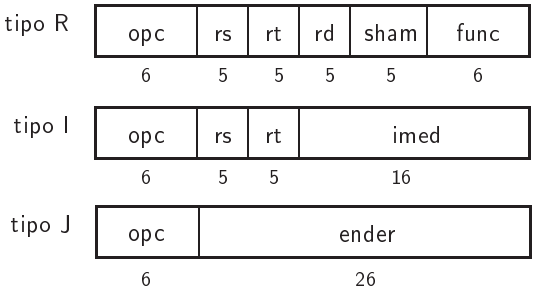
Revisão – Modos de Endereçamento vs Codificação

- **a registrador:** operandos e destino em registradores
- **imediato:** constante é parte da instrução
- **base-deslocamento:** end_efetivo = reg + deslocamento
- **relativo a PC:** end_efetivo = PC + deslocamento
- **(pseudo)absoluto:** end_efetivo é parte da instrução



Qual a relação entre codificação e modos de endereçamento?

Suporte a Subrotinas

- Endereço de retorno é o endereço da instrução após a instr que muda o fluxo de execução
- endereço de retorno só é conhecido em tempo de execução
- no MIPS o endereço de retorno é sempre armazenado em $31
- a chamada de função no MIPS é
  `jal EnderDaFuncao # jump-and-link [$31←PC+4]`
  que faz o salto e carrega o endereço de retorno (PC+4) em $31.
- a última instrução da função deve ser
  `jr $31 # jump-register [PC← $31]`
  cujo efeito é copiar o conteúdo de $31 para o PC, retornando para a instrução seguinte à invocação = (PC+4)

Suporte a Subrotinas – exemplo

```
main:
     ...
2000 jal  40.8000    # salva o ender de retorno em r31
2004 add  r16,r14,r2 # ESTE é o ender de retorno de B()
     ...
B:
8000 sw r5,0(sp)
     ...
```

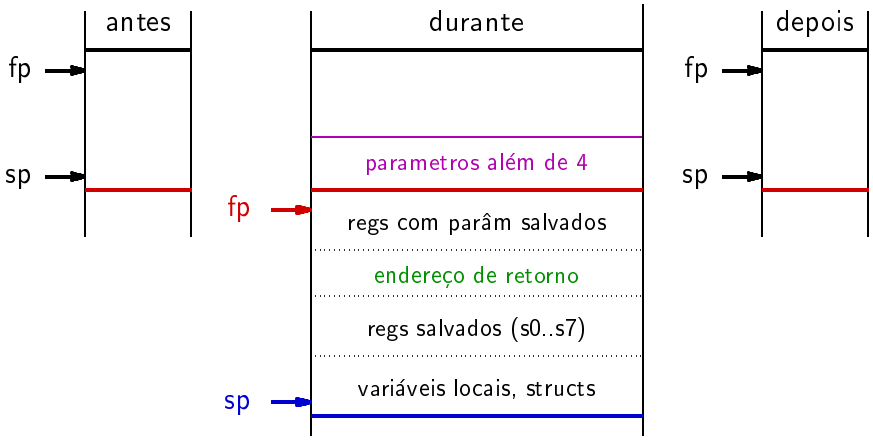
Suporte a Subrotinas

- Uma função deve salvar em memória registradores que modifica e que são usados pela função que a invocou
- A estrutura de dados onde os registradores são salvos é uma pilha.
- $31 deve ser salvo na pilha antes que outra função seja invocada.
- convenções do MIPS:
  - *caller save*: quem salva os registradores é quem chama
  - *callee save*: quem salva os registradores é a função chamada
  - endereço de retorno (return address) é $31 ($ra)
  - apontador de pilha (stack pointer) é $29 ($sp)
  - montador usa $4..$7 para passar 4 parâm em regs $a0-$a3
  - 5^{o} parâmetro e seguintes na pilha (antes de chamar função)
  - valores são retornados em $2 e $3 $v0, $v1

