ESCOLA SUPERIOR DOM HÉLDER CÂMARA

CIÊNCIA DA COMPUTAÇÃO

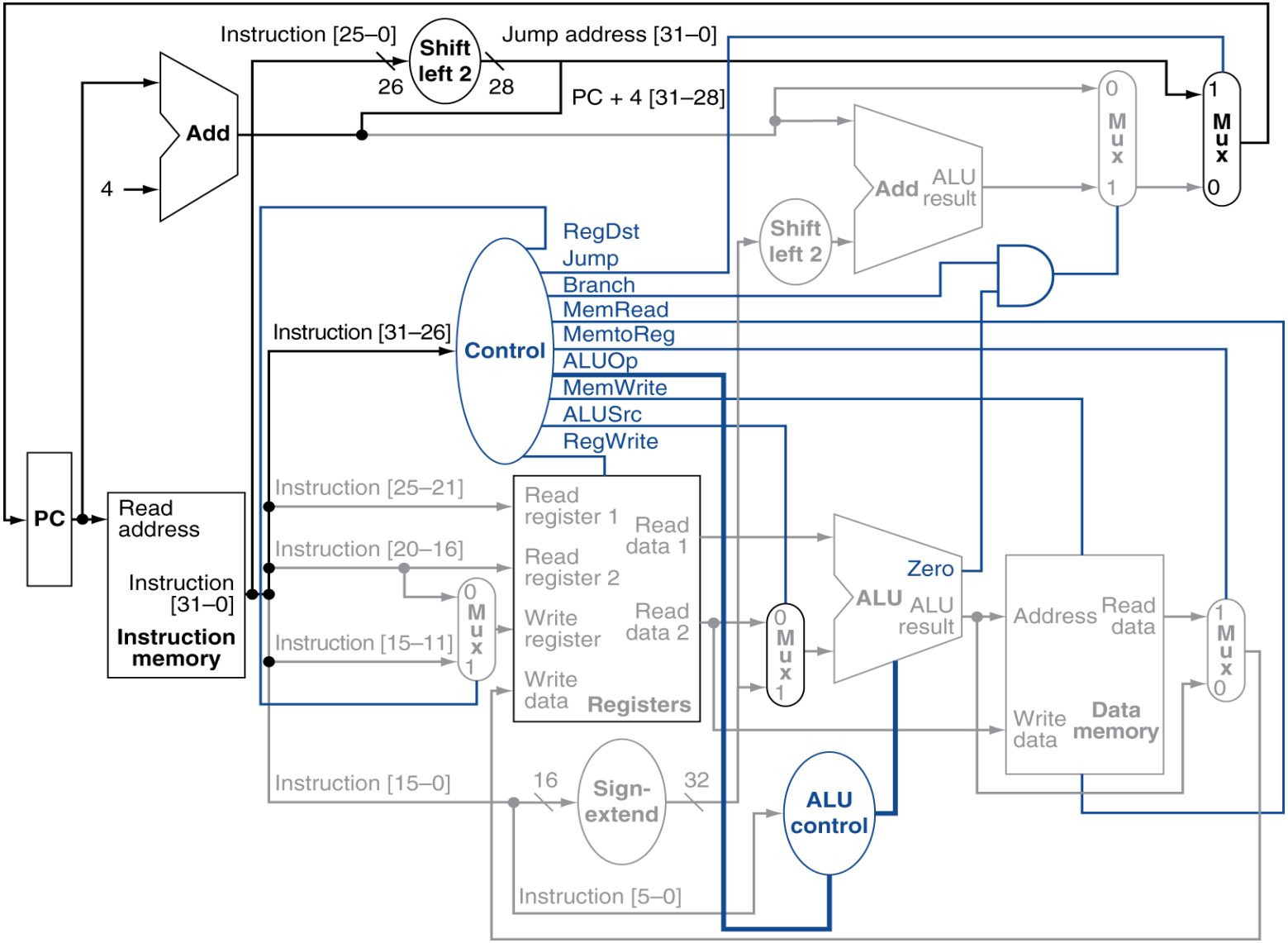
ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES I

Dom Helder
ESCOLA SUPERIOR

CAPÍTULO 2

INSTRUÇÕES: A LINGUAGEM DOS COMPUTADORES

Como uma instrução é executada?



Instruções

- Linguagem de Máquina: instruções.
- Conjunto de instruções: vocabulário ou língua na qual nos comunicamos com o processador.
- Nós trabalharemos com a arquitetura do conjunto de instruções do MIPS:
  - Microprocessor Without Interlocked Pipeline Stages
  - Conjunto de instruções mais simples.
  - RISC - Reduced Instruction Set Computer.
- Objetivo: manter a simplicidade para que o foco do estudo esteja no conceito, não no processador.

Programa Armazenado

- A “simplicidade do equipamento” é uma consideração tão valiosa para os computadores de hoje, quanto foi para os da década de 1950.
- Um conjunto de instruções deve seguir esse conselho, mostrando como ele é representado no hardware e o relacionamento entre as linguagens de programação de alto nível e essa linguagem mais primitiva.
- Aprendendo como representar as instruções, você também descobrirá o segredo da computação: o conceito de programa armazenado.
  - As instruções e os dados de muitos tipos podem ser armazenados na memória como binários.

Programa Armazenado

- Os processadores de uso geral devem ser capazes de executar os seguintes tipos de operações:
  - operações aritméticas;
  - operações lógicas;
  - operações relacionais;
  - operações de ponto flutuante;
  - transferência de dados;
  - desvios condicionais;
  - desvios incondicionais;
  - controle;
  - etc...

Arquitetura do Conjunto

- A Arquitetura do Conjunto de Instruções define os tipos de instruções que serão executadas por um processador, assim como o formato de cada instrução, quantidade de bits, a forma como essas instruções acessarão registradores e memórias (modos de endereçamento), a forma como conversarão com outras unidades funcionais, etc.
- A Arquitetura do conjunto de instruções será aqui utilizada para referir ao verdadeiro conjunto de instruções visíveis pelo programador.
- É a interface Hardware/Software!

MIPS

- Princípios fundamentais de projeto MIPS:
  - “Simplicidade favorece a regularidade”
  - “Menor significa mais rápido”
  - “Agilize os casos mais comuns”
  - “Um bom projeto exige bons compromissos”.

Classes de ISA

- Quase todas as arquiteturas de conjunto de instruções são classificadas como arquiteturas de registradores de propósito geral (GPRs), em que os operandos são registradores ou locais de memória.
- 80x86: 16 GPRs e 16 que podem manter dados de ponto flutuante (FPRs);
- RISC-V e MIPS: 32 GPRs e 32 FPRs.
- As duas versões populares dessa classe são arquiteturas de conjunto de instruções registrador-memória, como o 80x86, que podem acessar a memória como parte de muitas instruções, e arquiteturas de conjunto de instruções load-store, como o ARMv8, RISC-V e MIPS, que só podem acessar a memória com instruções load ou store.
- Todas as arquiteturas de conjunto de instruções desde 1985 são load-store.

Endereçamento de Memória

- Praticamente todos os computadores desktop e servidores utilizam endereçamento de byte para acessar operandos da memória.
- Algumas arquiteturas, como ARMv8 e MIPS, exigem que os objetos estejam alinhados.
- Um acesso a um objeto com tamanho de s bytes no endereço de byte A está alinhado se A mod s = 0.
- O 80x86 e o RISC-V não exigem alinhamento, mas os acessos geralmente são mais rápidos se os operandos estiverem alinhados.

Tipos e Tamanho de Operandos

- Assim como a maioria das arquiteturas de conjunto de instruções, o 80x86, o ARMv8 e o RISC-V admitem tamanhos de operando de 8 bits (caractere ASCII), 16 bits (caractere Unicode ou meia palavra), 32 bits (inteiro ou palavra), 64 bits (dupla palavra ou inteiro longo) e ponto flutuante IEEE 754 com 32 bits (precisão simples) e 64 bits (precisão dupla).
- O 80x86 também admite ponto flutuante de 80 bits (precisão dupla estendida)

Operações

- As categorias gerais de operações são transferência de dados, lógica e aritmética, controle (analisado em seguida) e ponto flutuante.
- O RISC-V e o MIPS são arquiteturas de conjunto de instruções simples e fáceis de executar em pipeline.
- O 80x86 possui um conjunto de operações maior e muito mais rico.

Codificando uma ISA

- Existem duas opções básicas na codificação: tamanho fixo e tamanho variável.
- Todas as instruções do ARMv8 e RISC-V possuem 32 bits de extensão, o que simplifica a decodificação da instrução.
- As instruções de tamanho variável podem ocupar menos espaço que as instruções de tamanho fixo, de modo que um programa compilado para o 80x86 normalmente é menor que o mesmo programa compilado para RISC-V.
- As opções mencionadas anteriormente afetarão o modo como as instruções são codificadas em uma representação binária.
  - Por exemplo, o número de registradores e o número de modos de endereçamento possuem impacto significativo sobre o tamanho das instruções, pois o campo de registrador e o campo de modo de endereçamento podem aparecer muitas vezes em uma única instrução.

Codificando uma ISA

- O ARMv8 e o RISC-V, mais tarde, ofereceram extensões, chamadas Thumb-2 e RV64IC, que oferecem uma mistura de instruções com 16 bits e 32 bits de extensão, respectivamente, para reduzir o tamanho do programa.
- O tamanho do código para essas versões compactas das arquiteturas RISC são menores que o do 80x86.

Arquiteturas

- A implementação de um computador possui dois componentes: organização e hardware.
- O termo organização inclui os aspectos de alto nível do projeto de um computador, como o sistema de memória, a interconexão de memória e o projeto do processador interno ou CPU (unidade central de processamento, na qual são implementados a aritmética, a lógica, os desvios e as transferências de dados).
- O termo microarquitetura também é usado no lugar de organização.
  - Por exemplo, dois processadores com as mesmas arquiteturas de conjunto de instruções, mas com organizações diferentes, são o AMD Opteron e o Intel Core i7.
  - Ambos implementam o conjunto de instruções 80x86, mas possuem organizações de pipeline e cache muito diferentes.

