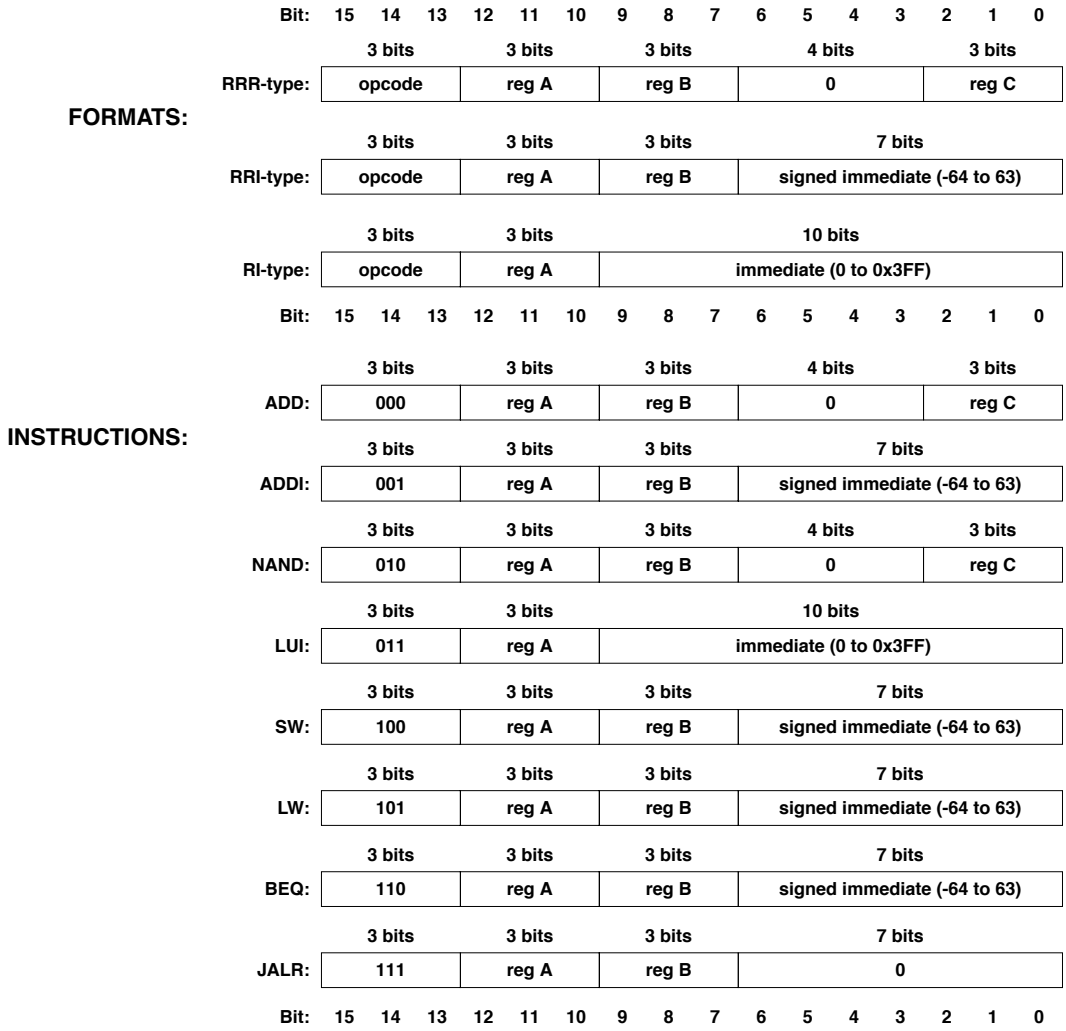
O Conjunto de Instruções da arquitetura RiSC-16

1. Conjunto de instruções RiSC 16

Este artigo descreve o conjunto de instruções do Ridiculously Simple Computer (Computador Ridiculamente Simples) de 16 bits, uma ISA didática que é baseada no Litter Computer (CL-896) desenvolvido por Peter Chen na Universidade de Michigan. O RiSC-16 é um computador de 8 registradores de 16 bits. Todos os endereços são do tipo shortword (isto é, o endereço 0 corresponde aos primeiros dois bytes da memoria principal, o endereço 1 corresponde aos segundos dois bytes da memória principal, etc.). Como o conjunto de instruções da arquitetura MIPS, por convenção de hardware, o registrador 0 sempre conterá o valor 0. A máquina impõe isso: leituras do registrador 0 sempre retornará 0, independente do que tiver sido escrito nele. O RiSC-16 é muito simples, mas é genérico o bastante para resolver problemas complexos. Há três formatos de código de máquina e um total de 8 instruções. Elas estão ilustradas na figura abaixo.



