Capítulo 3

Linguagem de Máquina

3.1 – Introdução

Neste capítulo vamos conhecer melhor a tarefa do montador. Vamos, de fato, conhecer o objeto que ele gera, a linguagem de máquina do MIPS. Já sabemos que existe uma correspondência de um para um entre uma instrução em linguagem de montagem e uma instrução em linguagem de máquina. Sabemos também que no MIPS todas as instruções têm o mesmo tamanho: 32 bits.

Bem, para começarmos a conhecer a linguagem de máquina, vamos abordar a conversão da instrução `add $8, $9, $10` em sua equivalente. Os bits que formam a instrução são divididos em grupos que indicam a operação a ser realizada e os operandos. No MIPS os 6 bits mais significativos (mais a esquerda) formam um campo chamado ***opcode***. Ele pode conter um código de uma instrução em particular, ou um código que indica uma família de instruções. No nosso exemplo da instrução `add`, o *opcode* vale 000000_2 indicando uma família de instruções. Os 6 bits menos significativos são responsáveis então por definir que instrução está codificada. Este campo nós chamamos de função (***function***). Os operandos são indicados em uma ordem particular. O campo `rd` significa registrador de destino, o campo `rs`, registrador de fonte (*source*) e o campo rt, registrador de alvo (*target*), que pode ser destino ou fonte.

A Figura 3.1 mostra os campos da instrução `add $8, $9, $10`. Como o *opcode* vale 000000_2, a instrução é definida pelo campo *function*. Este campo vale 32 (100000_2) para instrução `add`. O campo `rs` indica o registrador que compõe a primeira parcela da soma. O campo `rt`, a segunda. O campo `rd` indica o registrador onde o resultado da soma será colocado.

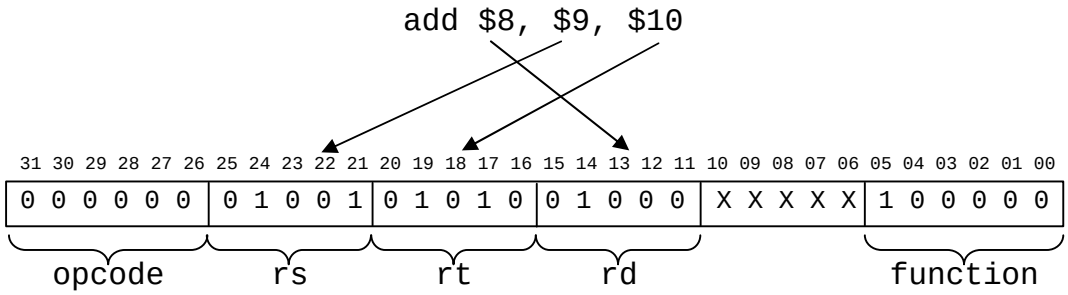


Figura 3.1: Instrução add $8, $9, $10 em linguagem de máquina

Os campos `rs`, `rd` e `rt` guardam os números dos registradores operandos da instrução. Existe um campo não utilizado entre os bits 6 e 10, inclusive. Veremos que este campo será utilizado para outra funcionalidade, mas por enquanto vamos considerá-lo como sendo sempre 00000_2.

Agora vamos a um exemplo inverso: dado o código de máquina, desejamos encontrar o código *assembly* correspondente. A instrução é dada como sendo $016c820_h$. A primeira coisa a fazer é encontrar a representação binária correspondente. Em seguida vamos procurar os 6 bits mais significativos e descobrimos que eles valem 000000_2. Ora, já sabemos que este valor no campo `opcode` indica uma família de instruções e a instrução dentro desta família é indicada pelos 6 bits menos significativos. Então olhamos para os 6 bits menos significativos, eles indicam 100000_2. Este valor no campo function indica uma instrução de soma. Bem, então agora é só mostrar quais são os campos da instrução e distribuir os bits neles. Daí, encontraremos `rs` valendo 01011_2, `rt` valendo 01100_2 e `rd` valendo 01101_2. Agora reconstruímos a instrução `add rs, rt, rd` = `add $11, $12, $13`. A Figura 3.2 mostra o processo de descoberta da instrução correspondente.

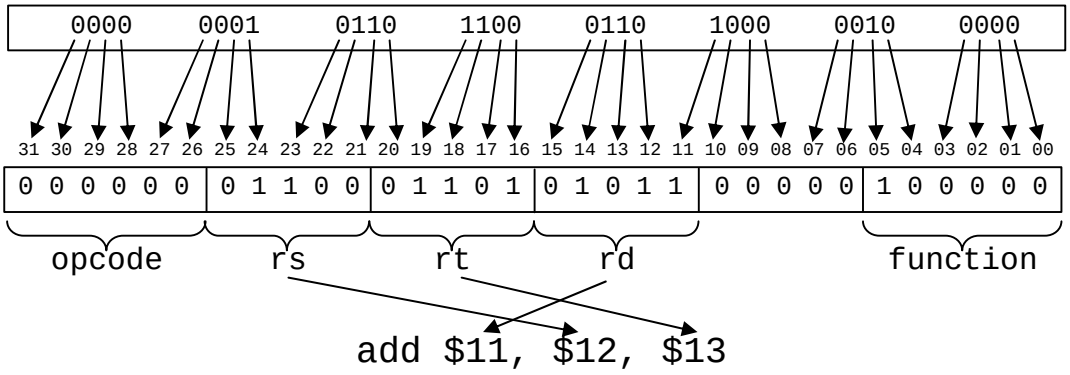


Figura 3.2: Instrução add $11, $12, $13 em linguagem de máquina

Agora vamos à outra instrução: `sub $8, $10, $11`. Esta instrução também tem um *opcode* igual a 000000_2. Isto indica que ela pertence à mesma família da instrução `add`. O valor do campo `function` é: 100010_2. A codificação segue o mesmo padrão: 6 bits para `opcode`, 5 para `rs`, 5 para `rt`, 5 para `rd`, 5 desconsiderados e 6 para `function`, nesta seqüência necessariamente. A Figura 3.3 mostra a codificação da instrução.

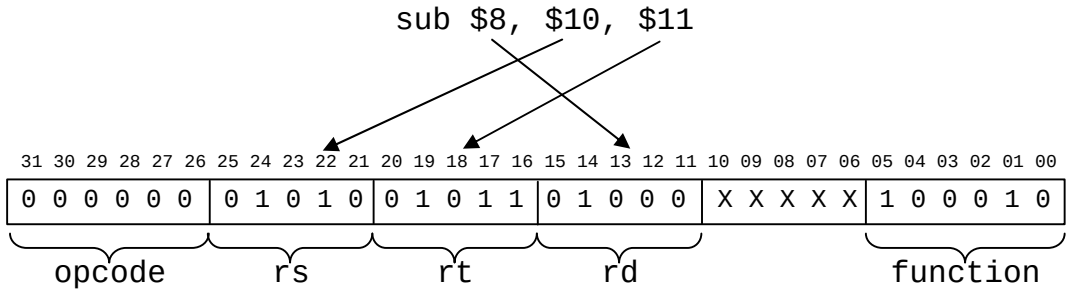


Figura 3.3: Instrução `sub $8, $10, $11` em linguagem de máquina

Para finalizarmos esta família vamos ver a codificação da instrução `sll $8, $9, 3`. Esta também é uma instrução que segue o mesmo padrão de codificação. Observe, entretanto que existe uma quantidade especificada na instrução, que é o montante de bits do dado no registrador `rt` que iremos deslocar para esquerda. `rd` deverá conter o resultado da operação e `rs` é desprezado (colocado em 00000_2). Bem, então fica faltando como especificar a quantidade 3 que representa o montante do deslocamento. Este montante é representado em um campo chamado montante de deslocamento, `sa` (*shift amount*). São usados os bits de 6 a 10 para ser o `sa`. A Figura 3.4 mostra a codificação da instrução `sll $8, $9, 3`, onde `function` vale 000000_2.

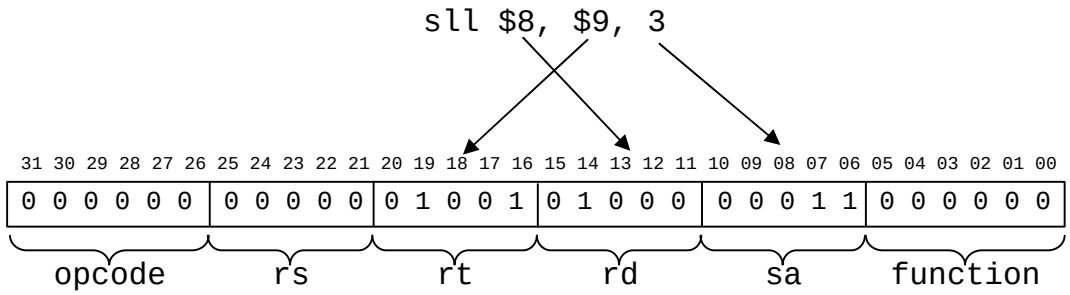


Figura 3.3: Instrução `sll $8, $9, 3` em linguagem de máquina

Este formato de codificação de instruções é compartilhado por muitas instruções e ele recebe um nome: formato R. A Tabela 3.1 mostra a codificação das principais instruções do tipo R. Observe que a instrução `mul`, embora compartilhe este formato, não tem como `opcode` o valor 000000_2.

Nome	Formato	Exemplo	Codificação					
			opcode	rs	rt	rd	sa	function
sll	R	sll $8, $9, 3	0	9	10	8	3	0
srl	R	srl $8, $9, 3	0	0	10	8	3	2
jr	R	jr $8	0	8	0	0	0	8
mfhi	R	mfhi $8	0	0	0	8	0	16
mflo	R	mflo $8	0	0	0	8	0	18
mult	R	mult $9, $10	0	9	10	0	0	24
multu	R	multu $9, $10	0	9	10	0	0	25
div	R	div $9, $10	0	9	10	0	0	26
divu	R	divu $9, $10	0	9	10	0	0	27
add	R	add $8, $9, $10	0	9	10	8	0	32
addu	R	addu $8, $9, $10	0	9	10	8	0	33
sub	R	sub $8, $9, $10	0	9	10	8	0	34
subu	R	subu $8, $9, $10	0	9	10	8	0	35
and	R	and $8, $9, $10	0	9	10	8	0	36
or	R	or $8, $9, $10	0	9	10	8	0	37
slt	R	slt $8, $9, $10	0	9	10	8	0	42
sltu	R	sltu $8, $9, $10	0	9	10	8	0	43
mul	R	mul $8, $9, $10	28	9	10	8	0	2

Tabela 3.1: Instruções da família R

Bem, agora já conhecemos a codificação de diversas instruções no MIPS. Vamos conhecer um outro formato possível. São as instruções do tipo I, onde um dado imediato vem na codificação da própria instrução.