Registradores e valores usados em funções

preservados		destruídos	
s0..s7	salvados	t0..t9	temporários
sp	apont de pilha	a0..a3	argumentos
ra	ender de retorno	v0..v1	resultados
gp	*global pointer*	k0..k1	usados pelo SO
		at	*assembler temporary*
	pilha acima do sp		pilha abaixo do sp

Registro de ativação



Compiladores

Função do compilador:

- todos os programas corretos executam corretamente
- maioria dos programas compilados executa rapidamente
- compilação rápida
- suporte a depuração

Conjunto de Instruções é "entrada" do compilador

Instruction Set Architecture = ISA
Conjunto de Instruções = CdI
= parte visível ao programador e ao compilador
contrato entre hardware e software

- simplifica compilador se conjunto é
  - ∗ ortogonal — o que um pode, todos podem
  - ∗ regular — coisas similares nos mesmos lugares
  - ∗ facilita composições — $\sum$operações simples = complexa
- codificação simples e regular simplifica hardware
  - ∗ operações — popularidade vs implementação
  - ∗ operandos — popularidade vs implementação
  - ∗ endereçamento de operandos — código compacto vs flexibilidade
  - ∗ codificação — código compacto vs decodificação

Compiladores – do que eles gostam?

- quem escreve um compilador deseja
  - ⋆ regularidade — simplifica análise de casos
  - ⋆ ortogonalidade — suporta todas as combinações
  - ⋆ composabilidade — primitivas ao invés de soluções
  - ⋆ as três permitem operações simples combinadas em operações complexas
- compiladores efetuam análise de casos gigantesca
  - ⋆ opções demais dificultam escolhas
- conjuntos de instruções ortogonais quanto a
  - ⋆ operações
  - ⋆ tipos de dados
  - ⋆ modos de endereçamento
  - ⋆ completude
    - uma, ou — *condições de desvio* $\Longrightarrow$ *eq lt*
    - todas as soluções, — *condições de desvio* $\Longrightarrow$ *eq ne lt gt le ge*
    - mas não só algumas escolhas idiossincráticas

Conjunto de Instruções do MIPS

Projeto de RISCs na década de 80 para obter implementação segmentada num único CI

Reduced Instruction Set Computers
reduced == simples, e não "pequeno"

- ênfase em
  - decodificação rápida
  - instruções com tamanho fixo
  - codificação regular
- compilador poderia escalonar instruções para execução
- código maior que equivalentes CISC (C=*complex*)

MIPS

- endereços alinhados de 32 bits
- modo de endereçamento é *deslocamento* load/store
- tipos de dados simples
- registradores
  - 32 regs de uso geral, de 32 bits (R0 = 0)
  - 16 regs de ponto flutuante de 64 bits (ou 32 de 32bits) regsPF
  - registrador de status para ponto flutuante
  - sem registrador de status para inteiros ∄CondCodeReg
- três formatos de instrução com mesmo tamanho

MIPS - modos de endereçamento

modo	endereço efetivo	exemplo
a registrador	R	add r4,r3,r2
imediato	imed	add r4,#8
deslocamento	M[R+imed]	add r4,100(r1)
indireto a registrador	[R]	jr r4
absoluto	ender	j ender

MIPS

- transferência de dados
  - load/store byte/half/word/doubleword
  - load/store PF single/double
  - move de-para regs e regsPF
- lógica e aritmética
  - add/sub/mult/div
  - and/or/xor
  - sll/srl (lógicos), sra (aritmético) deslocamentos
  - loadHigh (usado para constantes de 32 bits)

MIPS

- ponto flutuante
  - add/sub/mult/div single/double
  - conversões de-para inteiros
  - desvios (liga/desliga bits para desvios)
- controle
  - desvios condicionais: $=0$ $\neq 0$ bits_PF
  - jump/jr jump-register
  - jal jump-and-link-register
  - trap/rte return from exception