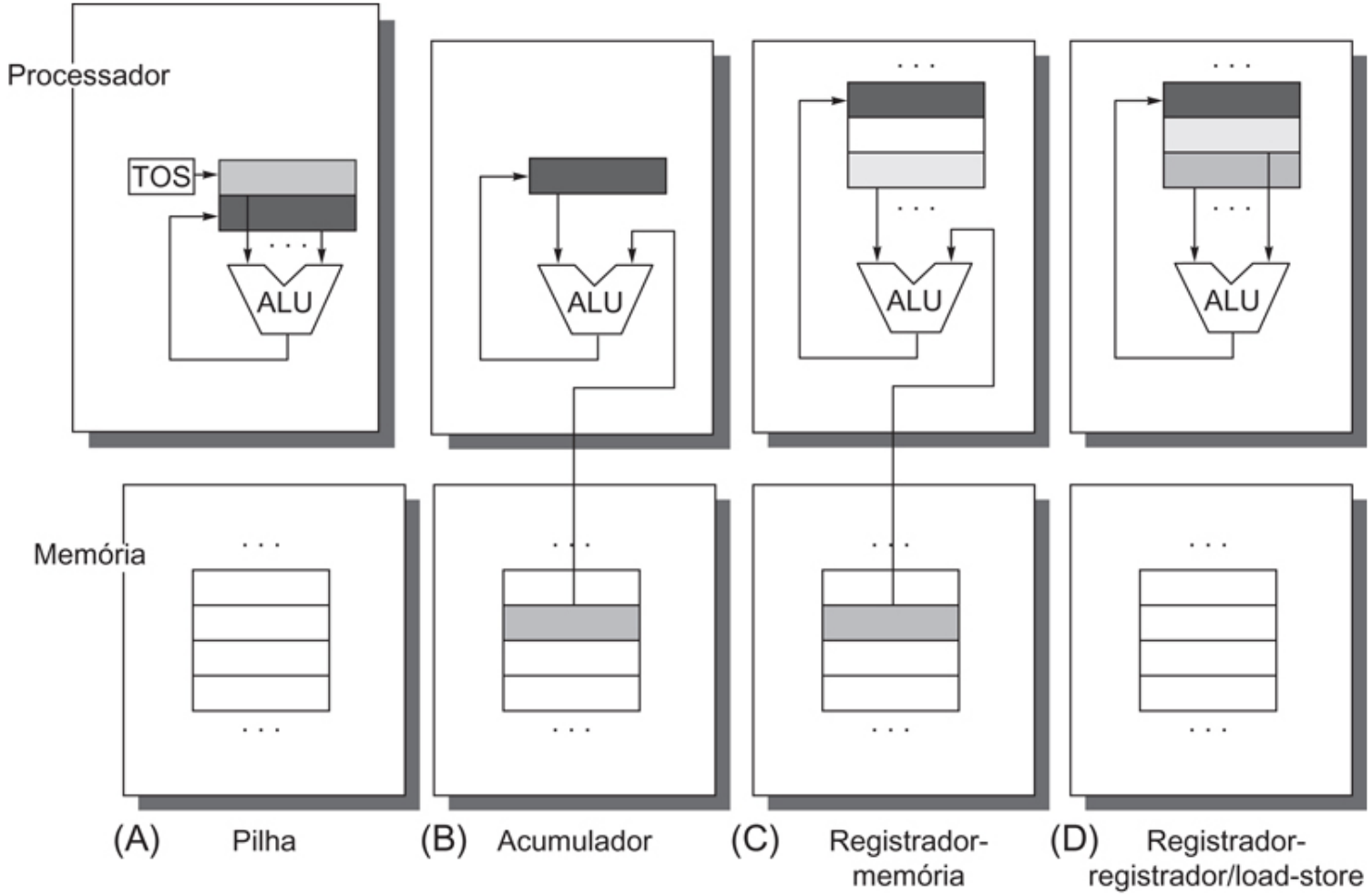
Arquiteturas

- A mudança para os processadores múltiplos levou ao termo core usado também para processadores.
- Em vez de se dizer microprocessador multiprocessador, o termo multicore (significando múltiplos núcleos) foi adotado.
- Dado que quase todos os chips têm múltiplos processadores, o nome unidade central de processamento, ou CPU, está perdendo popularidade.
- Hardware: refere-se aos detalhes específicos de um computador, incluindo o projeto lógico detalhado e a tecnologia de encapsulamento.
- Normalmente, uma linha de computadores contém máquinas com arquiteturas de conjunto de instruções idênticas e organizações quase idênticas, diferindo na implementação detalhada do hardware.

Armazenamento Interno

- As principais escolhas são: uma pilha, um acumulador ou um conjunto de registradores.
- Os operandos podem ser nomeados explícita ou implicitamente: os operandos em uma arquitetura de pilha estão implicitamente no topo da pilha e, em uma arquitetura de acumulador, um operando é implicitamente o acumulador.
- As arquiteturas de registradores de propósito geral possuem apenas operandos explícitos - registradores ou locais de memória.

Armazenamento Interno



Armazenamento Interno

- As setas indicam se o operando é uma entrada ou o resultado da operação da unidade lógica e aritmética (ALU) ou se é ao mesmo tempo uma entrada e o resultado.
- Os sombreados mais claros indicam entradas, e o sombreado mais escuro indica o resultado.
- Em (A), um registrador de topo da pilha (TOS) aponta para o operando de entrada superior, que é combinado com o operando abaixo.
- O primeiro operando é removido da pilha, o resultado assume o lugar do segundo operando, e o TOS é atualizado para apontar para o resultado.
- Todos os operandos estão implícitos.

Armazenamento Interno

- Em (B), o acumulador é tanto um operando de entrada implícito quanto um resultado.
- Em (C), um operando de entrada é um registrador, um está em memória e o resultado vai para um registrador.
- Todos os operandos são registradores em (D) e, como a arquitetura de pilha, só pode ser transferido para a memória através de instruções separadas: push ou pop para (A) e load ou store para (D).

Arquitetura

- A sequência de código para C = A + B para quatro classes de conjuntos de instruções.
- Observe que a instrução Add possui operandos implícitos para arquiteturas de pilha e de acumulador, e operandos explícitos para arquiteturas de registrador.
- Considera-se que A, B e C pertençam à memória e que os valores de A e B não podem ser destruídos.

Pilha	Acumulador	Registrador (registrador-memória)	Registrador (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

Endereçamento de Memória

- Em muitos computadores, os acessos a objetos maiores que um byte devem estar alinhados.
- Um acesso a um objeto de tamanho s bytes no endereço de byte A é alinhado se A mod s = 0.

	Valor dos 3 bits de ordem inferior do endereço de byte							
Largura do objeto	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1 byte (byte)	Alinhado	Alinhado	Alinhado	Alinhado	Alinhado	Alinhado	Alinhado	Alinhado
2 bytes (meia palavra)	Alinhado		Alinhado		Alinhado		Alinhado	
2 bytes (meia palavra)		Desalinhado		Desalinhado		Desalinhado		Desalinhado
4 bytes (palavra)	Alinhado				Alinhado			
4 bytes (palavra)		Desalinhado				Desalinhado		
4 bytes (palavra)			Desalinhado				Desalinhado	
4 bytes (palavra)				Desalinhado				Desalinhado
8 bytes (palavra dupla)	Alinhado							
8 bytes (palavra dupla)		Desalinhado						
8 bytes (palavra dupla)			Desalinhado					
8 bytes (palavra dupla)				Desalinhado				
8 bytes (palavra dupla)					Desalinhado			
8 bytes (palavra dupla)						Desalinhado		
8 bytes (palavra dupla)							Desalinhado	
8 bytes (palavra dupla)								Desalinhado

Operações no Conjunto de Instruções

- Os operadores aceitos pela maioria das arquiteturas de conjunto de instruções podem ser categorizados conforme abaixo:

Tipo de operador	Exemplos
Aritmético e lógico	Operações aritméticas e lógicas de inteiros: adição, subtração e (and), ou (or), multiplicação, divisão
Transferência de dados	Loads-stores (instruções de movimentação em computadores com endereçamento de memória)
Controle	Desvio, salto, chamada e retorno de procedimento, traps
Sistema	Chamada de sistema operacional, instruções de gerenciamento de memória virtual
Ponto flutuante	Operações de ponto flutuante: adição, multiplicação, divisão, comparação
Decimal	Adição decimal, multiplicação decimal, conversão decimal para caracteres
String	Movimento de string, comparação de string, pesquisa de string
Gráficos	Operações de pixels e vértices, operações de compactação/ descompactação

Operações no Conjunto de Instruções

Classificação	Instrução do 80x86	Média de inteiro (% total executada)
1	Load	22%
2	Desvio condicional	20%
3	Compare	16%
4	Store	12%
5	Add	8%
6	And	6%
7	Sub	5%
8	Move registrador-registrador	4%
9	Call	1%
10	Return	1%
Total		**96%**

Instruções para Fluxo de Controle

- Como as medidas de comportamento de desvios e saltos são completamente independentes das outras medidas e aplicações, examinamos agora o uso das instruções de fluxo de controle, que têm pouco em comum com as operações das seções anteriores.
- Não há uma terminologia consistente para instruções que mudam o fluxo de controle.
- Na década de 1950, elas normalmente eram chamadas transferências.
- A partir de 1960, o nome desvio começou a ser usado.
- Usaremos salto quando a mudança no controle for incondicional e desvio quando a mudança for condicional.

Endereçamento para Controle

- O endereço de destino de uma instrução de fluxo de controle sempre deve ser especificado.
- Ele é especificado explicitamente na instrução, na grande maioria dos casos - sendo o retorno de procedimento a principal exceção, já que, para o retorno, o destino não é conhecido em tempo de compilação.
- O modo mais comum de especificar o destino é fornecer um deslocamento que seja acrescentado ao contador de programa (PC).
- As instruções de fluxo de controle desse tipo são chamadas relativas ao PC.

Codificação das Instruções

- As escolhas mencionadas anteriormente afetarão o modo como as instruções são codificadas em uma representação binária para execução pelo processador.
- Essa representação afeta não só o tamanho do programa compilado, mas também a implementação do processador, que precisa decodificá-la para encontrar rapidamente a operação e seus operandos.
- A operação normalmente é especificada em um campo chamado opcode (código de operação).
- Essa é uma decisão importante: como codificar os modos de endereçamento com as operações.

Codificação das Instruções

- Essa decisão depende da faixa dos modos de endereçamento e do grau de independência entre opcodes e modos.
- Alguns computadores mais antigos possuem de um a cinco operandos com 10 modos de endereçamento para cada operando.
- Para um número tão grande de combinações, normalmente um especificador de endereço separado é necessário para cada operando: o especificador de endereço informa que modo de endereçamento é usado para acessar o operando.
- No outro extremo estão computadores load-store com um único operando de memória e apenas um ou dois modos de endereçamento; obviamente, nesse caso, o modo de endereçamento pode ser codificado como parte do opcode.

Codificação das Instruções

- Ao codificar as instruções, o número de registradores e o número de modos de endereçamento têm um impacto significativo no tamanho das instruções, já que o campo de registrador e o campo de modo de endereçamento podem aparecer muitas vezes em uma única instrução.
- Para a maioria das instruções, muito mais bits são consumidos nos campos de modo de endereçamento e de registrador do que na especificação do opcode.
- O arquiteto precisa equilibrar várias forças concorrentes ao codificar o conjunto de instruções.

Codificação das Instruções

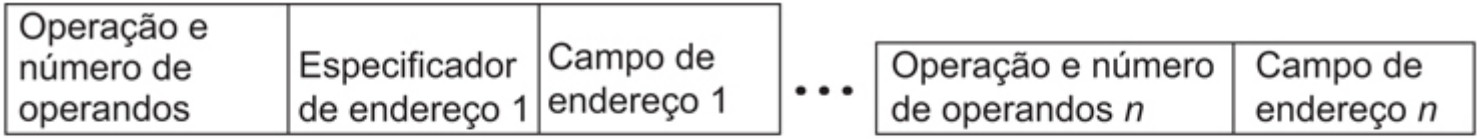
1. O desejo de ter o máximo de registradores e modos de endereçamento possíveis.
2. O impacto do tamanho dos campos de registrador e do modo de endereçamento no tamanho médio da instrução e, consequentemente, no tamanho médio do programa.
3. O desejo de ter instruções codificadas em tamanhos que sejam fáceis de controlar em uma implementação com pipeline.
   - Como um mínimo, o arquiteto deseja que as instruções sejam em múltiplos de bytes, e não em um comprimento em bits arbitrário.
   - Muitos arquitetos de computadores desktop e de servidores escolhem usar uma instrução de tamanho fixo para ganhar vantagens de implementação enquanto sacrificam o tamanho médio do código.

Codificação das Instruções

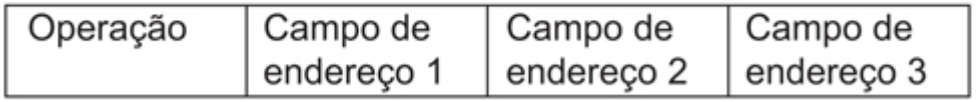
- A Figura a seguir mostra três escolhas comuns para codificar o conjunto de instruções.
- À primeira chamamos variável, já que permite que quase todos os modos de endereçamento estejam com todas as operações.
- Esse estilo é melhor quando há muitos modos de endereçamento e operações.
- À segunda opção, chamamos fixa, uma vez que combina a operação e o modo de endereçamento no opcode.
- Frequentemente, a codificação fixa terá um único tamanho para todas as instruções; ela funciona melhor quando há poucos modos de endereçamento e operações.