Comecemos com a instrução `addi $8, $9, 3`. Esta instrução especifica uma parcela da soma em um operando e a outra na própria instrução. A Figura 3.4 mostra a codificação utilizada. Um campo

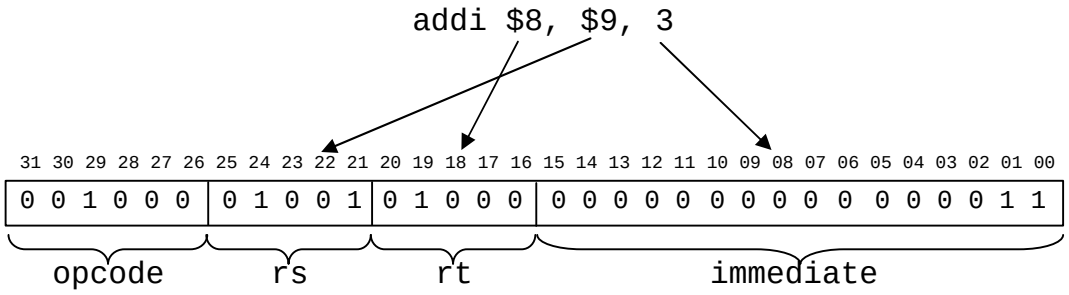


Figura 3.4: Instrução `addi $8, $9, 3` em linguagem de máquina

`immediate` é usado para indicar a quantidade que será somada ao registrador `rs`. O destino é o registrador `rt`. O campo `opcode` indica agora a instrução especificamente, e não uma família.

Observe que o campo `immediate` contém 16 bits, o que implica na limitação de um dos operandos à faixa entre –32768 e +32767. Isto é algo importante na nossa programação, pois uma instrução `addi $8, $9, 128256`, embora pareça válida, não é possível ser codificada.

Outra instrução do tipo I é a `lw`. `lw` usa o campo `immediate`, como sendo um deslocamento a partir de um endereço base especificado em um registrador, como vimos no Capítulo anterior. A Figura 3.5 mostra a codificação de `lw`, que também é seguida por `sw`.

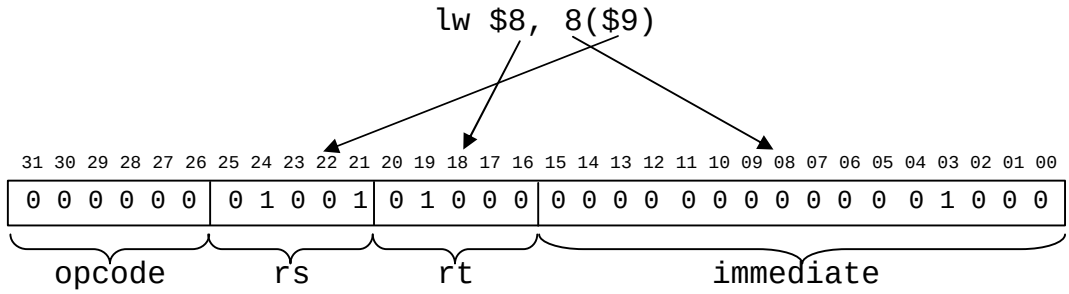


Figura 3.5: Instrução `lw $8, 8($9)` em linguagem de máquina

Finalmente, dentro deste formato existem as instruções `beq` e `bne`. Vamos precisar entender agora como é formado um endereço alvo de uma instrução condicional de salto. Assim como na instrução de carga de registradores existe um endereço base e um deslocamento, na instrução de desvio também existe um deslocamento. O registrador que guarda o endereço base, entretanto, está implícito, o que significa dizer que ele não será declarado na instrução. Bem, este tal registrador é justamente o PC. Então o deslocamento, estipulado no campo `immediate` da instrução de desvio condicional é somado ao valor de PC para informar ao processador qual o endereço da próxima instrução a ser executada.

A Figura 3.6 mostra a codificação da instrução `beq $8, $9, endereço_alvo`. Veja que neste caso um deslocamento, a partir do valor de PC, é aplicado, o que nos leva a refletir que a instrução `beq` é um **desvio condicional relativo a PC**. Este **modo de endereçamento**, é chamado relativo a PC, pois o próximo valor a ser endereçado depende do valor atual do PC.

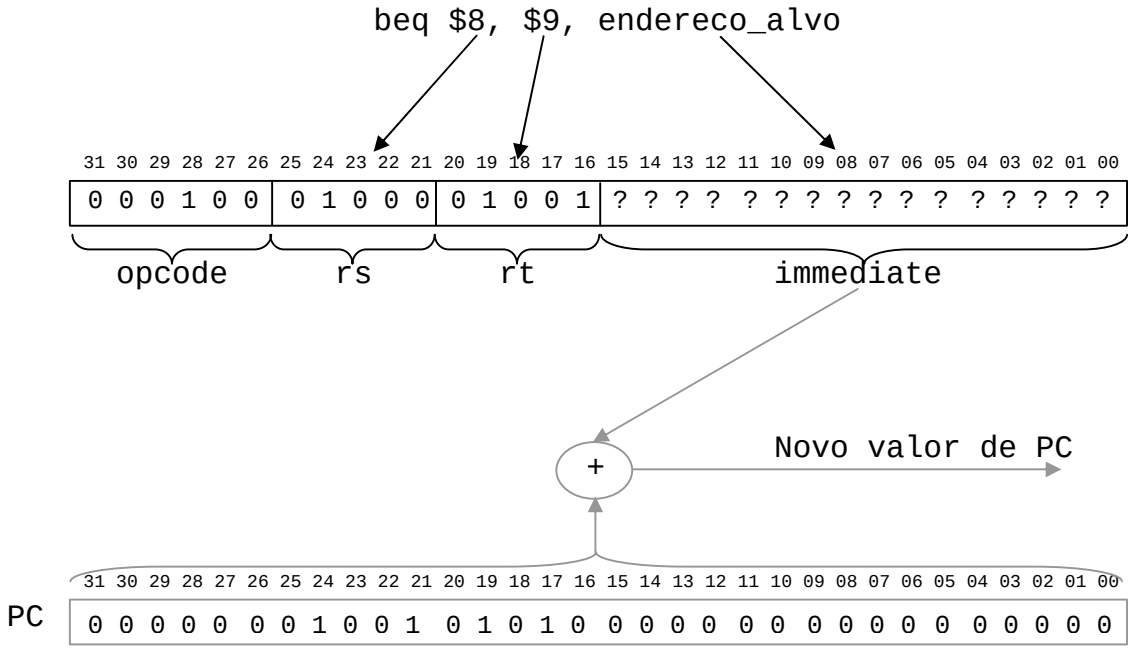


Figura 3.6: Instrução `beq $8, $9, alvo` em linguagem de máquina

Um aspecto muito importante nesta construção de desvios (instruções de saltos, como `beq` e `bne`) é que todas as instruções têm 32 bits, o que implica que a instrução antecessora da atual difere, em seu endereço de –4bytes. Seria um desperdício então utilizar este campo `immediate` para endereçar bytes, já que nunca os bits 0 e 1 seriam diferentes de 0. Por este motivo, o valor imediato especificado na instrução, antes de ser somado ao valor de PC é deslocado duas casas para esquerda, ou seja, o campo `immediate`, no final das contas, indica para quantas instruções antes ou depois da atual deve ser o salto, caso ocorra.

Vamos recorrer a uma adaptação do exemplo da Figura 2.11, mostrada novamente na Figura 3.7. A codificação do `opcode` e dos registradores formam os primeiros 16 bits da instrução: 1109_h. O valor do deslocamento é especificado como a distância entre a instrução atual e o seu alvo. Como o alvo da instrução de salto está duas instruções abaixo, precisamos especificar o campo `immediate` como sendo +2, isto é, 0002_h. Assim a codificação final da instrução é: 11090002_h, como visto na figura.

Quando estudarmos *pipeline* veremos que esta codificação é uma aproximação da realidade, mas vamos deixar esta discussão para o futuro. Por enquanto vamos assumir exatamente este modelo.

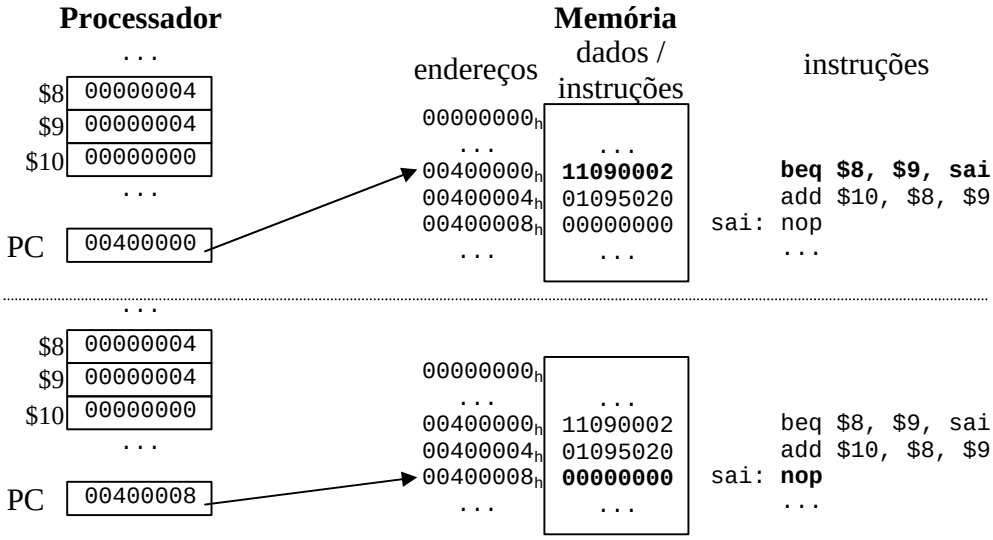


Figura 3.7: Execução e codificação, em linguagem de máquina, da instrução `beq $8, $9, sai`