A tabela a seguir descreve as operações das instruções.

Mnemônico	Nome e Formato	Opcode (binário)	Formato Assembly	Ação
add	Add RRR-type	000	add rA, rB, rC	Adiciona o conteúdo de **regB** com **regC** e armazena o resultado em **regA**.
addi	Add Imediato RRI-type	001	addi rA, rB, imm	Adicionar o conteúdo de **regB** com **imm** e armazena o resultado em **regA**
nand	Nand RRR-type	010	nand rA, rB, rC	Faz nand do conteúdo de **regB** com **regC** e armazena o resultado em **regA**.
lui	Load Upper Immediate RI-type	011	lui rA, imm	Coloca os 10 primeiros bits do **imm** de 16 bits dentro dos 10 primeiros bits de **regA**, colocando os seus últimos bits de **regA** em zero.
sw	Store Word RRI-type	101	sw rA, rB, imm	Armazena o valor de **regA** dentro da memoria. O endereço na memória é formado pela adição de **imm** com o conteúdo de **regB**.
lw	Load Word RRI-type	100	lw rA, rB, imm	Carrega valor da memória para **regA**. O endereço na memória é formado pela adição de **imm** com o conteúdo de **regB**.
beq	Branch if Equal RRI-type	110	beq rA, rB, imm	Se o conteúdo de **regA** e **regB** são iguais, desvia para o endereço PC + 1 + **imm**, onde PC é o endereço da instrução beq.
jalr	Jump and Link Register RRI-type	111	jalr rA, rB	Desvia para o endereço em **regB**. Armazena PC + 1 em regA, onde PC é o endereço da instrução jarl.

2. A linguagem Assembly RiSC-16 e o Assembler

O formato para uma linha de código é:

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

O campo mais a direita na linha é o campo de rótulo. Rótulos RiSC válidos são qualquer combinação de letras e números seguidos por dois pontos (:). Os dois pontos no fim não são opcionais – um rótulo sem os dois pontos é interpretado como um opcode. Depois do rótulo opcional vem um espaço em branco, (espaço(s) ou tab(s)). Então vem o campo de opcode, onde o opcode por ser qualquer das instruções mnemônicas da linguagem assembly listados na tabela acima. Após mais um espaço em branco vem uma séria de campos separados por vírgulas e possivelmente espaços em branco (é preciso ter ou um espaço em branco ou vírgula ou os dois entre os campos). Todos os campos de valores para registadores são números **decimais**, opcionalmente precedidos pela letra ‘r’ ... como em r0, r1,r2, etc. Campos de valores imediatos são número no formato decimal, octal ou hexadecimal. Número octais são precedidos pelo caractere ‘0’ (zero). Por exemplo, 032 é interpretado como o número octal ‘zero-três-dois’ que corresponde ao número decimal 26. Ele *não* é interpretado como o número decimal 32. Números hexadecimais são precedidos pelos caracteres

‘0x’ (zero-x). Por exemplo, 0x12 é ‘hex-um-dois’ e corresponde ao número decimal 18, não ao decimal 12. Vocês que conhecem a linguagem de programação C, se sentirão perfeitamente em casa.

O número de campos depende da instrução. A seguinte tabela descreve as instruções.

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]

`beq`	`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

Qualquer coisa depois de ‘#’ é considerado um *comentário* e é ignorado. O campo de comentário termina no fim da linha. Comentários são vitais para criar programas em linguagem assembly compreensíveis, porque as instruções em si são um tanto obscuras.

Em adição as instruções RiSC-16, um programa assembly pode conter diretivas para o assembler. Essas diretivas são geralmente chamadas *pseudo-instruções*. As seis diretivas do assemble que nós vamos usar são **nop**, **halt**, **lli**, **movi**, **.fill** e **.space** (note o . no inicio de **.fill** e **.space**, que significa simplesmente que estes valores representam dados, não instruções executáveis).