Codificação das Instruções

- O compromisso entre codificação variável e codificação fixa é o do tamanho dos programas contra a facilidade de decodificação no processador.
- A variável tenta usar o mínimo possível de bits para representar o programa, mas as instruções individuais podem variar bastante em tamanho e na quantidade de trabalho a ser executado.



(A) Variável (por exemplo, Intel 80x86, VAX)



(B) Fixo (por exemplo, RISC-V, ARM, MIPS, PowerPC, SPARC)

Codificação das Instruções

- O formato variável pode aceitar qualquer número de operandos, com cada especificador de endereço determinando o modo de endereçamento e o tamanho do especificador para esse operando.
- Ele geralmente permite a menor representação de código, já que não é preciso incluir os campos não usados.
- O formato fixo sempre tem o mesmo número de operandos, com os modos de endereçamento (se houver opções) especificados como parte do opcode.
- Normalmente, ele resulta no tamanho de código maior.

Operações do HW MIPS

- Todas as instruções aritméticas têm 3 operandos e a ordem dos operandos é fixa:

INSTRUÇÃO DESTINO, ORIGEM, ORIGEM

- Exemplo: Código C: A = B + C;
  - Código MIPS: add A, B, C # a recebe o conteúdo da soma dos conteúdos dos endereços B e C;
- Exemplo: Código C: A = B + C;

  D = A - E;
  - Código MIPS: add A, B, C

    sub D, A, E # D recebe o conteúdo da diferença entre os conteúdos dos endereços A e E, nessa ordem.
- Essa notação é rígida no sentido de que cada instrução aritmética do MIPS realiza apenas uma operação e sempre precisa ter exatamente três variáveis.

MIPS

- Para se somar quatro variáveis a arquitetura do MIPS necessita três linhas de código:
  - add a, b, c #
  - add a, a, d #
  - add a, a, e #

Operandos do MIPS

Nome	Exemplo	Comentários
32 registradores	`$s0-$s7`, `$t0-$t9`, `$zero`, `$a0-$a3`, `$v0-$v1`, `$gp`, `$fp`, `$sp`, `$ra`, `$at`	Localizações rápidas para dados. No MIPS, os dados precisam estar em regiões para realizar aritmética. O registrador `$zero` sempre é igual a 0 e o registrador `$at` é reservado pelo montador para lidar com constantes grandes.
2^{30} palavras de memória	Memória[0], Memória[4], ..., Memória[4294967292]	Acessada apenas pelas instruções de transferência de dados. O MIPS utiliza endereços de byte, de modo que os endereços sequenciais de dados diferem em 4. A memória mantém estruturas de dados, matrizes e registros espalhados.

MIPS

Linguagem assembly do MIPS

Categoria	Instrução	Exemplo	Significado	Comentários
Aritmética	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Três operandos; dados nos registradores
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Três operandos; dados nos registradores
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Usada para somar constantes
Transferência de dados	load word	lw $s1,20($s2)	$s1 = Memória[$s2 + 20]	Dados da memória para o registrador
	store word	sw $s1,20($s2)	Memória[$s2 + 20] = $s1	Dados do registrador para a memória
	load half	lh $s1,20($s2)	$s1 = Memória[$s2 + 20]	Halfword da memória para registrador
	load half unsigned	lhu $s1,20($s2)	$s1 = Memória[$s2 + 20]	Halfword da memória para registrador
	store half	sh $s1,20($s2)	Memória[$s2 + 20] = $s1	Halfword de um registrador para memória
	load byte	lb $s1,20($s2)	$s1 = Memória[$s2 + 20]	Byte da memória para registrador
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memória[$s2 + 20]	Byte da memória para registrador
	store byte	sb $s1,20($s2)	Memória[$s2 + 20] = $s1	Byte de um registrador para memória
	load linked word	ll $s1,20($s2)	$s1 = Memória[$s2 + 20]	Carrega word como 1ª metade do swap atômico
	store condition, word	sc $s1,20($s2)	Memória[$s2+20]=$s1;$s1=0 or 1	Armazena word como 2ª metade do swap atômico
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Carrega constante nos 16 bits mais altos
Lógica	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Três operadores em registrador; AND bit a bit
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Três operadores em registrador; OR bit a bit
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Três operadores em registrador; NOR bit a bit
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	AND bit a bit registrador com constante
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	OR bit a bit registrador com constante
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Deslocamento à esquerda por constante
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Deslocamento à direita por constante
Desvio condicional	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Testa igualdade; desvio relativo ao PC
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Testa desigualdade; relativo ao PC
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compara menor que; usado com beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compara menor que sem sinal
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compara menor que constante
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compara menor que constante sem sinal
Desvio incondicional	jump	j 2500	go to 10000	Desvia para endereço de destino
	jump register	jr $ra	go to $ra	Para switch e retorno de procedimento
	jump and link	jal 2500	$ra = PC + 4; go to 10000	Para chamada de procedimento

MIPS

- **Tipos de Dados:**

  Todas as instruções são de 32 bits

  - Byte = 8 bits
  - Halfword = 2 bytes
  - Word = 4 bytes
  - Um caractere ocupa 1 byte na memória
  - Um inteiro ocupa 1 word(4 bytes) na memória

- **Formatações:**

  - Números são representados normalmente. Ex: 4
  - Caracteres ficam entre aspas simples. Ex: ‘a’
  - Strings ficam entre aspas duplas. Ex: “palavra”

MIPS

- **Registradores**
  - 32 registradores
  - Os registradores são precedidos de $ nas instruções
  - Número do registrador: $0 até $31
  - A pilha começa da parte alta da memória e cresce em direção a parte baixa da memória.
- **Estrutura do Programa**
  - A declaração de dados deve vir anterior ao código do programa.
- **Declaração de Dados:**
  - Os nomes declarados das variáveis são usados no programa. Dados guardados na memória principal (RAM).

MIPS

# do Reg.	Nome	Descrição
0	$zero	retorna o valor 0
1	$at	(assembler temporary) reservado pelo assembler
2~3	$v0-$v1	(values) das expressões de avaliação e resultados de função
4~7	$a0-$a3	(arguments) Primeiros quatro parametros para subrotinas. Não é preservado em chamadas de procedures.
8~15	$t0-$t7	(temporaries) Subrotinas pode usar salvando-os ou não. Não é preservado em chamadas de procedures.
16~23	$s0-$s7	(saved values) Uma subrotina que usa um desses deve salvar o valor original e restaurar antes de terminar. Preservados na chamada de procedures.
24~25	$t8-$t9	(temporaries) Usados em adição aos $t0-$t7
26~27	$k0-$k1	Reservados para uso do tratamento de interrupção/trap.
28	$gp	(global pointer) Aponta para o meio do block de 64k de memoria no segmento de dados estaticos.
29	$sp	(stack pointer) Aponta para o top da pilha
30	$s8/$fp	(saved values/ frame pointer) Preservado na chamada de procedures.
31	$ra	(return address)
	$f0	Recebe o retorno de floats em funções
	$f12/$f14	Usados para passar floats para funções
	($f12,$f13) ($f14,$f15)	Usados em conjunto para passar doubles para funções

MIPS

- Leitura/Escrita
  - Acesso à memória RAM apenas com instruções de leitura e escrita.
  - Todas outras instruções usam registradores como operando.
- As instruções aritméticas usam sempre três operandos: destino, origem, origem.

MIPS

- Princípio de projeto 1: simplicidade favorece a regularidade. Como?
- Cada instrução aritmética executa apenas uma operação.
- Código C: A = B + C + D;

  E = F - A;

  Código MIPS: add t0, B, C

  add A, t0, D

  sub E, F, A

  Código C: F = (G + H) - (I + J);
- Código MIPS: add t0, G, H

  add t1, I, J

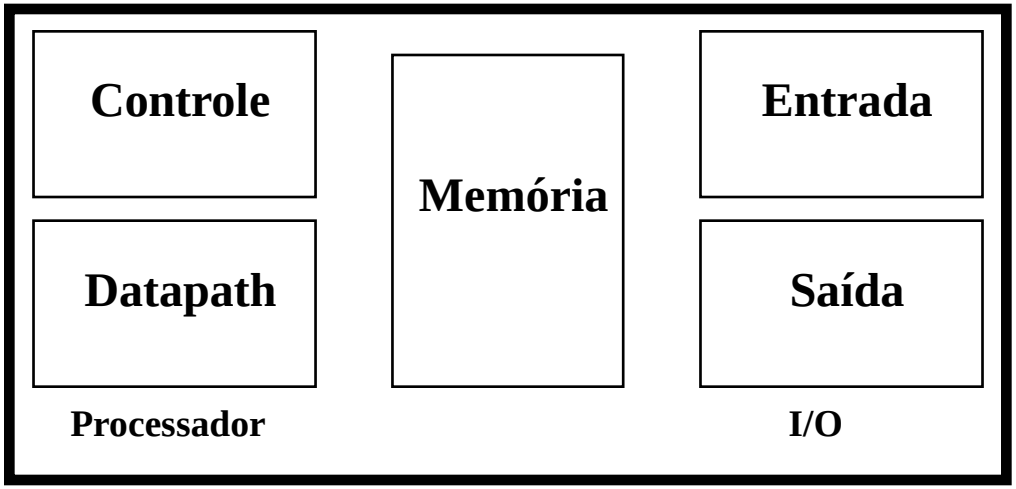
  sub F, t0, t1

O compilador precisa quebrar esta linha de código, porque cada instrução executa apenas uma operação

MIPS

- Porém as instruções da linguagem do MIPS não conseguem trabalhar com variáveis A, B, C…, como está nos exemplos anteriores.
- Os operandos das instruções vêm da MEMÓRIA INTERNA do processador.

→ REGISTRADORES.



MIPS

- MIPS possui registradores de 32 bits

BITS →	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reg. 0 →	0	1	0	0	1	0	1	1	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	0	1	1	0	0	0	1	1	0
Reg. 1 →	1	1	1	0	1	1	0	0	0	0	0	0	1	0		1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1
Reg. 2 →	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
Reg. 3 →	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Reg. 30 →	0	0	0	0	1	1	1	0	1	1	1	1	0	0	1	1	0	1	0	1	0	0	0	1	1	1	1	1	1	1	1	1
Reg. 31 →	0	0	1	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0	1	0	1	1	0	1	0	1	1	0	0	0	1	1

- Os operandos das instruções aritméticas precisam ser escolhidos entre os registradores disponíveis.
- Os registradores são referenciados usando o caracter ‘$’.
- A associação entre as variáveis do seu programa e os registradores do processador é feita pelo compilador.

MIPS

- Princípio de Projeto 2: Menor significa mais rápido.
- Uma quantidade muito grande de registradores pode aumentar o tempo do ciclo do clock simplesmente porque os sinais eletrônicos levam mais tempo quando precisam atravessar uma distância maior.
- Orientações como “menor significa mais rápido” não são absolutas; 31 registradores podem não ser mais rápidos do que 32.
- O projetista precisa equilibrar o desejo dos programas por mais registradores, com o desejo do projetista de manter o ciclo de clock rápido.
- Outro motivo para não usar mais de 32 é o número de bits que seria necessário no formato da instrução.

MIPS

- Código C: F = (G + H) - (I + J)
- Supondo que as variáveis estão armazenadas da seguinte forma:
  - F → $s0
  - G → $s1
  - H → $s2
  - I → $s3
  - J → $s4
- Código MIPS:

```
add $t0, $s1, $s2      → reg. t0 recebe G + H
add $t1, $s3, $s4      → reg. t1 recebe I + J
sub $s0, $t0, $t1      → F recebe (G + H) - (I + J)
```

MIPS

- Instruções tipo R (aritméticas):

Sintaxe: operação registrador de destino, operando 1, operando 2

- Formato da Instrução:

opcode	rs	rt	rd	shamt	funct
6 bit	5 bits	5 bits	5 bits	5 bits	6 bits

- Nas instruções aritméticas, o opcode é sempre 0 (000000). A sub-operação estará no campo funct (Tabela).

