1. RiSC-16 Instruction Set

This paper describes the instruction set 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.

---

**FORMATS:**

- **RRR-type:**
  - opcodes: 3 bits
  - reg A: 3 bits
  - reg B: 3 bits
  - reg C: 3 bits

- **RRI-type:**
  - opcodes: 3 bits
  - reg A: 3 bits
  - reg B: 3 bits
  - signed immediate: 7 bits

- **RI-type:**
  - opcodes: 3 bits
  - reg A: 3 bits
  - immediate: 10 bits

**INSTRUCTIONS:**

- **ADD:**
  - opcodes: 3 bits
  - reg A: 3 bits
  - reg B: 3 bits
  - reg C: 3 bits

- **ADDI:**
  - opcodes: 3 bits
  - reg A: 3 bits
  - reg B: 3 bits
  - signed immediate: 7 bits

- **NAND:**
  - opcodes: 3 bits
  - reg A: 3 bits
  - reg B: 3 bits
  - signed immediate: 7 bits

- **SW:**
  - opcodes: 3 bits
  - reg A: 3 bits
  - reg B: 3 bits
  - signed immediate: 7 bits

- **LUI:**
  - opcodes: 3 bits
  - reg A: 3 bits
  - immediate: 10 bits

- **LW:**
  - opcodes: 3 bits
  - reg A: 3 bits
  - reg B: 3 bits
  - signed immediate: 7 bits

- **BNE:**
  - opcodes: 3 bits
  - reg A: 3 bits
  - reg B: 3 bits
  - signed immediate: 7 bits

- **JALR:**
  - opcodes: 3 bits
  - reg A: 3 bits
  - reg B: 3 bits
  - signed immediate: 7 bits
The following table describes the different instruction operations.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name and Format</th>
<th>Opcode (binary)</th>
<th>Assembly Format</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Add RRR-type</td>
<td>000</td>
<td>add rA, rB, rC</td>
<td>Add contents of regB with regC, store result in regA.</td>
</tr>
<tr>
<td>addi</td>
<td>Add Immediate RRI-type</td>
<td>001</td>
<td>addi rA, rB, imm</td>
<td>Add contents of regB with imm, store result in regA.</td>
</tr>
<tr>
<td>nand</td>
<td>Nand RRR-type</td>
<td>010</td>
<td>nand rA, rB, rC</td>
<td>Nand contents of regB with regC, store results in regA.</td>
</tr>
<tr>
<td>lui</td>
<td>Load Upper Immediate RI-type</td>
<td>011</td>
<td>lui rA, imm</td>
<td>Place the 10 ten bits of the 16-bit imm into the 10 ten bits of regA, setting the bottom 6 bits of regA to zero.</td>
</tr>
<tr>
<td>sw</td>
<td>Store Word RRI-type</td>
<td>101</td>
<td>sw rA, rB, imm</td>
<td>Store value from regA into memory. Memory address is formed by adding imm with contents of regB.</td>
</tr>
<tr>
<td>lw</td>
<td>Load Word RRI-type</td>
<td>100</td>
<td>lw rA, rB, imm</td>
<td>Load value from memory into regA. Memory address is formed by adding imm with contents of regB.</td>
</tr>
<tr>
<td>bne</td>
<td>Branch If Not Equal RRI-type</td>
<td>110</td>
<td>bne rA, rB, imm</td>
<td>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.</td>
</tr>
<tr>
<td>jalr</td>
<td>Jump And Link Register RRI-type</td>
<td>111</td>
<td>jalr rA, rB</td>
<td>Branch to the address in regB. Store PC+1 into regA, where PC is the address of the jalr instruction.</td>
</tr>
</tbody>
</table>

2. RiSC-16 Assembly Language and Assembler

The distribution includes a simple assembler for the RiSC-16 (this is the first project assigned to my students in the computer organization class). The 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).

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. 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 (you need to have either whitespace or a comma or both in between each 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. It is not inter-
interpreted as the decimal number 32. Hexadecimal numbers are preceded by the string ‘0x’ (oh-x). For example, 0x12 is ‘hex-one-two’ and corresponds to the decimal number 18, not decimal 12. For those of you who know the C programming language, you should be perfectly at home.

The number of fields depends on the instruction. The following table describes the instructions.

<table>
<thead>
<tr>
<th>Assembly-Code Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi regA, regB, immed</td>
<td>$R[regA] \leftarrow R[regB] + \text{immed}$</td>
</tr>
<tr>
<td>nand regA, regB, regC</td>
<td>$R[regA] \leftarrow \neg(R[regB] &amp; R[regC])$</td>
</tr>
<tr>
<td>lui regA, immed</td>
<td>$R[regA] \leftarrow \text{immed} &amp; 0xffc0$</td>
</tr>
<tr>
<td>sw regA, regB, immed</td>
<td>$R[regA] \rightarrow \text{Mem}[R[regB] + \text{immed}]$</td>
</tr>
<tr>
<td>lw regA, regB, immed</td>
<td>$R[regA] \leftarrow \text{Mem}[R[regB] + \text{immed}]$</td>
</tr>
</tbody>
</table>
| bne regA, regB, immed | if ( $R[regA] \neq R[regB]$ ) { \(\begin{align*} 
PC & \leftarrow PC + 1 + \text{immed} \\
(\text{if label, } PC & \leftarrow \text{label}) \end{align*}\) } |
| jalr regA, regB       | $\text{PC} \leftarrow R[regB], R[regA] \leftarrow \text{PC} + 1$ |

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).

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

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 jalr 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 JALR 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.

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 2,1,1    # load reg2 with -1 (uses numeric address)
start:      # addi 1,1,-1
    add 1,1,2 # decrement reg1 -- could have been addi 1,1,-1
    bne 0,1,start # loop again if reg1!= 0
done:       # end of program
    halt
count:      # will contain the address of start (2)
cneg1:      .fill -1
startAddr:  .fill start
```

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).

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). And, as mentioned, any 16-bit number can be moved into a register in two operations (lui+lli).

- **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).