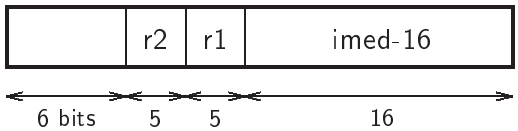
MIPS

Formatos das instruções

- formato-I
  - instruções de ALU com imediatos
  - load e store
  - desvios condicionais
  - jump-register

imediato

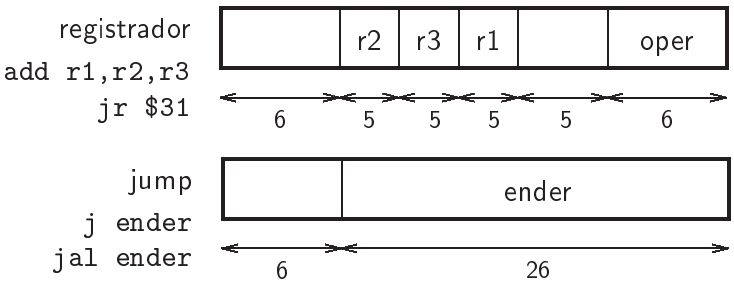
```
addi r1,r2,const
 lw $8,desl($15)
```



MIPS

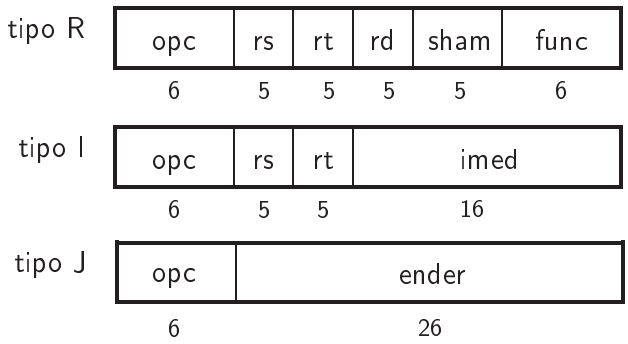
Formatos das instruções

- formato-R
  - instruções de ALU com três operandos
- formato-J
  - saltos incondicionais



MIPS

- instruções regulares facilitam decodificação → **hw rápido**
- instruções simples facilitam construção do compilador e geração de código → **sw rápido**
  e ainda circuitos simples → **hw rápido**



Endereçamento em CISCs (≠ do MIPS)

Complex Instruction Set Computers

Auto-incremento, auto-decremento tem no PowerPC

Ideal para andar em vetores ou operações em pilhas.

```
# MIPS: ------------------------------------------
lw   $8, sSt($19)    # $8← S[$19]
addi $19, $19, 4     # S[i+1], i = $19

# CISC: ------------------------------------------

lw+  $8, sSt($19)    # $8← S[$19], $19← $19+4
lb+  $8, sSt($19)    # $8← S[$19], $19← $19+1 OCTETOS
```

Auto-incremento = pop, auto-decremento = push

Endereçamento em CISCs (≠ do MIPS)

Operandos em memória

```
# MIPS: ------------------------------------------
lw   $8, 0($19)
add  $16, $17, $8         # $16 = $17 + mem[$19]

# CISC: ------------------------------------------
addm $16, $17, 0($19)     # $16 = $17 + mem[$19]
```

- Problemas:
  - addm usa 3 registradores **mais** um imediato
    → instruções com tamanho variável
  - no VAX, qualquer operando pode estar em memória
    → implementação fica complicadíssima, e portanto, lenta

Endereçamento em CISCs (≠ do MIPS)

Instruções complexas

tem no PowerPC

```
for (i = 0; i < j; i++) { ...  }

# MIPS: ------------------------------------------
Loop:  ...
addi $19, $19, 1       $19=i, $20=j, $8=tmp
slt  $8, $19, $20      # $8 ← 1 se $19 < $20
bne  $8, $0, Loop      # desvie se i<j ($19 < $20)

# CISC: ------------------------------------------
Loop:  ...             # increment-compare-and-branch
icb  $19, $20, Loop    # $19 = $19+1; desvie se $19<$20
```


Endereçamento em CISCs (≠ do MIPS)

Instruções complexas

- Problemas com instruções complexas:
  - `icb $19,$20,Loop` serve somente para o laço acima
    → não é geral o bastante para ser **realmente** útil
  - implementação aumenta muito a complexidade do hardware
    e portanto faz toda a máquina ficar mais lenta
    conforme veremos nos capítulos 5 e 6