Vamos fazer um exemplo inverso, dado a codificação em hexadecimal de uma instrução de salto vamos encontrar como seria sua equivalente em linguagem de montagem. A instrução é: $1509fffd_h$. Primeiro começamos encontrando a codificação equivalente em binário. O `opcode` é 000101_2. Se observarmos o exemplo original na Figura 2.11, vamos perceber que se trata do `opcode` da instrução `bne`. Ora, `bne` possui o mesmo formato de codificação de `beq`, então podemos separar os campos `rs` e `rt` para encontrar os registradores que serão comparados, neste caso $8 e $9 e vamos encontrar o alvo. Os 16 bits menos significativos indicam: $fffd_h$. Portanto, este é um número negativo, considerando que sua codificação está em 16bits. O seu equivalente em decimal vale –3. Agora chegamos à conclusão que se trata da instrução `bne $8, $9, -3`. Este –3 é a representação numérica de um rótulo do código. Um possível trecho de código que usa esta instrução é mostrado a seguir e a desmontagem está na Figura 3.8.

```
        ...
rotulo: lw ....
        lw ...
        addi ...
        bne $8, $9, rotulo #alvo à -3 instruções
```

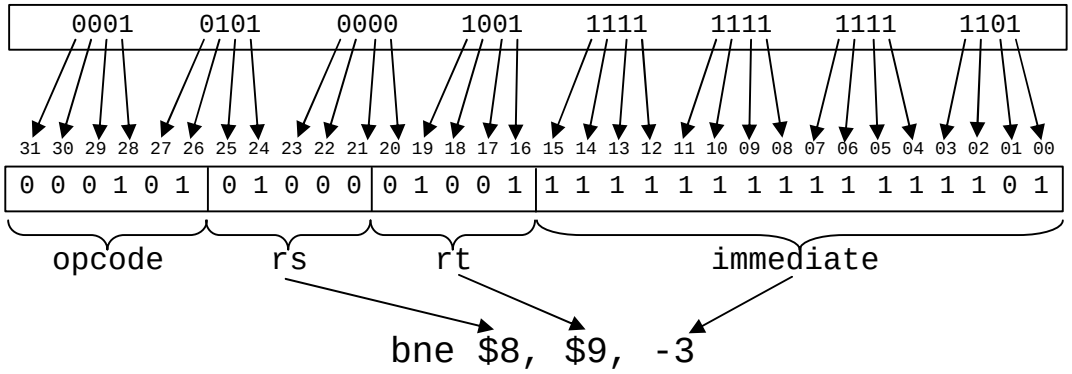


Figura 3.8: Desmontagem da instrução `bne $8, $9, -3` em linguagem de máquina

Uma observação final neste modelo é que podemos saltar para instruções que estão distantes até +32767 ou –32768 palavras da atual.

Vamos sumarizar as instruções do tipo I. A Tabela 3.2 mostra as principais instruções do MIPS que seguem este formato. Cuidado com a especificação dos registradores de `beq` e `bne`, pois eles são colocados em ordem direta na codificação e também, em `lui`, não existe o registrador `rs`.

Nome	Formato	Exemplo	Codificação			
			opcode	rs	rt	immediate
beq	I	beq $8, $9, 3	4	8	9	3
bne	I	bne $8, $9, 3	5	8	9	3
addi	I	addi $8, $9, 3	8	9	8	3
addiu	I	addiu $8, $9, 3	9	9	8	3
slti	I	slti $8, $9, 3	10	9	8	3
sltiu	I	sltiu $8, $9, 3	11	9	8	3
andi	I	andi $8, $9, 3	12	9	8	3
ori	I	ori $8, $9, 3	13	9	8	3
lui	I	lui $8, 3	15	0	8	3
lw	I	lw $8, 4($9)	35	9	8	4
sw	I	sw $8, 4($9)	43	9	8	4

Tabela 3.2: Instruções da família I

Finalmente chegamos ao último formato de instruções admitido pelo MIPS. Trata-se do formato J. Este formato é usado por instruções de desvio incondicional, como `j`, e `jal`. Ele é composto basicamente de um `opcode` e todos os demais bits são utilizados para endereçamento. A Figura 3.9 mostra o exemplo para instrução `j endereço`.

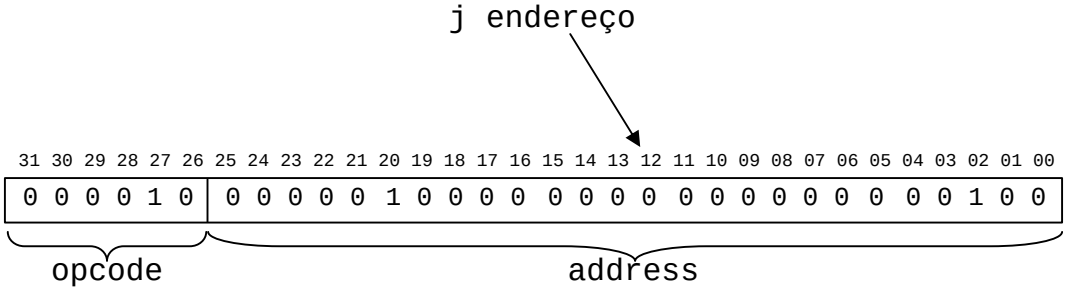


Figura 3.9: Instrução `j endereço` em linguagem de máquina

Assim como as instruções `bne` e `beq` usam um campo de 16 bits que é transformado em 18 bits para ser somado ao PC, os 26 bits do campo `address` deste formato também são deslocados à esquerda para serem transformados em 28 bits. Mas as similaridades param por aí. O cálculo do endereço de alvo do salto incondicional é feito de forma completamente diferente. A idéia é que haja uma independência do valor de PC. Este modo de endereçamento é chamado de **endereçamento direto**.

Infelizmente apenas 28 bits de endereço (26 explícitos e 2 implícitos) são carregados na instrução, o que não permite completar os 32 bits exigidos pela arquitetura. Assim, tomamos emprestado os 4 bits mais significativos do valor de PC para completar o endereço do alvo da instrução de salto. Formamos assim um modo de endereçamento chamado de **endereçamento pseudo-direto** pois ele tende a ser independente do PC, mas precisa utilizar alguns bits dele.

A Figura 3.10 mostra como é feito o cálculo do endereço alvo de uma instrução `j`. O campo *address* da instrução é escrito em PC, independentemente do que ali existia anteriormente. Entretanto, apenas os bits de número 2 a 27 são alterados. O *nibble* mais significativo do valor de PC fica mantido intacto. Veja que os nossos programas ocupam, no modelo de memória estabelecido, entre os endereços 00400000_h até $0fffffc_h$. Isto nos permite uma certa tranqüilidade, pois sabemos que o primeiro *nibble* do endereço contido em PC sempre será 0_h.

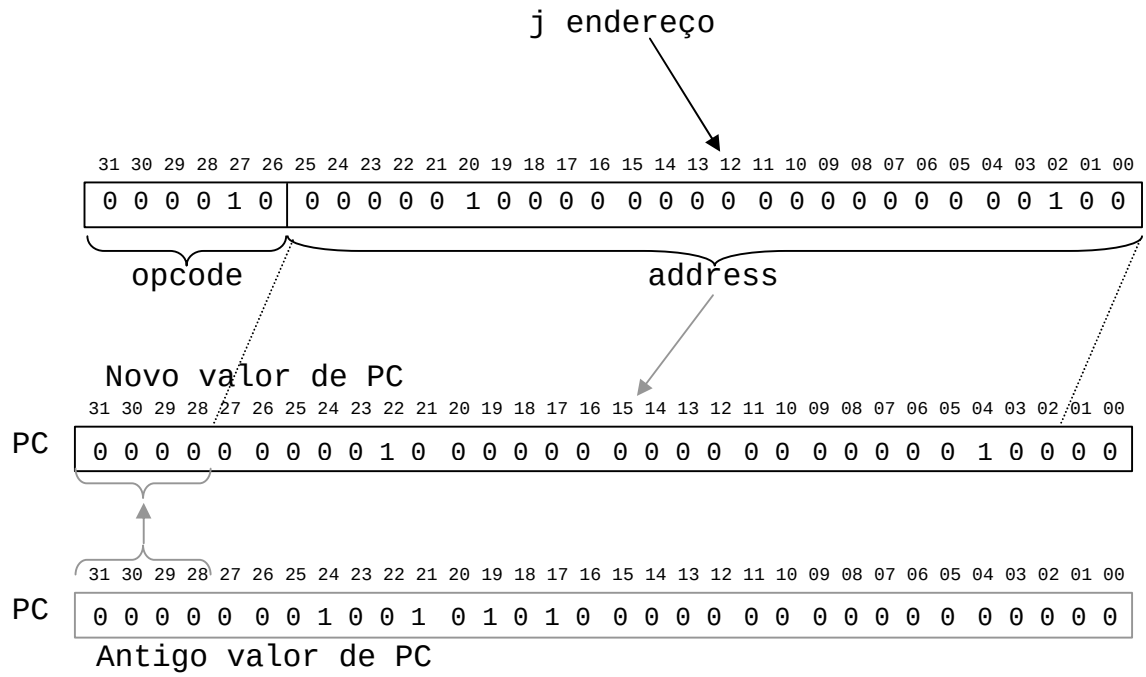


Figura 3.10: Instrução e cálculo de endereço para instrução `j endereço` em linguagem de máquina

Como deduzimos da Figura 3.9, o `opcode` associado à instrução `j` é 000010_2. Agora vamos ver um exemplo de como é feito o cálculo do campo `address` desta instrução. Vamos utilizar o exemplo da Figura 2.15 reproduzida a seguir, como Figura 3.11, para facilitar a leitura. Vamos prestar atenção na codificação da instrução `j sai`. Ela é exatamente a mesma mostrada na Figura 3.9. O Campo `address` contém o valor 0100004_h. Quando o processador vai executar esta instrução ele multiplica este valor por quatro (desloca dois bits à esquerda). Isto nos leva a 0400010_h. Este valor irá sobrepor os 28 bits mais baixos do PC. O primeiro *nibble* do PC fica como estava: 0_h. O endereço de destino passa então a ser 00400010_h. Justamente onde o rótulo `sai` está.

O campo de 26 bits não é operado (somado, subtraído, etc) a nenhum outro valor, portanto, não faz sentido em falarmos de valores sinalizados ou não.