MIPS

- Tabela de Functs/Opcodes

Funct/Opcode	DECIMAL	BINÁRIO
ADD	32	100 000
SUB	34	100 010
OR	37	100 101
AND	36	100 100
SLT	42	101 010
BNE	4	000 100
BEQ	5	000 101
J	2	000 010
JR	8	001 000
LW	35	100 011
SW	43	101 011

Opcode = 0

MIPS

- Tabela Endereços Registradores

Nome do Registrador	Endereço	Binário	Uso
$zero	0	000 000	constante zero
$at	1	000 001	reservado para o montador
$v0	2	000 010	avaliação de expressão e resultados de uma função
$v1	3	000 011	avaliação de expressão e resultados de uma função
$a0	4	000 100	argumento 1 (passam argumentos para as rotinas)
$a1	5	000 101	argumento 2
$a2	6	000 110	argumento 3
$a3	7	000 111	argumento 4

MIPS

- Tabela Endereços Registradores

Nome do Registrador	Endereço	Binário	Uso
$t0	8	001 000	$t0
$t1	9	001 001	$t1
$t2	10	001 010	$t2
$t3	11	001 011	$t3
$t4	12	001 100	$t4
$t5	13	001 101	$t5
$t6	14	001 110	$t6
$t7	15	001 111	$t7

MIPS

- Tabela Endereços Registradores

Nome do Registrador	Endereço	Binário	Uso
$s0	16	010 000	temporário salvo (valores de longa duração e devem ser preservados entre chamadas)
$s1	17	010 001	temporário salvo
$s2	18	010 010	temporário salvo
$s3	19	010 011	temporário salvo
$s4	20	010 100	temporário salvo
$s5	21	010 101	temporário salvo
$s6	22	010 110	temporário salvo
$s7	23	010 111	temporário salvo

MIPS

- Tabela Endereços Registradores

Nome do Registrador	Endereço	Binário	Uso
$t8	24	011 000	temporário
$t9	25	011 001	temporário
$k0	26	011 010	reservado para o Kernel do sistema operacional
$k1	27	011 011	reservado para o Kernel do sistema operacional
$gp	28	011 100	ponteiro para área global
$sp	29	011 101	stack pointer (aponta para o último local da pilha)
$fp	30	011 110	frame pointer (aponta para a primeira palavra do frame de pilha do procedimento)
$ra	31	011 111	endereço de retorno de uma chamada de procedimento

MIPS

- Assim:

add $t0, $s1, $s3

Sintaxe: operação registrador de destino, operando 1, operando 2

Formato da Instrução:

opcode	rs	rt	rd	shamt	funct
6 bit	5 bits	5 bits	5 bits	5 bits	6 bits
0	17	19	8	0	32
000000	10001	10011	01000	00000	100000

Registradores x Memória

- Como o processador só possui 32 registradores ⇒ armazenamento de dados extras na memória.
- Organização da memória externa no MIPS:
  - A memória é vista como um único e grande vetor (array).
  - Cada endereço deste vetor acessa uma posição com 8 bits ⇒ 1 BYTE.

0	**8 bits de dados**
1	**8 bits de dados**
2	**8 bits de dados**
3	**8 bits de dados**
4	**8 bits de dados**
5	**8 bits de dados**
6	**8 bits de dados**

Organização da Memória

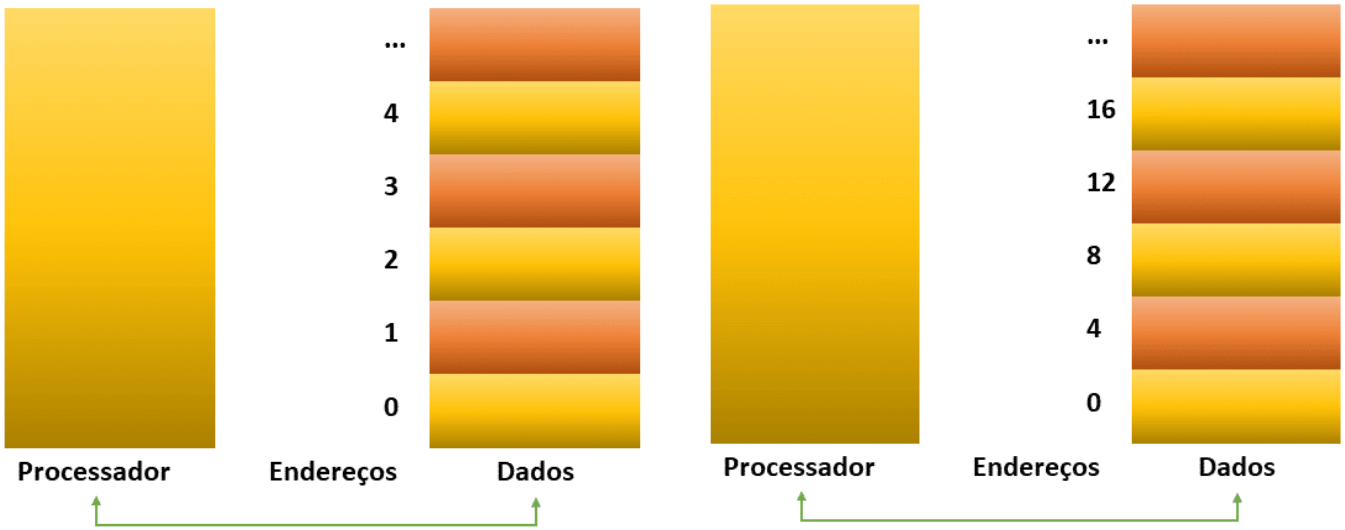
- Bytes são pequenos, portanto vários processadores usam "words" (palavra)
- Para o MIPS, uma palavra tem 32 bits ou 4 bytes.

0	8 bits de dados	8 bits de dados	8 bits de dados	8 bits de dados	→ 32 bits
4	8 bits de dados	8 bits de dados	8 bits de dados	8 bits de dados	→ 32 bits
8	8 bits de dados	8 bits de dados	8 bits de dados	8 bits de dados	→ 32 bits
12	8 bits de dados	8 bits de dados	8 bits de dados	8 bits de dados	→ 32 bits
...					

- Os BYTES da memória externa são endereçados através de qualquer um dos registradores internos do processador:
  - Os registradores internos possuem 32 bits ⇒
  - ⇒ 2^{32} bytes com endereço de byte de 0 to 2^{32}-1
  - ⇒ 2^{30} palavras com endereço de byte 0, 4, 8, ... 2^{32}- 4

Organização da Memória

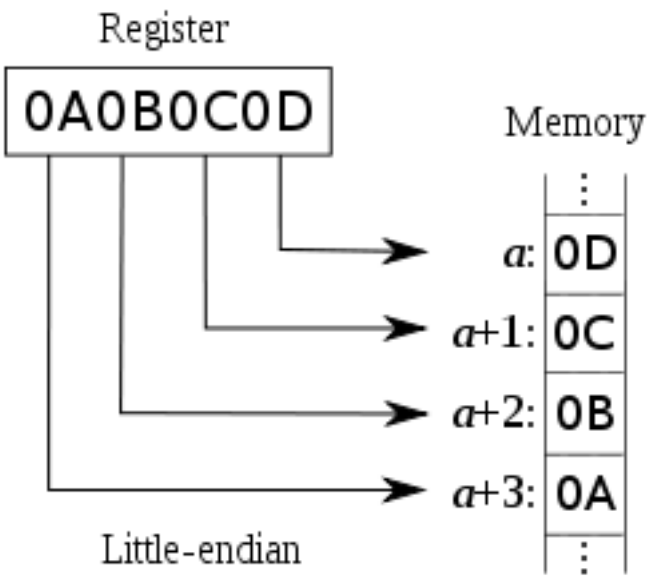
- O MIPS possui 2 instruções que permitem a transferência de dados de e para a memória (Instruções tipo I).
- Restrição de Alinhamento: as palavras precisam começar em endereços que sejam múltiplos de 4 (muitas arquiteturas exigem).



Endianness

- Little Endian
- Processadores Little-Endian numeram os bytes dentro de uma palavra (word) de forma que o byte com o valor mais baixo é o mais à direita.

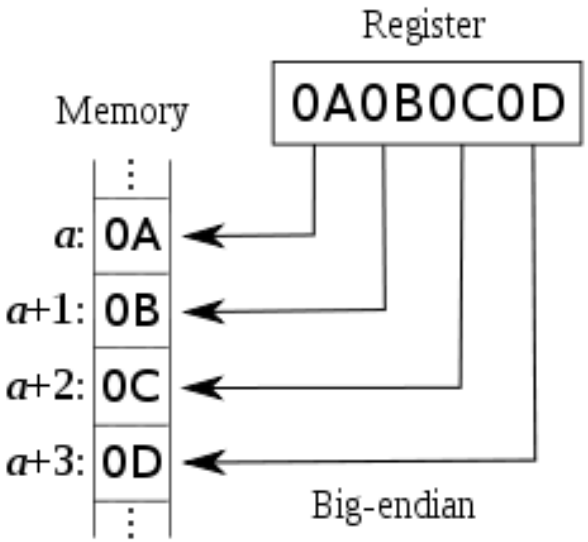
Byte #			
3	2	1	0



Endianness

- Big Endian
- Processadores Big-Endian numeram os bytes dentro de uma palavra (word) de forma que o byte com o valor mais baixo é o mais à esquerda.

Byte #			
0	1	2	3



Load e Store

- Código C: g = h + A[8];

  Supondo: g → $s1

  h → $s2

  endereço base do vetor A → $s3

- Código MIPS:

- lw → load word → carrega 4 bytes da memória para o registrador interno.

Sintaxe: lw reg. destino, deslocamento (endereço base).

- (8) é chamado de offset e o registrador acrescentado para formar o endereço ($s3) é chamado de registrador base.

opcode	base	rt	deslocamento
6 bits	5 bits	5 bits	16 bits
35 (lw)	19 ($s3)	8 ($t0)	32

Load e Store

- Código C: A[7] = B - D;

  Supondo: B → $s3

  D → $s4

  Endereço base do vetor A → $s0

- Código MIPS:

- sw → store word → armazena 4 bytes de um dos registradores na memória.

Sintaxe: sw reg. origem, deslocamento (base).

opcode	base	rt	deslocamento
6 bits	5 bits	5 bits	16 bits
43 (sw)	16 ($s0)	8 ($t0)	28

Load e Store

- Código C: G = H + B[3];

  Supondo: G → $s1

  H → $s2

  Endereço base de B → $s3

- Código C: A[12] = H - A[8];

  Supondo: H → $s5

  Endereço base de A → $s7

Load e Store

Juros[3] = X + Z;

Quantia[5] = W;

Total[10] = Quantia[5] + Juros[3];

Supondo	X →	$s0	End. Base Juros	→ $s4
	Z →	$s1	End. Base Quantia	→ $s5
	W →	$s2	End. Base Total	→ $s6

Constantes

- E quando precisarmos utilizar um valor diretamente, como nas instruções de alto nível abaixo:
  - a = b + 25;
- Os projetistas do MIPS, prevendo este uso, desenvolveram as instruções imediatas, que oferecem versões das instruções aritméticas e lógicas em que um dos operandos é um valor imediato.

