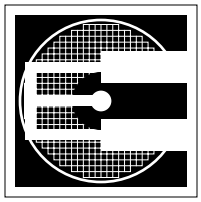

The RiSC-16 Instruction-Set Architecture

ENEE 646: Digital Computer Design, Fall 2002
Prof. Bruce Jacob

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:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RRR-type:	3 bits: opcode			3 bits: reg A			3 bits: reg B			4 bits: 0				3 bits: reg C		
RRI-type:	3 bits: opcode			3 bits: reg A			3 bits: reg B			7 bits: signed immediate (-64 to 63)						
RI-type:	3 bits: opcode			3 bits: reg A			10 bits: immediate (0 to 0x3FF)									

INSTRUCTIONS:

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

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 10 ten bits of the 16-bit **imm** into the 10 ten 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**.
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.

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-

preted 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.

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 label, PC <- label)
		}

`jalr`	`regA, regB`	PC <- R[regB], R[regA] <- 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
`nop`		do nothing
`halt`		stop machine & print state
`lli`	`regA, immed`	R[regA] <- R[regA] + (immed & 0x3f)
`movi`	`regA, immed`	R[regA] <- immed
`.fill`	`immed`	initialized data with value *immed*
`.space`	`immed`	zero-filled data array of size *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 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-ofOrder 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:      add     1,1,2           # decrement reg1 -- could have been addi 1,1,-1
            bne     0,1,start       # loop again if reg1!= 0
done:       halt                    # end of program
count:      .fill   5
neg1:       .fill   -1
startAddr:  .fill   start           # will contain the address of start (2)
```

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).