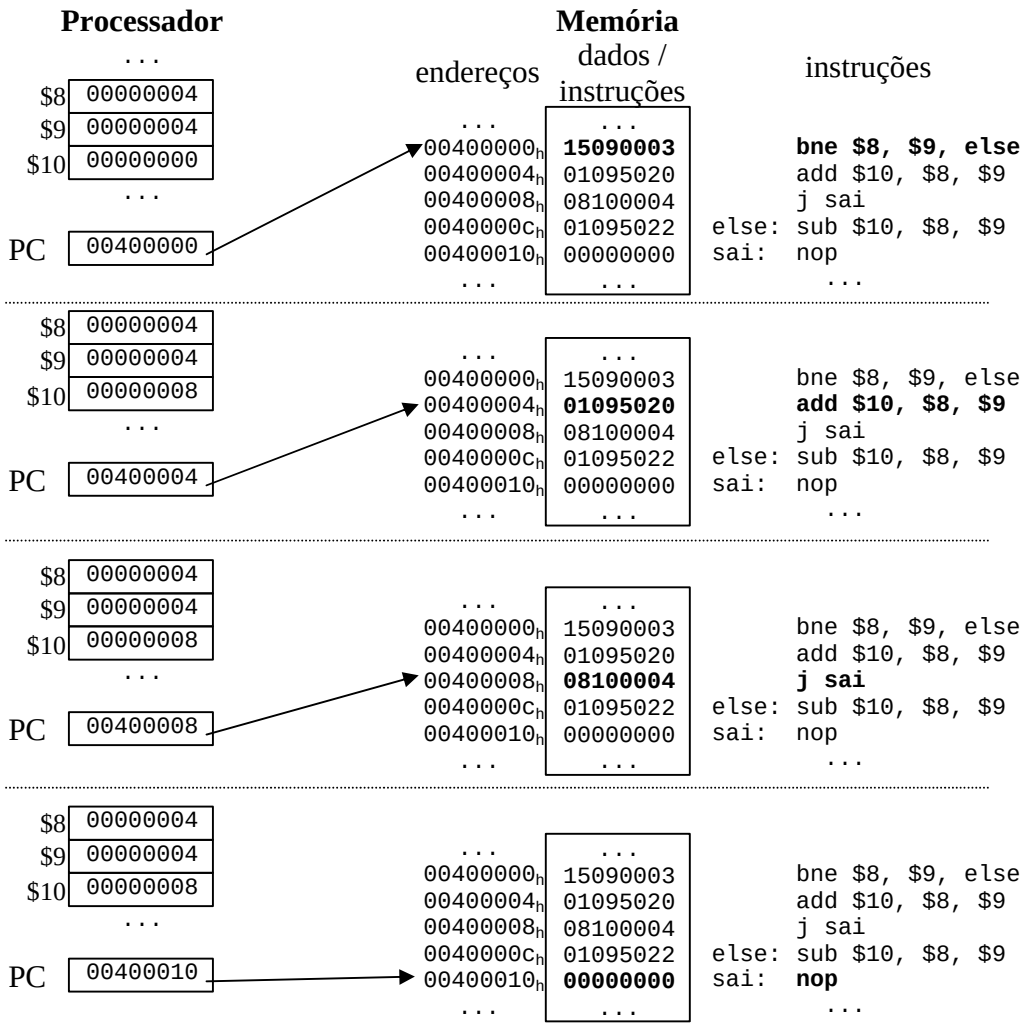


Figura 3.11: Execução de uma estrutura *if-then-else* no MIPS

A instrução `jal` opera exatamente da mesma forma. Apenas a funcionalidade de escrever o valor de PC+4 em `$ra` é acrescentada. A codificação segue a mesma lógica, mas o valor do `opcode` passa a ser 3. A Tabela 3.3 mostra a codificação das instruções do tipo J.

Nome	Formato	Exemplo	Codificação	
			opcode	
j	J	j 1000	2	1000
jal	J	jal 1000	3	1000

Tabela 3.3: Instruções da família J

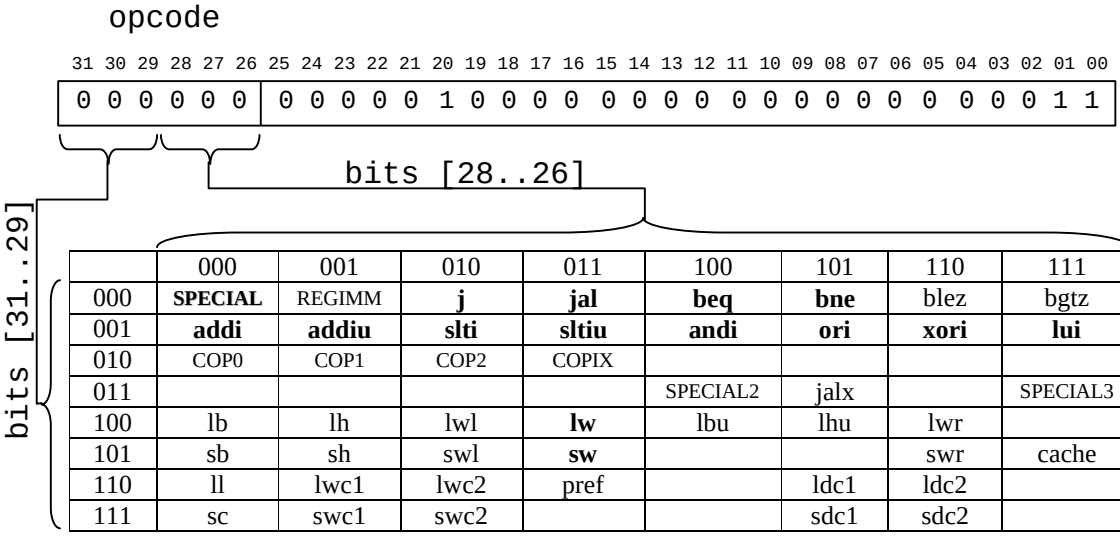
Agora vamos apresentar o quadro completo com os formatos utilizados pelas principais instruções do MIPS. Este quadro é muito didático, mas tende a ficar muito grande se o número de instruções também o for. Por isto apresentaremos também a forma mais convencional encontrada nos manuais das ISAs dos processadores.

Nome	**Formato**	**Exemplo**	**Codificação**					
			opcode	**rs**	**rt**	**rd**	**sa**	**function**
sll	R	sll $8, $9, 3	0	9	10	8	3	0
srl	R	srl $8, $9, 3	0	0	10	8	3	2
jr	R	jr $8	0	8	0	0	0	8
mfhi	R	mfhi $8	0	0	0	8	0	16
mflo	R	mflo $8	0	0	0	8	0	18
mult	R	mult $9, $10	0	9	10	0	0	24
multu	R	multu $9, $10	0	9	10	0	0	25
div	R	div $9, $10	0	9	10	0	0	26
divu	R	divu $9, $10	0	9	10	0	0	27
add	R	add $8, $9, $10	0	9	10	8	0	32
addu	R	addu $8, $9, $10	0	9	10	8	0	33
sub	R	sub $8, $9, $10	0	9	10	8	0	34
subu	R	subu $8, $9, $10	0	9	10	8	0	35
and	R	and $8, $9, $10	0	9	10	8	0	36
or	R	or $8, $9, $10	0	9	10	8	0	37
slt	R	slt $8, $9, $10	0	9	10	8	0	42
sltu	R	sltu $8, $9, $10	0	9	10	8	0	43
mul	R	mul $8, $9, $10	28	9	10	8	0	2
			opcode	**rs**	**rt**	**immediate**		
beq	I	beq $8, $9, 3	4	8	9	3		
bne	I	bne $8, $9, 3	5	8	9	3		
addi	I	addi $8, $9, 3	8	9	8	3		
addiu	I	addiu $8, $9, 3	9	9	8	3		
slti	I	slti $8, $9, 3	10	9	8	3		
sltiu	I	sltiu $8, $9, 3	11	9	8	3		
andi	I	andi $8, $9, 3	12	9	8	3		
ori	I	ori $8, $9, 3	13	9	8	3		
lui	I	lui $8, 3	15	0	8	3		
lw	I	lw $8, 4($9)	35	9	8	4		
sw	I	sw $8, 4($9)	43	9	8	4		
			opcode	**address**				
j	J	j 1000	2	1000				
jal	J	jal 1000	3	1000				

Tabela 3.4: Sumário da codificação das principais instruções do MIPS

A forma apresentada a seguir usa uma abordagem parecida com o que acontece de fato em um processador para que o mesmo possa **decodificar** a instrução. A decodificação significa encontrar a instrução que deverá ser executada para que todas os sinais elétricos sejam enviados à via de dados corretamente. A decodificação no MIPS ocorre em primeiro plano na observação dos 6 bits que formam o `opcode`. A Figura 3.12 mostra a decodificação inicial. Veja que em alguns as células da tabela estão grafadas em maiúsculo. Neste caso, não se trata de uma instrução específica, mas de uma ligação para outra tabela. Nós vamos mostrar apenas a tabela chamada SPECIAL, por isto ela está sublinhada. As instruções que nós estudamos estão destacadas em negrito. As demais instruções e tabelas ficam por conta da curiosidade do leitor.

Veja que neste tipo de tabela não existe informação sobre a codificação do restante dos campos. É preciso o auxílio de uma figura com os formatos e a listagem das instruções para cada formato. Os formatos estão na Figura 3.13.