Exemplos:

- addi rt, rs, imm # operação soma imediata, opcode 8
- ori rt, rs, imm # operação lógica OR imediata, opcode 13
- xori rt, rs, imm # operação lógica XOR imediata, opcode 14
- slti rt, rs, imm # instrução SLT imediata, opcode 10

Constantes

- Formato

Opcode	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

- Constantes ocorrem com frequência e, incluindo-as dentro das instruções aritméticas, as operações ficam muito mais rápidas e usam menos energia do que se as constantes fossem lidas da memória (lw)!
- A constante zero tem outro emprego, que é simplificar o conjunto de instruções por oferecer variações úteis.
- O MIPS dedica o registrador $zero para ter sempre o valor zero (ele é o registrador número 0).
- O uso da frequência para justificar as inclusões de constantes é outro exemplo da grande ideia de tornar o caso comum veloz.

addi

- add imediato ou addi.
- Para somar 4 ao registrador $s0, simplesmente escrevemos:
  - addi $t0, $s0, 4 # o endereço $t0 recebe o de $s0 + 4;

Opcode	rs	rt	imm
6 bits	5 bits	5 bits	16 bits
8	16	8	4
001000	10000	01000	0000000000000100

Números com e sem sinal

- Com sinal e sem sinal se aplica a loads e também à aritmética. A função de um load com sinal é copiar o sinal repetidamente para preencher o restante do registrador (chamado extensão do sinal), mas sua finalidade é colocar uma representação correta do número dentro desse registrador. Os loads sem sinal simplesmente preenchem com 0s à esquerda dos dados, pois o número representado pelo padrão de bits é sem sinal.
- Ao carregar uma word de 32 bits em um registrador de 32 bits, o ponto é discutível; loads com sinal e sem sinal são idênticos.
- O MIPS oferece dois tipos de loads de byte: load byte (lb) trata o byte como um número com sinal e, assim, estende por sinal para preencher os 24 bits mais à esquerda do registrador, enquanto load byte unsigned (lbu) trabalha com inteiros sem sinal.

Multiply

- MIPS oferece um par separado de registradores de 32 bits, a fim de conter o produto de 64 bits, chamados Hi e Lo.
- Para apanhar o produto de 32 bits inteiro, o programador usa move from lo (mflo).
- O montador MIPS gera uma pseudoinstrução para multiplicar, que especifica três registradores de uso geral, criando instruções mflo e mfhi que colocam o produto nos registradores.

Multiply

- Multiply → mul

  mul destino, origem, origem

- Exemplo: Código C: B = F * AUX;
- Supondo B → $s0

  F → $s1

  AUX → $s2

Código MIPS:

opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
011100				0	2

Multiply

- Purpose: Multiply Integers (with result to GPR)
- MUL: Multiply Words Signed, Low Word
- MUH: Multiply Words Signed, High Word
- MULU: Multiply Words Unsigned, Low Word
- MUHU: Multiply Words Unsigned, High Word
- Description:
- MUL: GPR[rd] $\leftarrow$ lo_word (multiply.signed (GPR[rs] $\times$ GPR[rt]));
- MUH: GPR[rd] $\leftarrow$ hi_word (multiply.signed (GPR[rs] $\times$ GPR[rt]));
- MULU: GPR[rd] $\leftarrow$ lo_word (multiply.unsigned (GPR[rs] $\times$ GPR[rt]));
- MUHU: GPR[rd] $\leftarrow$ hi_word (multiply.unsigned (GPR[rs] $\times$ GPR[rt]));

Multiply

- Multiply → mul (mult e multu)

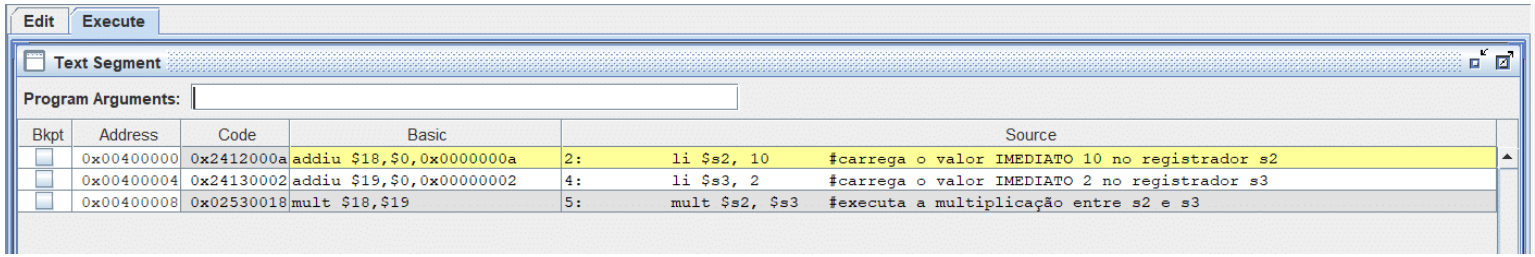
  mult rs, rt (removido na versão 6 do MIPS 32)

opcode	rs	rt	rd + shamt	funct
6 bits	5 bits	5 bits	10 bits	6 bits
000000				011000 (mult) 011001 (multu)

Multiply

- O resultado da multiplicação é armazenada no registrador **lo**, que é o registrador que armazena a palavra menos significativa do produto.
- A palavra mais significativa do produto é armazenada no registrador **hi**.

```
.text
        li $s2, 10          # carrega o valor imediato 10 no registrador $s2
        li $s3, 2           # carrega o valor imediato2 no registrador $s3
        mult $s2, $s3       # executa a multiplicação entre $s2 e $s3
```



Multiply

- Ex: Código C: X1 = X2 * 12;

  Código MIPS: addi $t0, $zero, 12 # $t0 = 0 + 12

  carregamos o valor 12 em $t0

Registrador $zero contém sempre o valor 0 armazenado.

Multiply

- Ex.: Código C: A = B + 4;

  C = A * 100;

  D = C - 9;

Vetores Indexados

- Durante a programação, frequentemente utilizamos a indexação de posições de vetores através de variáveis.
- Ex.: Código C: G = H + A[i];
- A[i] deve ser carregado em um registrador interno antes de realizarmos a operação:
  - Supondo G $\rightarrow$ $s1

    H $\rightarrow$ $s2

    i $\rightarrow$ $s4

    End. Base de A $\rightarrow$ $s3

Vetores Indexados

- Código MIPS:

Vetores Indexados

- Determine o código de montagem correspondente ao código C abaixo

```
swap(int v[], int k);
{
        int temp;
        temp = v[k]
        v[k] = v[k+1];
        v[k+1] = temp;
}
```

Vetores Indexados

- Determine o código de montagem correspondente ao código C abaixo

```
swap(int v[], int k);
{
        int temp;
        temp = v[k]
        v[k] = v[k+1];
        v[k+1] = temp;
}
```

```
swap:
        addi $t0, $zero, 4
        mul $t2, $s5, $t0
        add $t2, $s4, $t2
        lw $t5, 0($t2)
        lw $t6, 4($t2)
        sw $t6, 0($t2)
        sw $t5, 4($t2)
        jr $ra
```

Derramamento de Registradores

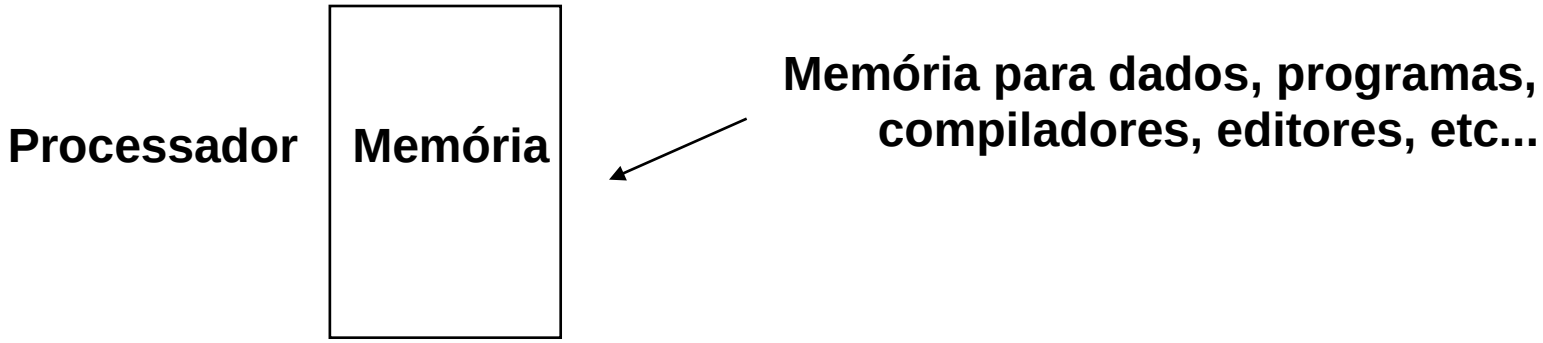
- Como visto, o processador possui um pequeno número de registradores para realizar as operações aritméticas.
- Em programas com grande quantidade de variáveis, o armazenamento das que sobram é feito na memória da máquina (lw e sw).
- O compilador mantém nos registradores da CPU somente as varíaveis que são muito utilizadas.
- A linguagem C permite que o programador especifique que uma variável seja sempre mantida no registrador, através da especificação de tipo register.
- O processo de armazenamento das variáveis menos utilizadas é chamado de DERRAMAMENTO DE REGISTRADORES.

Linguagem de Máquina

- Princípio de Projeto 3: Um bom projeto exige bons compromissos.
- O compromisso escolhido pelos projetistas do MIPS é manter todas as instruções com o mesmo tamanho, exigindo, assim, diferentes tipos de formatos para diferentes tipos de instruções.

Programa Armazenado

- As instruções são bits → tratadas da mesma forma que os dados.
- Esta estratégia possibilitou que programas sejam armazenados em memória: lidos ou escritos da mesma forma que os dados



Programa Armazenado

- Fetch (Ciclo de Busca) & Execute Cycle (Ciclo de Execução)
- Instruções são buscadas e colocadas em um registrador especial (PC - Program Counter).
- Registrador de controle indica as ações subsequentes.
- O programa busca a próxima instrução e continua a execução.
- As instruções contêm em si a operação a ser feita e os operandos sobre os quais atuar.

Operações Lógicas

- A primeira classe dessas operações é chamada de shifts (deslocamentos). Elas movem todos os bits de uma word para a esquerda ou direita, preenchendo os bits que ficaram vazios com 0s.
- Suponha que um endereço contenha o número
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101_{(2)} = 5_{(10)}$

e que fosse executada uma instrução para deslocar 4 bits à esquerda. O novo valor seria

  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101\ 0000_{(2)} = 80_{(10)}$
- O dual de um shift à esquerda é um shift à direita.
- Os nomes reais das duas instruções shift no MIPS são shift left logical (sll) e shift right logical (srl).

Operações Lógicas

- Essas instruções são do tipo R.
- A codificação de sll é 0 nos campos op e funct, rd contém 10 (registrador $t2), rt contém $s0 e shamt contém o deslocamento requerido. O campo rs não é utilizado e, por isso, é definido como 0.
- O deslocamento lógico à esquerda oferece um benefício adicional. O deslocamento à esquerda de i bits gera o mesmo resultado que multiplicar por 2^i.
- A instrução sll anterior desloca 4 posições à esquerda, o que gera o mesmo resultado que multiplicar por 2^4 ou 16.