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*

Os seguintes parágrafos descrevem essas pseudo-instruções em maior detalhe:

- A pseudo-instrução **nop** significa “nada a fazer neste ciclo” e é substituída por **add 0,0,0** (a qual claramente não faz nada).
- A pseudo-instrução **halt** significa “pare de executar instruções e imprima o atual estado da máquina” e é substituída por **jarl 0,0** com um campo imediato diferente de zero. Isto é descrito com maiores detalhes nos documentos *The*

pipelined RiSC16 e *Out-Of-Order RiSC-16*, em que HALT é um subconjunto das instruções de chamada de sistema para o proposito de manipular interrupções e exceções: qualquer instrução JALR com um valor imediato diferente de zero usa este imediato como um opcode da chamada de sistema. Isso permite que tais instruções se comportem como chamadas de sistema, parada, returno de exceção, etc.

- A pseudo-instrução **lli** (*load-lower-immediate*) significa "OU os últimos seis bits deste número para dentro do registrador indicado" e é substituído por **addi X,X,imm6**, onde **X** é o registrador especificado e **imm6** é igual a **imm & 0x3f**. Esta instrução pode ser usada in conjunto com **lui**: a instrução **lui** primeiro move os dez primeiros bits de um dado número (ou endereço, se um rótulo é especificado) para o registrador, colocando os seis últimos bits como 0; a instrução **lli** move os seis últimos bits. O número de seis bits será interpretado como positivo, evitando assim extensão de sinal; assim, o **addi** resultante é essencialmente a concatenação dos dois campos de bits.
- A pseudo-instrução **movi** é somente um atalho para a combinação **lui+lli**. Observe, contudo, que a instrução **movi** *parece* representar uma única instrução, enquanto que de fato ela representa duas. Isso pode atrapalhar sua contagem se você está esperando uma certa distância entre as instruções. Assim, é sempre uma boa ideia usar rótulos sempre que possível.
- A diretiva **.fill** diz para o assemble colocar um número naquele lugar onde a instrução seria normalmente armazenada. A diretiva **.fill** usa um campo, o qual pode ser ou um valor numérico ou um endereço simbólico. Por exemplo, ".fill 32" coloca o valor 32 onde a instrução normalmente seria armazenada. Usando **.fill** com um endereço simbólico armazenará o endereço do rótulo. No exemplo acima, a linha ".fill start" armazenará o valor 2, porque o rótulo "start" refere ao endereço 2.
- A diretiva .**space** tem um inteiro **n** como argumento e é substituído por **n** cópias de ".fill 0" no código; isto é, ela resulta na criação de **n** palavras de 16 bits todas inicializadas com 0 (zero).

A seguir está um programa em linguagem assembly que decrementa a partir de 5, parando quando chega a 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
             beq     0,1,1         # goto end of program when reg1==0
             beq     0,0,start     # go back to the beginning of the loop
done:        halt                  # end of program
count:       .fill   5
neg1:        .fill   -1
startAddr:   .fill   start         # will contain the address of start (2)
```

Em geral, código assembly RiSC aceitável tem uma instrução por linha. Tudo bem se há uma linha em branco, se é comentada (isto é, a linha começa com #) ou não (isto é, somente uma linha em branco). Contudo, um rótulo **não pode** aparecer em uma linha sozinho; ele tem que ser seguido por uma instrução válida na mesma linha (a diretiva **.fill** ou **halt**, **nop**, **etc** contam como uma instrução).

Note que as 8 instruções básicas da arquitetura RiSC-16 formam uma ISA completa que pode executar computação arbitrária. Por exemplo:

- **Mover valores constante para os registradores**. O número 0 pode ser movido para qualquer registrador em um ciclo (`add rX r0 r0`). Qualquer número entre -64 e 63 pode ser colocado em um registrador em uma operação usando a instrução ADDI (`addi rX r0 number`). E, como mencionado, qualquer número de 16 bits pode ser movido para um registrador em duas operações (**lui+lli**).
- **Subtraindo números.** Subtração é simplesmente a adição do valor negativo. Qualquer número pode ser transformado em um número negativo em duas instruções, pela inversão de seus bits e adicionado 1. Inversão de bits pode ser alcançada fazendo um NAND com o próprio valor; adicionar 1 é feita com a instrução ADDI. Por tanto, subtração é um processo de três instruções. Note que sem um registrador extra, este é um processo destrutivo.
- **Multiplicando números.** Multiplicação é facilmente feita por repetidas adições, teste de bit e deslocamento de um bit a esquerda de uma máscara de bits (o qual é o mesmo que uma adição consigo mesmo).[i]

---

[i] Todas as imagens foram retiradas do artigo *The RiSC-16 Instruction-Set Architecture* escrito pelo Prof. Bruce Jacob. O texto integral e em inglês pode ser encontrado em https://www.ece.umd.edu/~blj/RiSC/RiSC-isa.pdf