	000	001	010	011	100	101	110	111
000	**SPECIAL**	REGIMM	**j**	**jal**	**beq**	**bne**	blez	bgtz
001	**addi**	**addiu**	**slti**	**sltiu**	**andi**	**ori**	**xori**	**lui**
010	COP0	COP1	COP2	COPIX				
011					SPECIAL2	jalx		SPECIAL3
100	lb	lh	lwl	**lw**	lbu	lhu	lwr	
101	sb	sh	swl	**sw**			swr	cache
110	ll	lwc1	lwc2	pref		ldc1	ldc2	
111	sc	swc1	swc2			sdc1	sdc2	

Figura 3.12: Tabela de codificação das instruções

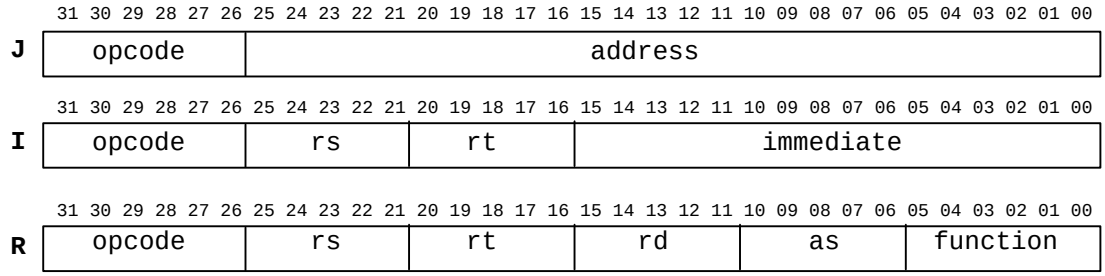
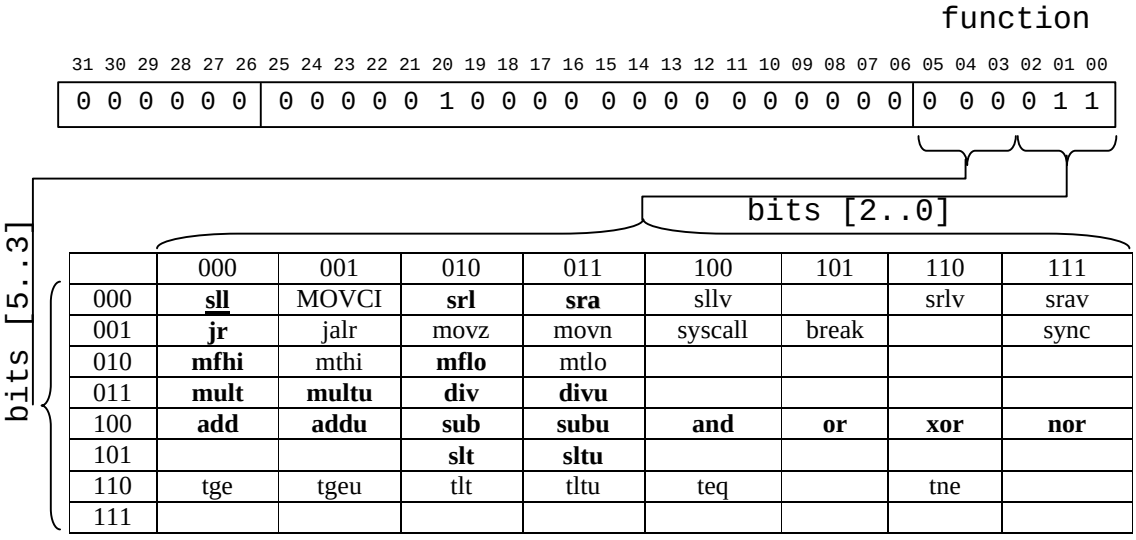


Figura 3.13: Codificação dos formatos



	000	001	010	011	100	101	110	111
000	**sll**	MOVCI	**srl**	**sra**	sllv		srlv	srav
001	**jr**	jalr	movz	movn	syscall	break		sync
010	**mfhi**	mthi	**mflo**	mtlo				
011	**mult**	**multu**	**div**	**divu**				
100	**add**	**addu**	**sub**	**subu**	**and**	**or**	**xor**	**nor**
101			**slt**	**sltu**				
110	tge	tgeu	tlt	tltu	teq		tne	
111								

Figura 3.14: Tabela de codificação das instruções

Uma vez feita a primeira decodificação é preciso saber se ainda é necessário a busca por mais uma tabela. É o caso das células que estão marcadas na tabela original como SPECIAL, entre outras. A Figura 3.14 mostra as instruções presentes nesta tabela. No exemplo da figura iríamos chegar a conclusão que se trata da instrução `sra`, cujo formato é R. Os restantes dos bits seriam decodificados nos campos `rs`, `rt`, `rd` e `as`.

Agora já chegamos a uma fase em que sabemos decodificar e montar qualquer instrução do MIPS (contida no texto). Vamos então abordar outros temas relevantes para este capítulo.

3.2 – Uso da linguagem de máquina

Nas seções seguintes iremos conhecer dois tópicos importantes na formação das linguagens de máquina. Como já mencionamos a complexidade dos sistemas eletrônicos nos levou a construir abstrações. Estas abstrações tampouco são manipuladas sem auxílio de ferramentas. Por isto, hoje ao projetarmos um processador vamos usar, em um primeiro estágio, uma linguagem de descrição de hardware, HDL (***Hardware Description Language***). Esta linguagem é um software capaz de descrever o comportamento de um hardware e de seus componentes. As mais famosas linguagens de descrição de hardware são VERILOG e VHDL. SystemC é recém-chegada ao grupo, mas está tendo um impacto muito importante na comunidade científica hoje.

Uma linguagem de descrição de hardware pode ser utilizada com dois propósitos: simular o comportamento da arquitetura e gerar um protótipo da mesma. Depois de simulada, a arquitetura passa por refinamentos constantes até atingir o ponto de satisfação desejado. Uma vez estável a descrição do hardware, um processo de síntese é iniciado. A síntese é a conversão do modelo descrito em uma HDL para um protótipo físico. Este processo também é feito com uma ferramenta, parecida com um compilador.

O processo de refinamento permite que partes do código sejam melhoradas e/ou detalhadas para que se tenha uma nova simulação e a corretude do modelo seja validada.

O processo de refinamento existe para prover uma melhor especificação do código, um melhor particionamento do projeto e a verificação de erros. Tudo isto, embora seja necessário, acaba consumindo muito tempo no projeto do sistema. Isto implica que o produto demora a chegar ao mercado. A Figura 3.15 Indica o processo básico de simulação e síntese de um projeto desenhado com uma HDL (SystemC).

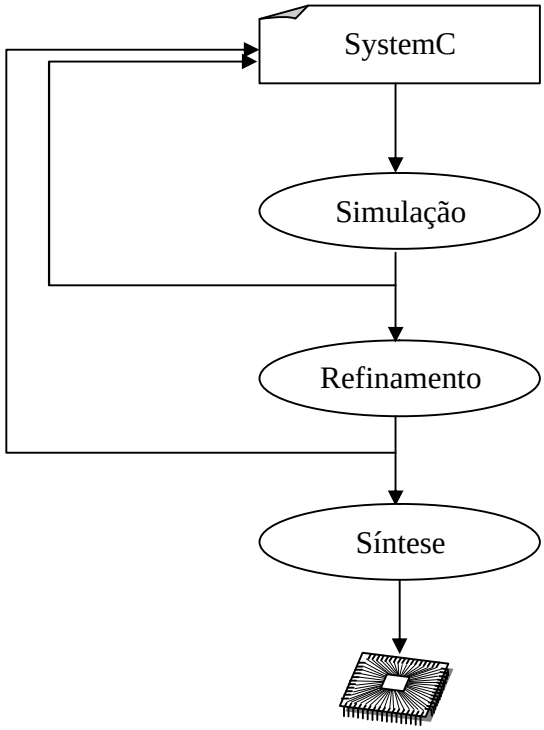


Figura 3.15: projeto de um sistema utilizando HDLs

Um produto para chegar ao mercado e fazer sucesso precisa ter associado a ele um conjunto de ferramentas básicas que permitam ao desenvolvedor de software utilizar os recursos da nova arquitetura. Este *toolkit* deve ser composto, no mínimo, de compiladores, montadores, ligadores e simuladores eficientes. Estas ferramentas, entretanto, são difíceis de serem geradas automaticamente a partir da descrição de um sistema em uma HDL porque as possibilidades de modelos de programação são muito variadas. Assim, estamos vendo hoje uma tendência no desenvolvimento de um projeto onde não mais utilizamos uma HDL, mas passamos a utilizar uma Linguagem de Descrição de Arquitetura, ADL (***Architecture Description Language***).

Um exemplo de ADL conhecida é LISA. Nós vamos estudar uma nova ADL, surgida recentemente, chamada ArchC, que gera código para SystemC. Baseados em ArchC, podemos criar automaticamente um montador e usar a mesma funcionalidade do SystemC para simulação da arquitetura. A geração automática do compilador para ArchC ainda está em desenvolvimento, mas existem técnicas de *re-target* do GCC no site da ferramenta.

Além de apresentar o ArchC na seção seguinte, vamos também explorar a anatomia de uma código executável. Quando compilamos um código, não somente informações de dados e instruções estão presentes no objeto gerado, mas também, e principalmente uma tabela de símbolos que permitem ao Sistema Operacional relocar o código para outras posições de memória. Vamos conhecer um pouco da especificação de um código executável no formato ELF (*Executable and Linking Format*), que é executado normalmente na maioria das distribuições linux.

Ambas as seções fazem parte da visão do software, que é de importância para o conhecimento de um aluno de um curso fortemente baseado em software.

3.3 – A visão do software – ArchC

O ArchC é uma ADL capaz de descrever o comportamento de uma arquitetura em diversos níveis de abstração. O modelo mais simples de descrição é o ***modelo funcional****, onde apenas as instruções são declaradas e suas funcionalidades sobre os recursos básicos da máquina, sem considerar, contudo, as inúmeras implicações que a temporização do modelo pode exercer sobre ele.*

A construção de um modelo funcional traz para o projetista a idéia de que as operações das quais ele precisa estão corretas e realizam as tarefas exatamente como proposto. Uma descrição neste nível de abstração requer muito pouco conhecimento da organização do computador e um modelo

simples de execução das instruções. A medida que refinamentos são feitos até chegarmos a um modelo mais complexo, acurado a ciclos, os recursos vão sendo redefinidos.

Neste capítulo iremos apresentar uma descrição funcional da ISA do MIPS no formato do ArchC.

O ArchC utiliza basicamente dois arquivos de entrada: o primeiro contendo a descrição da arquitetura e o segundo a descrição da ISA. A Figura 3.16 mostra o fluxo de projeto utilizando ArchC e a camada SystemC abaixo, até gerar uma especificação executável.

