100101, *~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.

Submitting Your Project

You should submit the following separate files:

- C program listing for your assembler
- C program listing for your simulator
- Assembly file for the multiplication program

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

Grading

We will grade primarily on functionality, including error handling, correctly assembling and simulating all instructions, input and output format, method of executing your program, and correctly multiplying. Be very careful to follow exactly the format for inputting the assembly language program, outputting the machine-code file (from the assembler), inputting the machine-code file (into the simulator), and outputting the state of the RiSC-16 machine while running the program. We will be running an automated checker on your files, so they must be in exactly the right format.

Error Checking

We will not intentionally feed incorrect input into your assembler or simulator, so you need not do any error checking. However, to keep your sanity while debugging, you will want to do error checking on your own, to help you identify the difference between bugs in your C code and bugs in the input (your test files may not be perfect). For example, you may want to have your assembler 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). You probably 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. Trust us, these problems happen, and it will save you hours of debugging time if you know the problem is in the input file and not your C code.