Operações Lógicas

- sll $t2, $s0, 4 # $t2 = $s0 << 4 bits
- A representação fica assim:

opcode	rs	rt	rd	shamt	funct
0	0	16	10	4	0
6 bit	5 bits	5 bits	5 bits	5 bits	6 bits

- Em linguagem de Máquina:

Operações Lógicas

- srl $t2, $s0, 7 # $t2 = $s0 << 7 bits
- A representação fica assim:

opcode	rs	rt	rd	shamt	funct
0	0	16	10	7	2
6 bit	5 bits	5 bits	5 bits	5 bits	6 bits

- Em linguagem de Máquina:

Operações Lógicas

- Operação AND (E) bit a bit

$t2 0000 0000 0111 0000 0000 1101 0000 0000

$t1 0000 0000 0000 0000 0011 1100 0000 0000

and $t0, $t1, $t2 - $t0 0000 0000 0000 0000 0000 1100 0000 0000

opcode	rs	rt	rd	shamt	funct
0	9	10	8	0	37

- A operação AND pode ser utilizada para verificar o valor de bits individuais em determinado registrador.
- Em linguagem de Máquina:

Operações Lógicas

- Operação OR (OU) bit a bit

$t2 0000 0000 0111 0000 0000 1101 0000 0000

$t1 0000 0000 0000 0000 0011 1100 0000 0000

or $t0, $t1, $t2 - $t0 0000 0000 0111 0000 0011 1101 0000 0000

opcode	rs	rt	rd	shamt	funct
0	9	10	8	0	36

- A operação OR pode ser utilizada para colocar 1 em determinados bits.
- Em linguagem de Máquina:

Desvio Condicional

- Qual a diferença entre um computador e uma calculadora?
- Decisões → dependem das instruções anteriores:
  - possibilidade de alterar o fluxo de controle ou
  - continuar com a próxima instrução a ser executada.
- O MIPS possui duas instruções de desvio condicional "branch":

beq $t0, $t1, Label → branch on equal

bne $t0, $t1, Label → branch on not equal

Ex.: Código C:

```
if (i==j)
    h = i + j;
```

Código MIPS:

```
bne $s0, $s1, Label
add $s3, $s0, $s1
Label:  ...
```

Desvio Condicional

- Código em C: if (i == j) go to L1;

  f = g + h;

  L1: f = f - i;

- Supondo f → $s0

  i → $s3

  g → $s1

  j → $s4

  h → $s2

Código MIPS:

O montador controla o endereço dos labels utilizados, sendo desnecessário ao compilador manter o controle do mesmo

Desvio Condicional

- BEQ (Branch On Equal) - Desvie se for igual.
- Essa instrução força um desvio para o comando com o LABEL (nome de desvio) se o valor no registrador 1 for igual ao valor no registrador 2, portanto, é uma instrução de comparação.
- Sintaxe: beq registrador1, registrador2, endereço de desvio
- Formato: Tipo I

opcode	rs	rt	endereço
6 bits	5 bits	5 bits	16 bits

Desvio Condicional

- MIPS: beq $s3, $s4, L1 # Desvia para L1 se $s3 = $s4, se não, linha seguinte

  add $s0, $s1, $s2

  L1: sub $s0, $s0, $s3

opcode	rs	rt	endereço
6 bits	5 bits	5 bits	16 bits
4	19	20	L1 (10008)
00100	10011	10100	0010 0111 0001 1000

Desvio Condicional

- MIPS: bne $s3, $s4, L1 # Desvia para L1 se $s3 ≠ $s4, se não, linha seguinte

```
add  $s0, $s1, $s2
L1: sub  $s0, $s0, $s3
```

opcode	rs	rt	endereço
6 bits	5 bits	5 bits	16 bits
5	19	20	L1 (10008)
00101	10011	10100	0010 0111 0001 1000

Desvio Incondicional

- O MIPS possui 1 instrução de desvio incondicional:

  j label → jump

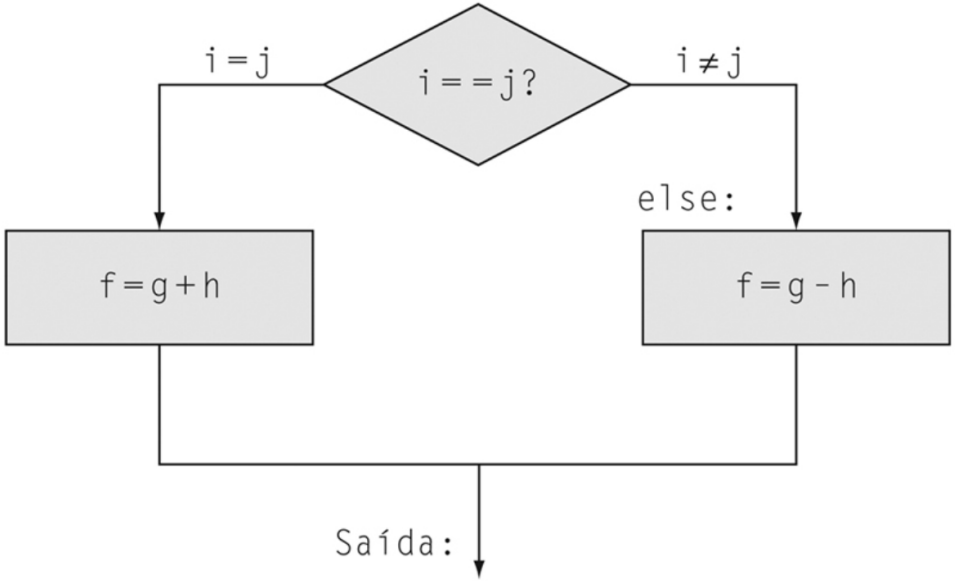
- Instrução tipo J

OpCode	Endereço
6 bits	26 bits
2	Endereço

Desvio Incondicional

- Exemplo: comando if-then-else em linguagem de montagem

```
if (i=j)
f = g+h;
else
f=g-h;
```



Sejam: f = $s0, g = $s1, h = $s2, i = $s3 e j = $s4

Desvio Incondicional

```
beq $s4, $s5, Then      # Se i = j vá para Then
sub $s3, $s4, $s5       # Só roda se i ≠ j - o que equivale ao Else
j Exit
Then:   add $s3, $s4, $s5 # f = g + h
        Exit:...
```

Desvio Incondicional

- De modo geral, o código será mais eficiente se testarmos a condição oposta ao desvio no lugar do código que realiza a parte then subsequente do if (o rótulo Else é definido a seguir) e assim usamos o desvio se os registradores forem a instrução de não igual (bne):
-
```
bne $s4, $s5, Else
add $s3, $s4, $s5
j Exit
        Else: sub $s3, $s4, $s5
        Exit:...
```
- A primeira expressão compara a igualdade, de modo que poderíamos querer desviar se os registradores forem a instrução de igual (beq).

Loops

Código C:
```
while (save[i] == k)
    i = i + j;
```

- O primeiro passo é carregar save[i] em um registrador temporário.
- Antes que possamos carregar save[i] em um registrador temporário, precisamos ter seu endereço.
- Antes que possamos somar i à base do array save para formar o endereço, temos de multiplicar o índice i por 4, em razão do problema do endereçamento em bytes.
- Podemos usar o deslocamento lógico à esquerda, pois o deslocamento em 2 bits à esquerda multiplica por 2^2 ou 4.
- Precisamos acrescentar o rótulo Loop para podermos desviar de volta a essa instrução no final do loop:

Loops

Código C: while (save[i] == k)

i = i + j;

Supondo: i → $s3 k → $s5

j → $s4 end. de save → $s6

Código MIPS: Loop:

```
addi  $t0, $zero, 4      # Coloca o valor 4 em $t0
mul   $s3, $s3, $t0      # Multiplica 4 pelo índice i (deslocamento)
add   $t1, $s3, $s6      # Posição do elemento a ser utilizado
lw    $t0, 0($t1)        # Carregou o valor em $t0
bne   $t0, $s5, Exit     # Compara o valor do elemento encontrado com k
add   $s3, $s3, $s4      # Incrementa i com o valor j
j     Loop               # Volta para o loop
Exit: ...                #
```

Esse procedimento está errado!

Loops

Código C: while (save[i] == k)

i = i + j;

Supondo: i → $s3 k → $s5

j → $s4 end. de save → $s6

Código MIPS:

```
Loop:   addi  $t0, $zero, 4     # Coloca o valor 4 em $t0
        mul   $t1, $s3, $t0     # Multiplica 4 pelo índice i (deslocamento)
        add   $t2, $t1, $s6     # Posição do elemento a ser utilizado
        lw    $t0, 0($t2)       # Carregou o valor em $t0
        bne   $t0, $s5, Exit    # Compara o valor do elemento encontrado com k
        add   $s3, $s3, $s4     # Incrementa i com o valor j
        j     Loop              # Volta para o loop
Exit:   ...                     #
```

Set on Less Than

- Nós Temos: beq $\rightarrow$ branch on equal

  bne $\rightarrow$ branch on not equal
- Como produzir Branch-if-less-than?
- Instrução “set on less than”:

```
slt $t0, $s1, $s2        # $t0 = 1 se $s1 < $s2
                         # $t0 = 0 se $s1 ≥ $s2
```

Set on Less Than

- Ex.: if ($s1 < $s2)

```
                    slt $t1, $s1, $s2
        $t0 = 1;    bne $t1, $zero, Else
else                addi $t0, $zero, 1
        $t0 = 0;    j    Sai
Else:               addi $t0, $zero, 0
Sai:
```

Os compiladores MIPS utilizam as instruções slt, slti, beq, bne e o valor fixo 0 (sempre à disposição com a leitura do registrador $zero) para criar todas as condições relativas: igual, diferente, menor que, menor ou igual, maior que, maior ou igual.

Set on Less Than

- Suponha que o registrador $s0 tenha o número binário

11111111111111111111111111111111

e que o registrador $s1 tenha o número binário

00000000000000000000000000000000

- Quais são os valores dos registradores $t0 e $t1 após essas duas instruções?

  Qual o binário correspondente a cada instrução?

slt $t0, $s1, $s0 e sltu $t1, $s1, $s0?

Set on Less Than

- Tratar números com sinal como se fossem sem sinal é um modo de baixo custo para verificar se $0 \leq x < y$, que corresponde à verificação de índice fora de limite dos arrays.
- O principal é que os inteiros negativos na notação de complemento de dois se parecem com números grandes na notação sem sinal; ou seja, o bit mais significativo é um bit de sinal na primeira notação, mas uma grande parte do número na segunda.
- Assim, uma comparação sem sinal de $x < y$ também verifica se x é negativo, bem como se x é menor que y.

Instrução Case/Switch

- A maioria das linguagens de programação possui uma instrução case ou switch, para o programador poder selecionar uma dentre muitas alternativas, dependendo de um único valor.
- O modo mais simples de implementar switch é por meio de uma sequência de testes condicionais, transformando a instrução switch em uma cadeia de instruções if-then-else.
- Às vezes, as alternativas podem ser codificadas de forma mais eficiente como uma tabela de endereços de sequências de instruções alternativas, chamada tabela de endereços de desvio ou tabela de desvio, e o programa só precisa indexar na tabela e depois desviar para a sequência apropriada.

Instrução Case/Switch

- A tabela de desvios é, então, apenas um array de palavras com endereços que correspondem aos rótulos no código.
- O programa carrega a entrada apropriada a partir da tabela de desvio para um registrador.
- Depois, ele precisa desviar usando o endereço no registrador.
- Para apoiar tais situações, computadores como o MIPS incluem uma instrução jump register (jr), significando um desvio incondicional para o endereço especificado em um registrador.
- Depois desvia para o endereço apropriado usando essa instrução.

jump register

- No caso de desvios, vimos até agora:
  - CONDICIONAIS: beq → desvie se for igual
    bne → desvie se não for igual
  - INCONDICIONAL: j → jump: desvie
- O MIPS possui mais uma instrução de desvio INCONDICIONAL:

  jr $ra # Desvia para o endereço armazenado no registrador $ra (31)

  → JUMP REGISTER (especial)
- Sintaxe PC ← GPR[rs]

OpCode	rs	Zeros...	6 LSBs
6 bits	5 bits (31)	15 bits	JR = 8 JALR = 9
0	11111	000000000000000	001000 001001

jump register

- A instrução de jump register pula para o endereço armazenado no registrador $ra — que é exatamente o que queremos.
- Assim, o programa que chama, ou caller, coloca os valores de parâmetro em $a0-$a3 e utiliza jal X para desviar para o procedimento X (às vezes denominado callee).
- O callee, então, realiza os cálculos, coloca os resultados em $v0-$v1 e retorna o controle para o caller usando jr $ra.

jump and link

- A instrução de jump-and-link (jal) é escrita simplesmente como
- jal EndereçoProcedimento
- Salta para um endereço e simultaneamente salva o endereço da instrução seguinte em um registrador ($ra).
- A parte do link no nome da instrução significa que um endereço ou link é formado de modo a apontar para o local de chamada, permitindo que o procedimento retorne ao endereço correto.
- Esse “link”, armazenado no registrador $ra, é denominado endereço de retorno.
- Sintaxe:

OpCode	Endereço
6 bits	26 bits
3	Endereço

Procedimento

Na execução de um procedimento, o programa precisa seguir estas seis etapas:

1. Colocar parâmetros em um lugar onde o procedimento possa acessá-los.
2. Transferir o controle para o procedimento.
3. Adquirir os recursos de armazenamento necessários para o procedimento.
4. Realizar a tarefa desejada.
5. Colocar o valor de retorno em um local onde o programa que o chamou possa acessá-lo.
6. Retornar o controle para o ponto de origem, pois um procedimento pode ser chamado de vários pontos em um programa.