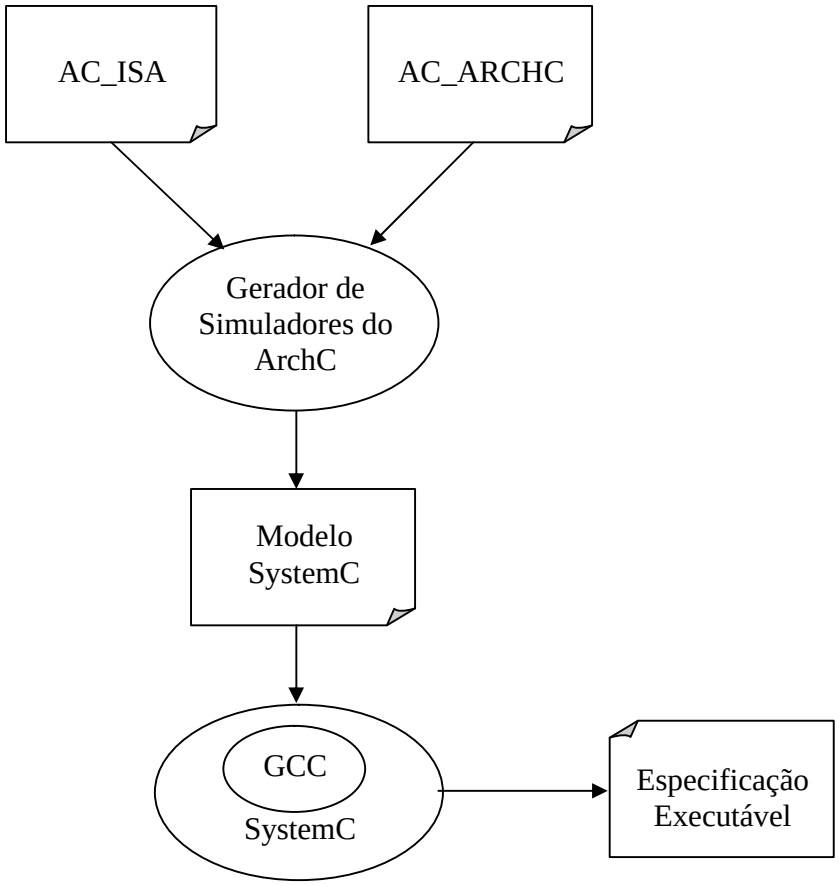


Figura 3.16: fluxo de projeto com ArchC

Dentro do arquivo de descrição do ISA (AC_ISA) o projetista especifica detalhes dos formatos das instruções, tamanhos e nomes e todas as informações necessárias para decodificação e para simulação do comportamento de cada instrução. O arquivo de Recursos de Hardware (AC_ARCH) contém informações acerca da memória, estrutura do pipeline etc.

A descrição de recursos que vamos utilizar é a mais simples possível, tipicamente para implementação de um modelo funcional. O arquivo `mips1.ac` *contém as linhas de código mostradas na Figura 3.17. O tamanho da palavra é 32 bits. Existe uma memória principal de 256kbytes e um banco de registradores com 34 registradores. O construtor do MIPS utiliza o arquivo "mips_isa.ac" para prover as informações sobre as instruções e o* endian *é setado para* big.

```
AC_ARCH(mips){
        ac_wordsize 32;
        ac_mem MEM:256k;
        ac_regbank RB:34;
        ARCH_CTOR(mips){
              ac_isa("mips_isa.ac");
              set_endian("big");
        };
};
```

Figura 3.17: Descrição do modelo funcional do MIPS em ArchC

A descrição do modelo funcional do ISA pode ser visto na Figura 3.18. Inicialmente são especificados os tipos e seus campos. Nesta especificação, `%op:6` *indica que existe um campo chamado* `op` *e cujo tamanho é de 6 bits. Os demais campos seguem o mesmo padrão.*

Em seguida são especificadas as instruções que utilizam cada um dos formatos descritos acima. `ac_instr<Type_J> j, jal;` *indica que as instruções* `j` *e* `jal` *possuem sua codificação binária seguindo o formato* `Type_J`*.*

Depois desta especificação é necessário indicar se existem apelidos para os registradores, o que é feito na seção `ac_asm_reg`*.*

Finalmente, o construtor do ISA especifica as instruções e a forma de decodificá-las. Por exemplo,

`lw.set_asm("lw %reg, %imm(%reg)", rt, imm, rs);`

indica que a instrução lw é formada por um registrador, um imediato e outro registrador. Os campos utilizados para estes três parâmetros são: `rt, imm` *e* `rs`*. Em seguida vem a decodificação que é feita exclusivamente pelo campo*

op, cujo valor deve ser 23_h (0x23 é outra forma de representar o valor em hexadecimal).

```
AC_ISA(mips){
  ac_format Type_R = "%op:6 %rs:5 %rt:5 %rd:5 0x00:5 %func:6";
  ac_format Type_I = "%op:6 %rs:5 %rt:5 %imm:16:s";
  ac_format Type_J = "%op:6 %addr:26";

  ac_instr<Type_R> add, addu, subu, multu, divu, sltu;
  ac_instr<Type_I> lw, sw, beq, bne, addi, andi, ori, lui, sltu;
  ac_instr<Type_J> j, jal;

  ac_asm_reg{
    "$"[0..31] = [0..31];
    "$zero" = 0;
    "$at" = 1;
    "$kt"[0..1] = [26..27];
    "$gp" = 28;
    "$sp" = 29;
    "$fp" = 30;
    "$ra" = 31;
  };

  ISA_CTOR(mips){
    lw.set_asm("lw %reg, %imm(%reg)", rt, imm, rs);
    lw.set_decoder(op=0x23);

    add.set_asm("add %reg, %reg, %reg", rd, rs, rt);
    add.set_decoder(op=0x00, func=0x20);
  };
}
```

Figura 3.18: Descrição do ISA do MIPS em ArchC

Uma vez descritos os dois arquivos, quando processados pelo gerador de simulador do ArchC é criado um arquivo de molde para a definição do comportamento de cada instrução. É também possível especificar um comportamento geral, para todas as instruções ou um comportamento específico para um tipo. Por exemplo, um trecho do molde do modelo do MIPS preenchido pode ser visto na Figura 3.19. A instrução lw por exemplo tem o seu comportamento descrito como,

`RB[rt] = DM.read(RB[rs]+ imm);`

Isto significa que o registrador apontado por `rt` *no banco de registradores irá receber o conteúdo da memória no endereço formado pela soma do valor* `imm` *da instrução e do valor guardado no registrador apontado por* `rs` *do banco de registradores.*

```
...
void ac_behavior( instruction ){
  ac_pc = ac_pc +4;
};

void ac_behavior( Type_R ){};

void ac_behavior( lw )
{
  RB[rt] = DM.read(RB[rs]+ imm);
};

void ac_behavior( slti )
{
  // Set the RD if RS< IMM
  if( (ac_Sword) RB[rs] < (ac_Sword) imm )
    RB[rt] = 1;
  // Else reset RD
  else
    RB[rt] = 0;
};
...
```

Figura 3.19: Comportamento do ISA do MIPS em ArchC

Veja também que existe um comportamento associado a todas as instruções: a soma do valor de PC (`ac_pc`). Esta variável pode ser reescrita por uma descrição de um comportamento em particular, como é o caso das instruções de salto.

Trechos das especificações do MIPS em ArchC foram apresentadas. Resta o leitor explorar a ADL para concluir a especificação de, ao menos, as instruções apresentadas até o presente momento.

3.4 – A visão do software – ELF

O formato de ligação e executável foi desenvolvido originalmente pela UNIX System Laboratories*. O Formato ELF foi um padrão projetado para ser executado em arquiteturas Intel de 32 bits numa variedade de sistemas operacionais. Hoje a plataforma que popularizou o ELF é a combinação da arquitetura Intel com o sistema operacional linux. Isto não significa que outras máquinas, de outras famílias não possam executar um arquivo neste formato.*

Existem três tipos de objetos que podem ser representados neste formato:

*- **um arquivo relocável**, que contém dados e instruções para serem ligados com outros arquivos objetos para criar um programa executável ou um arquivo objeto compartilhado;*

*- **um arquivo executável**, que contém informações suficientes para a criação de um processo pelo sistema operacional; e*

*- **um arquivo objeto compartilhado**, que contém informações suficientes para ligação em duas formas: o ligador pode ligá-lo com outro objeto compartilhado ou relocável; ou o ligador pode ligá-lo com um arquivo executável.*

Nós vamos nos ater ao formato de arquivo executável. Um arquivo executável normalmente contém algumas seções (trechos) de códigos e dados, que podem ser agrupados em segmentos. Tanto as seções como os segmentos estão, via de regra, dentro do próprio arquivo executável. Existe também um cabeçalho das seções, que é de fato uma tabela contendo informações sobre a localização, tipo e tamanho de cada seção; um cabeçalho de segmentos (chamado cabeçalho de programas), que indica localização, tipo e tamanho do segmento; e um cabeçalho geral que indica onde está cada cabeçalho no código.

A Figura 3.20 mostra a visão geral do formato. Interessante perceber que os segmentos possuem informações necessárias para o processo de carregamento do Sistema Operacional, portanto, um arquivo só será executável se possuir o cabeçalho de programas. Por outro lado, para ser ligável, é preciso utilizar informações particulares das seções, então um arquivo para ser ligado a outro pelo ligador precisa ter um cabeçalho de seções.

Os cabeçalhos geral, de seções e de programa possuem uma definição extremamente rígida e precisam seguir o padrão para que os conteúdos de cada parte do objeto sejam corretamente interpretados.

O cabeçalho geral é uma estrutura contendo 52 bytes que definem as informações contidas no arquivo ELF. Dentro deste cabeçalho estão definidos os deslocamentos, em bytes, desde o início do arquivo, do cabeçalho de programas e do cabeçalho de seções.

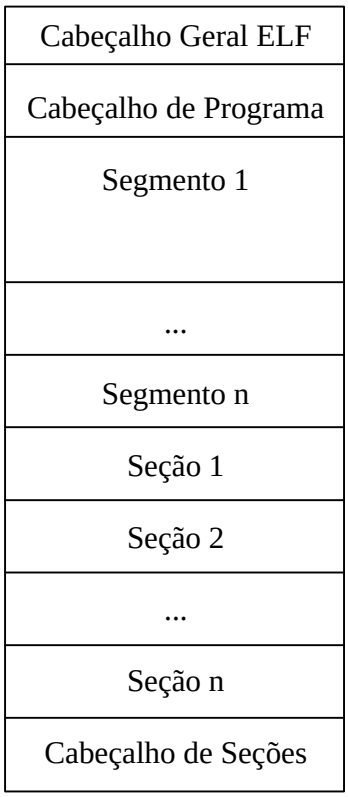


Figura 3.20: visão geral dos conteúdos de um arquivo executável e ligável no formato ELF

O cabeçalho de programas é uma tabela contendo em cada entrada informações sobre o segmento associado. O cabeçalho de seções segue o mesmo padrão, ou seja, contém uma tabela onde cada entrada contém informações sobre a seção associada. A Figura 3.21 ilustra esta hierarquia. Veja que no Cabeçalho Geral estão contidas as seguintes informações sobre o Cabeçalho de Programa: deslocamento desde o início do arquivo onde começa tal cabeçalho; tamanho de cada entrada no cabeçalho (todas as entradas têm o mesmo tamanho); e número de entradas na tabela.