Procedimento

- O software do MIPS utiliza a seguinte convenção na alocação de seus 32 registradores:
  - $a0-$a3: quatro registradores de argumento, para passar parâmetros
  - $v0-$v1: dois registradores de valor, para valores de retorno
  - $ra: um registrador de endereço de retorno, para retornar ao ponto de origem
- Além de alocar esses registradores, o assembly do MIPS inclui uma instrução apenas para os procedimentos: ela desvia para um endereço e simultaneamente salva o endereço da instrução seguinte no registrador $ra.

Procedimento

- Para dar suporte a tais situações, computadores como o MIPS utilizam uma instrução de jump register (jr), apresentada anteriormente para ajudar com as instruções case, significando um desvio incondicional para o endereço especificado em um registrador: jr $ra
- A instrução de jump register pula para o endereço armazenado no registrador $ra — que é exatamente o que queremos.
- O programa que chama, ou caller, coloca os valores de parâmetro em $a0-$a3 e utiliza jal X para desviar para o procedimento X (às vezes denominado callee).
- O callee, então, realiza os cálculos, coloca os resultados em $v0-$v1 e retorna o controle para o caller usando jr $ra.

Procedimento

- Num programa armazenado é necessário ter um registrador para manter o endereço da instrução atual sendo executada.
- Este é o PC (Program Counter) na arquitetura MIPS (um nome mais sensato teria sido registrador de endereço de instrução).
- A instrução jal salva o PC + 4 no registrador $ra para o link com a instrução seguinte, a fim de preparar o retorno do procedimento.

Procedimento

- Suponha que um compilador precise de mais registradores para um procedimento do que os quatro registradores para argumentos e os dois para valores de retorno.
- Como temos de cobrir nossos rastros após o término desta missão, quaisquer registradores necessários ao caller deverão ser restaurados aos valores que possuíam antes de o procedimento ser chamado.
- A estrutura de dados ideal para armazenar os spilled registers é uma pilha — uma fila do tipo “último a entrar, primeiro a sair”.
- Uma pilha precisa de um ponteiro para o endereço alocado mais recentemente na pilha, a fim de mostrar onde o próximo procedimento deverá colocar os spilled registers ou onde os valores antigos dos registradores estão localizados.

Procedimento

- O stack pointer é ajustado em uma palavra para cada registrador salvo ou restaurado.
- O software MIPS reserva o registrador 29 para o stack pointer, dando-lhe o nome óbvio $sp.
- As pilhas são tão comuns que possuem seus próprios termos para transferir dados da pilha e para ela: colocar dados na pilha é denominado push, e remover dados da pilha é denominado pop.

Operadores

- Caller: é o programa que chama o procedimento, fornecendo os valores dos parâmetros. Faz uso dos registradores de $a0 a $a3 para armazenar os parâmetros e também de jal para pular para o procedimento.
- Callee: é um procedimento que executa uma série de instruções armazenadas com base nos parâmetros fornecidos pelo Caller e depois retorna o controle para o Caller novamente.
- Spilled Registers: é o processo de colocar variáveis menos utilizadas na memória (ou variáveis que serão necessárias mais adiante).
- Pilha: São estruturas de dados do tipo LIFO (last-in first-out), onde o último elemento a ser inserido, será o primeiro a ser retirado. Assim, uma pilha permite acesso a apenas um item de dados - o último inserido.

Operadores

- Stack Pointer: é um valor que indica o endereço alocado mais recentemente em uma pilha, mostrando onde devem ser localizados os valores antigos dos registradores e onde os Spilled Registers devem ser armazenados. O registrador 29 é usado para o $sp.
- Push: coloca palavras para cada registrador salvo ou restaurando na pilha.
- Pop: remove palavras da pilha. Valores são retirados da pilha pela soma do valor do stack pointer.
- Por motivos históricos, as pilhas “crescem” de endereços maiores para endereços menores. Essa convenção significa que você põe valores na pilha subtraindo do valor do stack pointer. Somar ao stack pointer diminui essa pilha, removendo seus valores.

Interface HW/SW

- Uma variável em C é um local na memória e sua interpretação depende tanto do seu tipo quanto da sua classe de armazenamento. Ex. Integer, String, etc...
- A linguagem C possui duas classes de armazenamento: automática e estática.
  - Automáticas são locais a um procedimento e são descartadas quando o procedimento termina.
  - Estáticas permanecem durante entradas e saídas de procedimentos.
- As variáveis C declaradas fora de todos os procedimentos são consideradas estáticas, assim como quaisquer variáveis declaradas por meio da palavra reservada static. As outras são automáticas.
- Para simplificar o acesso aos dados estáticos, o software do MIPS reserva outro registrador, chamado ponteiro global ou $gp (registrador reservado para apontar para a área estática).

Pilha: novos dados

- A pilha também é utilizada para armazenar variáveis que são locais ao procedimento, que não cabem nos registradores, como arrays ou estruturas locais.
- O segmento da pilha que contém os registradores salvos e as variáveis locais de um procedimento é chamado frame de procedimento ou registro de ativação.
- O frame pointer ($fp) aponta para a primeira palavra do frame, normalmente um registrador de argumento salvo, e o stack pointer ($sp) aponta para o topo da pilha.
- A pilha é ajustada de modo a criar espaço para todos os registradores salvos e quaisquer variáveis locais residentes na memória.

Pilha: novos dados

- Como o stack pointer pode mudar durante a execução do programa, é mais fácil para os programadores referenciarem variáveis por meio do frame pointer estável, embora isso também pudesse ser feito por meio do stack pointer e um pouco de aritmética de endereços.
- Se não houver variáveis locais na pilha dentro de um procedimento, o compilador ganhará tempo não atribuindo um endereço ao frame pointer, e depois, restaurando-o.
- Quando um frame pointer é usado, ele é inicializado usando o endereço que está no $sp em uma chamada, e o $sp é restaurado usando o valor do $fp.

Pilha

- C: int exemplo_folha (int g, int h, int i, int j)

```
{
    int f
    f = (g + h) - (i + j)
    return f
}
```

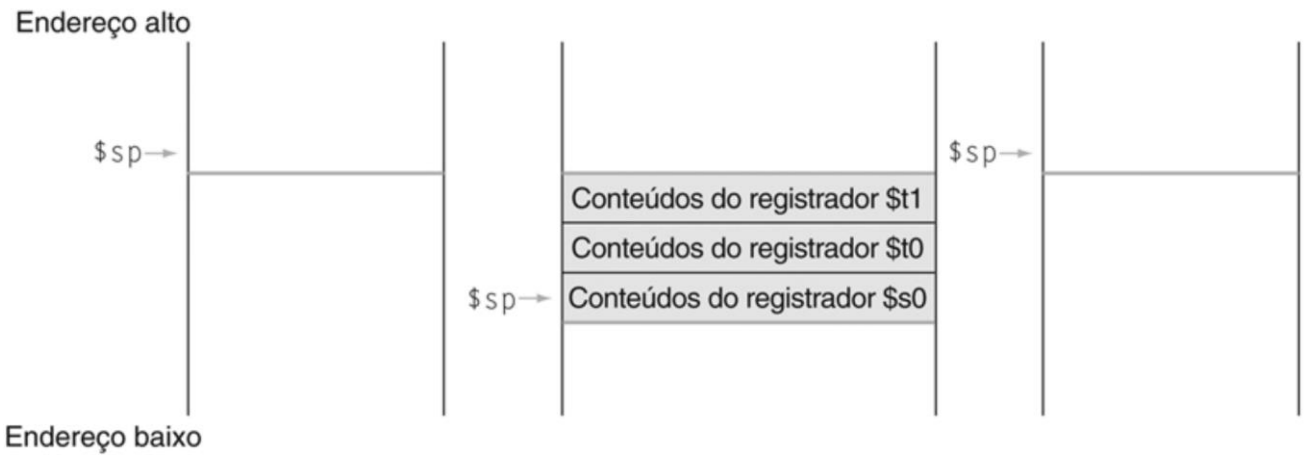
- Qual é o código assembly do MIPS compilado?

Pilha

- As variáveis de parâmetro g, h, i e j correspondem aos registradores de argumento $a0, $a1, $a2 e $a3, e f corresponde a $s0.
- O programa compilado começa com o rótulo do procedimento: exemplo_folha
- O próximo passo é salvar os registradores usados pelo procedimento.
- Precisamos salvar três registradores: $s0, $t0 e $t1.
- “Empilhamos” os valores antigos, criando espaço para três palavras (12 bytes) na pilha e depois as armazenamos:
  - addi $sp, $sp, -12 # ajusta pilha criando espaço para 3 itens
  - sw $t1, 8($sp) # salva reg. $t1
  - sw $t0, 4($sp) # salva reg. $t0
  - sw $s0, 0($sp) # salva $s0

Pilha

- A pilha fica:



- Os valores do stack pointer e a pilha antes, durante e após a chamada do procedimento.
- O stack pointer sempre aponta para o “topo” da pilha ou para a última palavra na pilha.

Pilha

- As três instruções seguintes correspondem ao corpo do procedimento

Código MIPS: add $t0, $a0, $a1 → reg. t0 recebe G + H

add $t1, $a2, $a3 → reg. t1 recebe I + J

sub $s0, $t0, $t1 → F recebe (G + H) - (I + J)

- Para retornar o valor de f, nós o copiamos para um registrador de valor de retorno: add $v0, $zero, $s0
- Antes de retornar, restauramos os três valores antigos dos registradores que salvamos, desempilhando-os :

```
lw $s0, 0 ($sp)      # restaura $s0 para o caller
lw $t0, 4 ($sp)      # restaura $t0 para o caller
lw $t1, 8 ($sp)      # restaura $t1 para o caller
addi $sp, $sp, 12    # ajusta a pilha para excluir os três itens
```