Por sua vez, cada entrada do Cabeçalho de Programas tem, entre outras informações, o deslocamento (em bytes) desde o início do arquivo onde se encontra o Segmento associado e o tamanho deste Segmento. Isto ocorre também, de forma similar, com o Cabeçalho de Seções.

Vamos a seguir verificar o que existe de fato dentro de cada parte dos cabeçalhos.

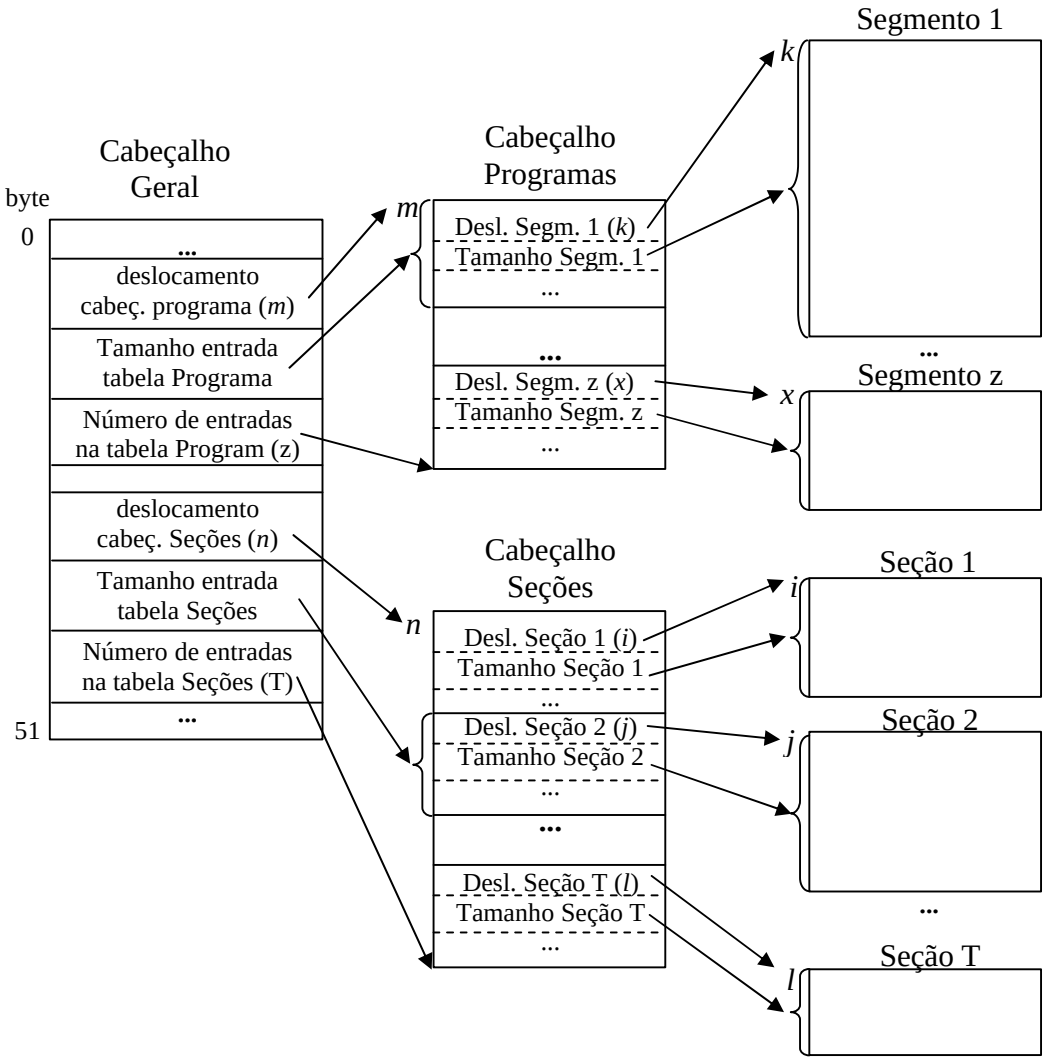


Figura 3.21: Cabeçalhos, Segmentos e Seções no formato ELF

O cabeçalho geral contém no primeiro byte um número específico: $7f_h$. Isto é obrigatório para identificação do tipo do arquivo, bem como a seqüência de bytes 45_h, $4c_h$, 46_h (que são os valores associados às letras E, L e F). Em seguida um byte especifica a `classe` *do arquivo: 0, classe inválida; 1, objeto de 32 bits; e 2 objeto de 64 bits. Atualmente a maioria das máquinas utilizam a classe 1. O próximo campo indica a forma como um número de 32 bits está organizado nos bytes. No caso de* **little endian** *o endereço de menor valor recebe o byte menos significativo, ou seja, mais à direita. No caso de* **big endian**, *o endereço menor recebe o byte mais*

significativo. Máquinas Intel normalmente utilizam little endian *e máquinas RISC como SPARC normalmente usam* big endian*. Há também máquinas que admitem os dois, como a MIPS. Finalmente chegamos à* `versão` *do cabeçalho usado. Hoje só utilizamos o valor 1. Os demais bits até o valor 15 podem ser usado para futuras especificações do ELF. No presente eles são postos todos em 0.*

Podemos já observar estes campos na Figura 3.22. O número apresentado sobre cada campo representa o byte do arquivo. Não há campos menores que um byte na especificação do ELF.

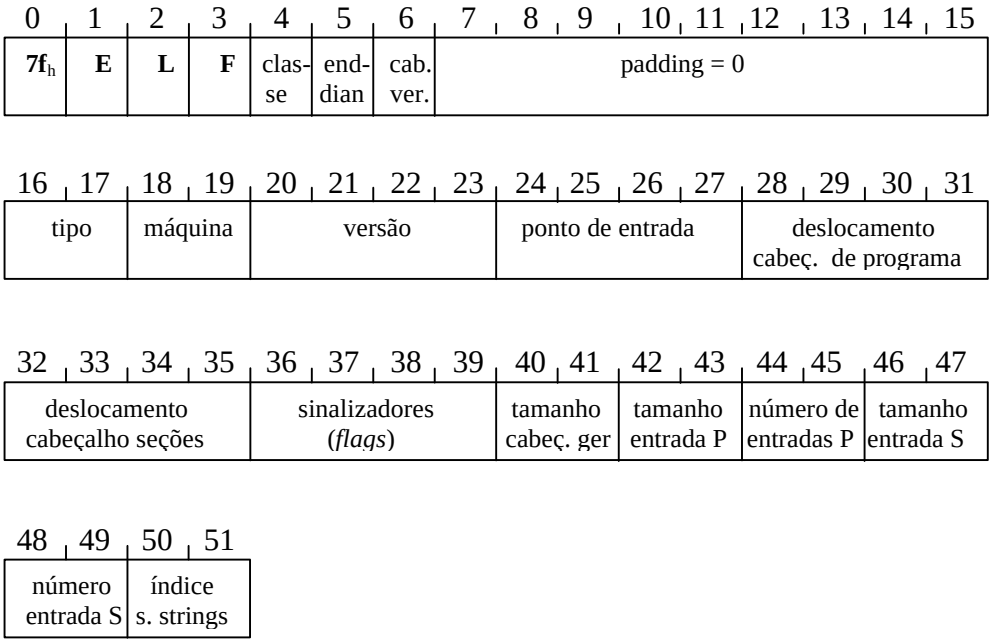


Figura 3.22: Cabeçalho Geral no formato ELF

O campo `tipo` *indica o tipo de objeto no formato ELF: 0, sem tipo; 1, arquivo de relocação; 2, arquivo executável; e 3, arquivo objeto compartilhado. Os nossos executáveis vão possuir então este campo assinalado como 1.*

O campo `máquina` *indica para qual máquina foi criado aquele objeto: 0, nenhuma; 1, AT&T WE 32100; 2, SPARC; 3, Intel 80386; 4, Motorola 68000; 5, Motorola 88000; 7, Intel 80860; e 8, MIPS RS3000. Outras máquinas podem ter valores associados no futuro.*

O campo `versão`*, indica a versão do objeto. A este campo é atribuído o valor 1 correntemente.*

O campo `ponto de entrada` *indica o endereço onde será iniciada a execução do programa. Um possível valor seria* 00400000_h *para o MIPS.*

Os próximos dois campos indicam em qual byte do arquivo começam as tabelas (cabeçalho) de programas e de seções respectivamente.

Os sinalizadores referem-se a bits de sinalização para máquina que irá executar o código. Este valor normalmente é deixado em 0.

Os bytes 40 e 41 informam o tamanho, em bytes, do cabeçalho geral e contém sempre o valor 34_h *(52).*

Os bytes 42 e 43 indicam o tamanho de cada entrada no cabeçalho de Programas, seguidos pelos bytes 44 e 45 que indicam quantas são estas entradas.

Os bytes 46 a 49 são os análogos à tabela de programas, mas agora associados com o cabeçalho de seções.

*Finalmente está indicado nos últimos bytes deste cabeçalho geral o número da seção que contém informações sobre os símbolos utilizados no programa, como por exemplo, o rótulo dos alvos de instruções de desvio e os nomes dos procedimentos. Esta seção especial contém cadeias de caracteres (*strings*) terminadas com o byte 0.*

Vamos agora analisar os dados presentes no cabeçalho de programas. A Figura 3.23 indica os campos presentes em cada entrada do cabeçalho. O campo `tipo` *indica que tipo de segmento está associado a esta entrada. O tipo mais comum é o 1, que indica um segmento que será carregado na memória para execução. O tamanho do segmento é especificado no campo* `tamanho segmento` *e a quantidade de memória necessária é especificada no campo* `tamanho memória`*. Em alguns casos o tamanho da memória requerida é maior que o tamanho do segmento, mas nunca ao contrário.*

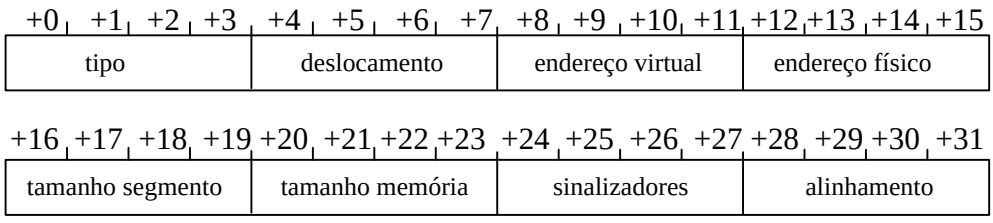


Figura 3.23: Cabeçalho de programas no formato ELF

O campo `endereço virtual` *diz o endereço virtual que deve ser utilizado para receber a primeira palavra do segmento (veremos o que é endereço virtual no capítulo 6).*

O campo `endereço físico` *diz o endereço físico que deve ser utilizado para receber a primeira palavra do segmento (veremos o que é endereço físico no capítulo 6).*

O campo `deslocamento` *indica o byte do arquivo onde se encontra o primeiro byte do segmento. Os demais campos ficam por conta da curiosidade do leitor em saber mais.*

Esta estrutura de dados se repete tantas vezes quantas forem as entradas do cabeçalho de programa.