Pilha

- O procedimento termina com um jump register usando o endereço de retorno

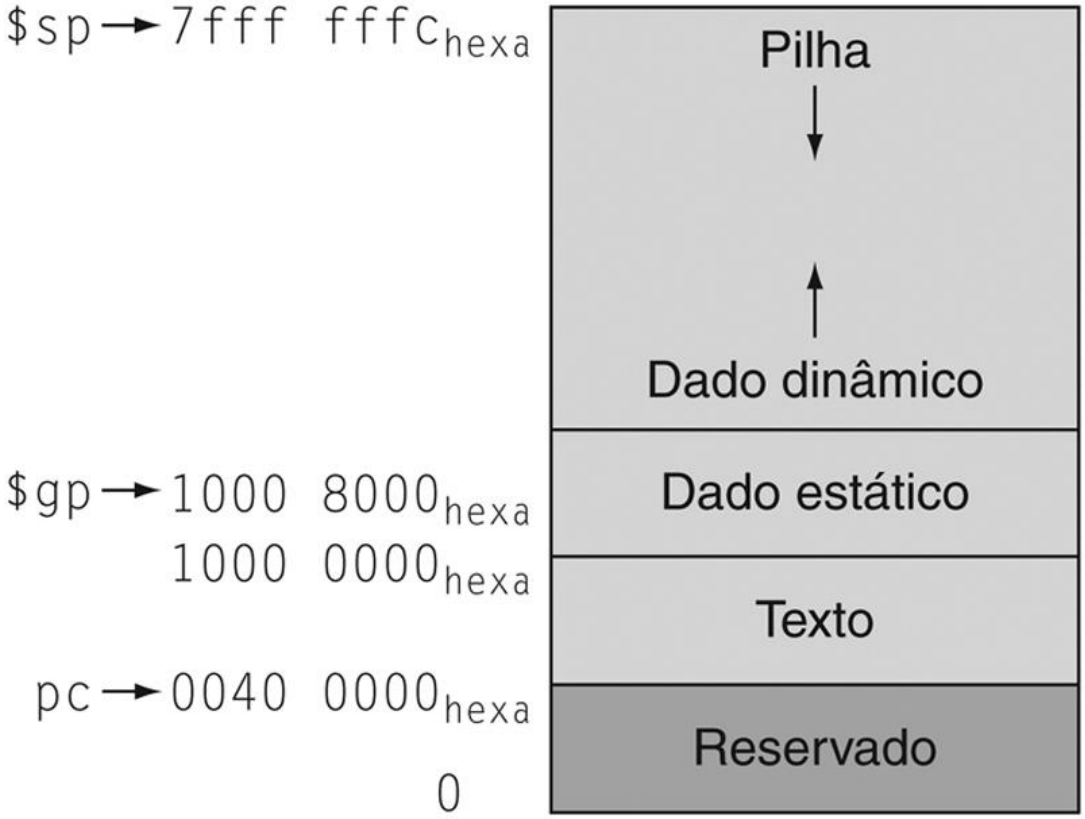
```
jr $ra          # desvia de volta à rotina que o chamou.
```

Organização da Memória

- Como foi visto, o endereçamento do memória do MIPS é feito com base em um dos registradores.
- lw $t0, 100($s0) Registrador que armazena o endereço de base do vetor.
- Os registradores do MIPS possuem 32 bits e os endereços de memória também são formados por 32 bits.
- O MIPS enxerga a memória como um grande vetor, indexado de 4 em 4 posições.
- Entretanto, o processador não pode ter acesso direto a qualquer posição de memória.

Organização da Memória

- Para o MIPS:



Organização da Memória

Esses endereços são apenas uma convenção do software e não fazem parte da arquitetura MIPS.

- De cima para baixo, o stack pointer é inicializado com 7fff $fffc_{hexa}$ e cresce para baixo, em direção ao segmento de dados.
- Na outra extremidade, o código do programa (“texto”) começa em 0040 0000_{hexa}.
- Os dados estáticos começam em 1000 0000_{hexa}.
- Os dados dinâmicos vêm seguida e crescem para cima em direção à pilha, em uma área chamada heap.
- O ponteiro global, $gp, é definido como endereço para facilitar o acesso aos dados. Ele é inicializado com 1000 8000_{hexa} para poder acessar de 1000 0000_{hexa} até 1000 $ffff_{hexa}$ usando os offsets de 16 bits positivos e negativos a partir do $gp.

Organização da Memória

- O segmento de pilha é utilizado para o armazenamento de dados temporários, como por exemplo quando os registradores do processador estão todos ocupados (DERRAMAMENTO DE REGISTRADORES).
- Os vetores são sempre armazenados no segmento de dados.
- O endereço base de um vetor pode ser qualquer endereço, múltiplo de 4, que esteja acima de $1000\ 0000_{(16)}$.

Operandos Imediatos

- Os endereços possuem 32 bits
- A carga não pode ser feita através da instrução addi porque**:**

001000	00000	01000	0000000000001010
6 bits	**5 bits**	**5 bits**	**16 bits**

A instrução addi só consegue carregar números de 16 bits

- Devem ser usadas duas novas instruções:
  - 1º carregamos os 16 bits de ordem mais alta:

    lui $t0, 61 → load upper immediate
  - 2º carregamos os 16 bits de ordem mais baixa:

    ori $t0, $t0, 2304 → or immediate

Operandos Imediatos

- Ex.: Suponha que deseja-se carregar o seguinte número binário no registrador $t0:

$$\underbrace{0000\ 0000\ 0011\ 1101}_{\mathbf{61_{(10)}}}\ \underbrace{0000\ 1001\ 0000\ 0000}_{\mathbf{2304_{(10)}}}$$

- 1º passo: carregar os 16 bits mais significativos:

  $0000\ 0000\ 0011\ 1101_{(2)} = 61_{(10)}$ → lui $t0, 61

- Depois da execução desta instrução o valor de $t0 é

  0000 0000 0011 1101 0000 0000 0000 0000

- 2º passo: carregar os 16 bits menos significativos:

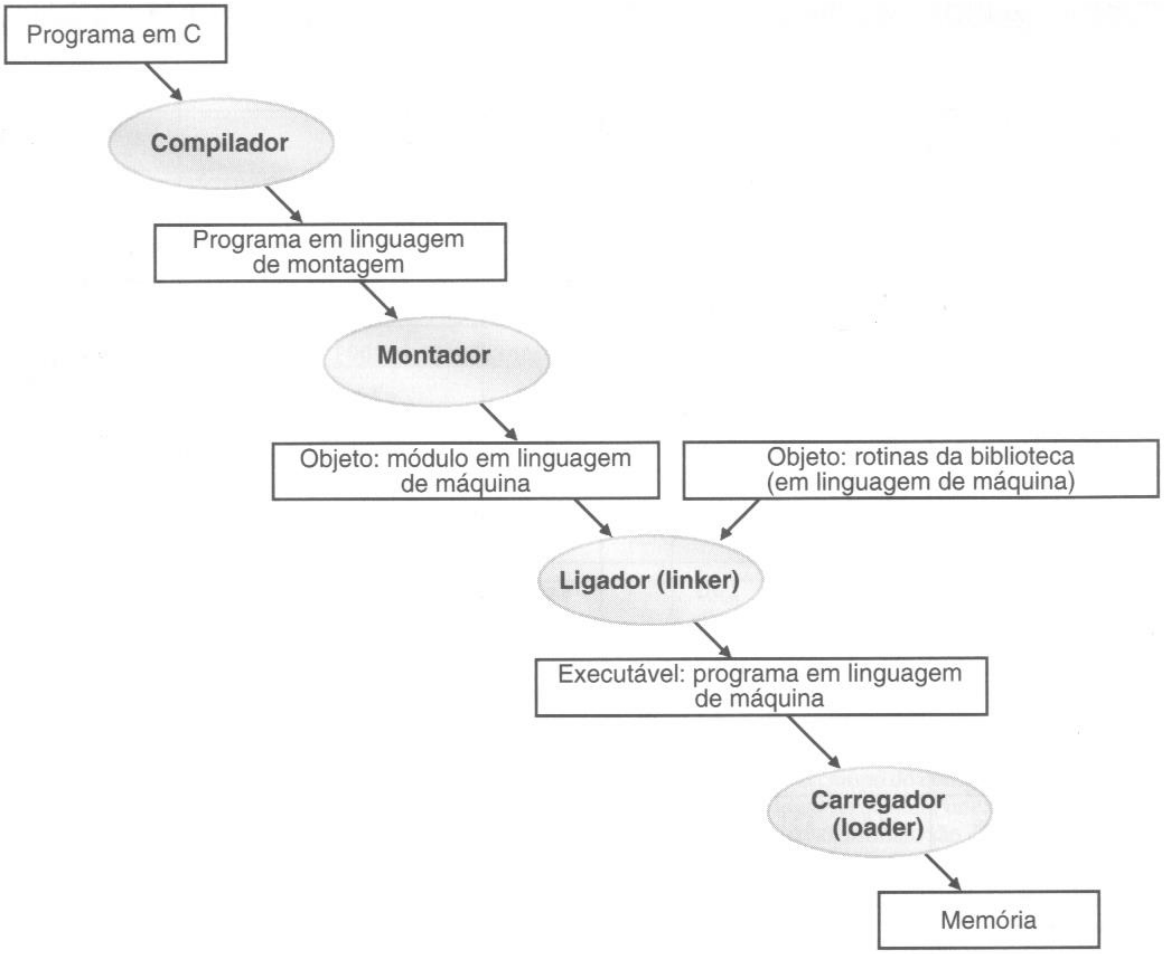
  $0000\ 1001\ 0000\ 0000_{(2)} = 2304_{(10)}$ → ori $t0, $t0, 2304

- Após esta instrução, o registrador $t0 contém

  0000 0000 0011 1101 0000 1001 0000 0000

Execução do Programa

- O processo de transformação de um programa escrito em linguagem de alto nível em um programa executável possui 4 passos.
- Alguns sistemas juntam alguns desses passos para reduzir o tempo de compilação do programa.



Compilador

- O compilador transforma um programa em linguagem de alto nível (C, Pascal, Cobol) em um expresso em linguagem de montagem.
- A linguagem de montagem é simplesmente uma linguagem simbólica que representa as instruções de determinado processador.
- Um mesmo programa escrito em duas linguagem de alto nível diferentes podem possuir exatamente o mesmo código de máquina, uma vez que o programa deve executar as mesmas tarefas.
- As vantagens da linguagem de alto nível foram vistas no capítulo 1 (ex.: produtividade, linguagens dedicadas, independência do hardware, etc.).

Montador

- Responsável por traduzir um programa em linguagem de montagem para um programa em código de máquina (o nível mais baixo).
- Aceita pseudo-instruções e as traduz para as instruções reais do processador.

Ex.: load immediate → li $t0, 100 # $t0 = 100

traduzida em

addi $t0, $zero, 100

move → move $t0, $t1 # $t0 = $t1

traduzida em

add $t0, $zero, $t1

- O conjunto completo de instruções do processador encontra-se no manual.

Montador

- O montador aceita instruções expressas em hexadecimal, da mesma forma que a linguagem C.
- Esta possibilidade facilita a carga de endereços da memória em registradores.
- A carga de um endereço base de um vetor pode ser feita da seguinte forma:
  - lui $t0, 0x1000 #Carrega os 16 bits da esquerda
  - ori $t0, $t0, 0x000C #Carrega os 16 bits da direita
- O montador também é o responsável pelo controle dos LABELS utilizados em instruções de desvio, por exemplo:
  - slt $t0, $s6, $s7
  - beq $t0, $zero, SAI

Este label é convertido em um endereço de instrução pelo montador

Ligador

- Qualquer modificação em um código de alto nível determina que este código seja recompilado.
- Em programas muito grandes esta compilação pode levar horas, o que se traduz em baixa produtividade.
- Solução: separar o programa em partes independentes, compilar cada parte separada e depois juntá-las.
- O LIGADOR (linker ou link-editor) é um programa responsável pela junção das partes de um programa.
- O ligador pega as partes do programa, JÁ EM LINGUAGEM DE MÁQUINA, e as une, gerando o código final.

Ligador

- A separação do código de um programa em pedaços traz dois benefícios:
  - Como já visto, mudanças no código são recompiladas rapidamente.
  - A separação permite a utilização de funções preexistentes (ex.: printf, scanf).
  - As funções das bibliotecas da linguagem são inseridas no código pelo ligador. → REUTILIZAÇÃO DE CÓDIGO
- O ligador produz o arquivo executável final, que pode ser executado pela máquina.

Carregador

- O carregador é o programa responsável por carregar o executável do disco para a memória.
- O carregador executa as seguintes tarefas:
  - Reserva um espaço de memória grande o suficiente para armazenar o código e os dados do programa.
  - A seguir, copia as instruções e os dados para a memória.
  - Inicializa os registradores e a memória.
  - Executa uma rotina de inicialização da máquina e chama a rotina principal do programa.

Bibliografia

- Capítulo 2: Patterson, David A. Organização e Projeto de Computadores: A Interface Hardware / Software, 5a. Edição. GEN LTC, 08/2017.