Informações semelhantes são encontradas em cada entrada do cabeçalho de seções. Vamos analisar os principais campos de uma entrada como apontados na Figura 3.24.

O campo `nome` *indica um deslocamento dentro da seção especial de cadeias de caracteres onde está o nome da seção.*

O campo tipo indica o tipo de seção. Existem muitas definições de tipos possíveis. Os principais estão mostrados na Tabela 3.5. Certamente a de maior impacto é a de bits de programa, que contém as informações das instruções e dos dados do programa.

Os sinalizadores de cada seção são muito importantes. Existem 3 sinalizados específicos: WRITE, valor 1, ALLOC, valor 2 e EXECINSTR, valor 4. WRITE significa que a seção contém dados que podem ser escritos (além de lidos). ALLOC significa que a seção irá ocupar espaço na memória e EXECINSTR significa que a seção contém instruções em linguagem de máquina. Os sinalizadores podem ser usados em combinações. Por exemplo, o valor 6 (2+4) indica que a seção é EXECINSTR e ALLOC ao mesmo tempo.

O campo `endereço` *indica o endereço na memória onde esta seção deve aparecer, caso ela aloque espaço na memória.*

O campo `deslocamento` *indica o primeiro byte onde está, no arquivo, a seção associada.*

O campo `tamanho` *indica o tamanho da seção, em bytes. Os demais campos ficam como exercícios.*

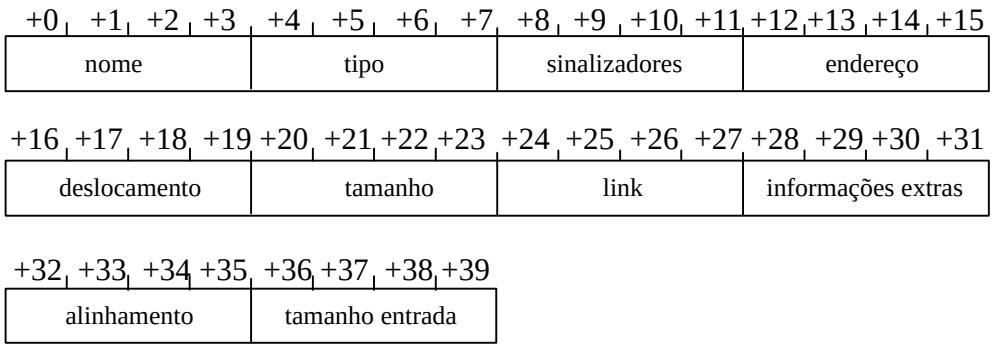


Figura 3.24: Cabeçalho de seções no formato ELF

Tipo	Valor	Descrição
NULL	0	Seção inexistente
PROGBITS	1	Bits do programa
SYMTAB	2	Tabela de Símbolos
STRTAB	3	Tabela de Strings
RELA	4	Informações de Relocação
HASH	5	Tabela de Espalhamento de Símbolos
DYNAMIC	6	Ligação dinâmica
NOTE	7	Anotações sobre o arquivo
NOBITS	8	Não ocupa espaço no arquivo
REL	9	Informações de Relocação
SHLIB	10	Sem semântica específica

Tabela 3.5: Tipos de Seções do ELF

Conhecer esta anatomia de um arquivo executável é importante para aqueles que querem aprender um pouco mais sobre programação. Aqui apresentamos um único formato, mas existem outros especificados, por exemplo, para o Sistema Operacional DOS e Windows. Ainda, existe outra anatomia para Java Bytecodes. Conhecer uma delas é importante para que as demais possam ser mais facilmente entendidas.

3.5 – Conclusões

Conhecemos neste capítulo como é a codificação binária de uma porção das instruções do MIPS. Não tratamos, entretanto de todas as instruções, como instruções de ponto flutuante ou de coprocessador. O que importa neste instante é que o leitor possa fazer uma tradução de código de máquina para código de montagem e vice-versa.

Vimos também que o número de formatos disponíveis no MIPS é de três. Isto pode parecer um número muito pequeno, mas existe um benefício muito importante associado. A decodificação de uma instrução é feita pela máquina e quanto mais formatos existirem, mais possibilidades de combinações de bits existirão. Isto tem implicação direta no desempenho do decodificador. Então, a escolha de um pequeno número de formatos melhora o desempenho da máquina.

Para finalizar, estudamos como é a anatomia de um arquivo executável no formato ELF. O conhecimento adquirido ao dividir os campos das instruções é diretamente aplicado para o reconhecimento do formato ELF. Não nos cabe, entretanto, discutir os detalhes do formato. A apresentação foi geral, mas com ela é possível extrair a maioria das informações no formato. Em particular uma seção muito especial precisa ser abordada: a seção de cadeia de caracteres (*strings*).

Cada seção também têm nome e eles podem indicar um tipo muito particular de seção. As seções mais conhecidas são: `.init` que contém um trecho de código que deve ser executado antes da chamada ao programa principal; `.text` que contém as instruções do programa propriamente dito; `.fini`, que é executada depois da finalização do código do programa em `.text`. `.init` e `.fini` executam instruções que contribuem para a criação e finalização do processo pelo Sistema Operacional. A seção `.data` contém os dados do programa. A seção `.rodata` (*read-ony data*) contém os dados que não serão passivos de escrita durante a execução do código. A seção `.strtab` contém *strings* usadas no programa e a seção `.symtab` contém informações sobre os símbolos utilizados.

Juntado os nossos conhecimentos, aprendemos a criar uma descrição de uma arquitetura em uma ADL, a ArchC. Fizemos um esboço de uma descrição do MIPS usando a linguagem. Usamos um modelo funcional que esconde os detalhes da organização do computador, mas explicita os detalhes da construção do conjunto de instruções. É um bom exercício completar a especificação.

3.6 – Prática com Simuladores

Usando o SPIM podemos observar como é a codificação binária de cada instrução que executamos, basta olhar a coluna que mostra o valor hexadecimal equivalente.

Um outro exercício com simuladores, envolve a prática com o ArchC. O leitor se sinta impelido a instalar o SystemC e o ArchC para poder fazer simulações de descrição de ISAs.

Por fim, o programa `objdump` pode ser útil na decodificação automática (*desassembler*) de arquivos do formato ELF. Mas para ir além seria interessante usar um programa que apresente o código hexadecimal de cada byte de um arquivo, como o `HexEdit`.

3.7 – Exercícios

3.1 – Pesquise na web e produza um resumo sobre a seção strtab.

3.2 – Pesquise na web e produza um resumo sobre como especificar *pseudo-instruções* do MIPS em ArchC.

3.3 – Para cada código criado nos exemplos do capítulo 2 encontre a codificação em linguagem de máquina do MIPS.

3.4 – Preencha a Tabela 3.6 como o exemplo.

3.5 – Dado o arquivo ELF a seguir, indique qual o programa que será carregado na memória para ser executado.

# byte	Conteúdo
00000000	**7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00**
00000010	**00 02 00 08 00 00 00 01 00 40 00 00 00 00 00 34**
00000020	**00 00 00 a0 00 00 00 00 00 34 00 20 00 01 00 28**
00000030	**00 02 00 00 00 00 00 01 00 00 00 60 00 40 00 00**
00000040	**00 40 00 00 00 00 00 18 00 00 00 18 00 00 00 07**
00000050	**00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00**
00000060	**01 09 58 2a 11 60 00 03 01 09 50 20 08 10 00 05**
00000070	**01 09 50 22 00 00 00 00 00 00 00 00 00 00 00 00**
00000080	**01 09 58 2a 11 60 00 03 01 09 50 20 08 10 00 05**
00000090	**01 09 50 22 00 00 00 00 00 00 00 00 00 00 00 00**
000000a0	**00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00**
000000b0	**00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00**
000000c0	**00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 01**
000000d0	**00 00 00 06 00 04 00 00 00 00 00 80 00 00 00 18**
000000e0	**00 00 00 00 00 00 00 00 00 00 00 04 00 00 00 00**

Categoria	Nome	Formato	Exemplo	Operação (ArchC)	Comentários
Aritmética	add	R	add rd, rs, rt	RB[rd] = RB[rs] + RB[rt];	*Overflow* gera exceção
	sub				*Overflow* gera exceção
	addi				*Overflow* gera exceção Valor do imediato na faixa entre –32.768 e +32.767
	addu				*Overflow* não gera exceção
	subu				*Overflow* não gera exceção
	addiu				*Overflow* não gera exceção Valor do imediato na faixa entre 0 e 65.535
	mul				*Overflow* não gera exceção HI, LO imprevisíveis após a operação
	mult				*Overflow* não gera exceção
	multu				*Overflow* não gera exceção
	div				*Overflow* não gera exceção
	divu				*Overflow* não gera exceção
lógicas	or				
	and				
	xor				
	nor				
	andi				Imediato em 16 bits
	ori				Imediato em 16 bits
	sll				Desloc. ≤ 32
	srl				Desloc. ≤ 32
	sra				Desloc. ≤ 32. Preserva sinal
Transferência de dados	mfhi				
	mflo				
	lw				
	sw				
	lui				Carrega constante na porção alta do registrador de destino. Zera a parte baixa.
Suporte a decisão	bne				
	beq				
	j				
	slt				Opera com números sinalizados
	sltu				Opera com números não sinalizados
Suporte a procedimentos	jal				
	jr				

Tabela 3.6: Instruções do MIPS