Bacharelado em Ciências da Computação

Software Básico

CET088

Prof. Dany S. Dominguez

dsdominguez@uesc.br
NBCGIB – Sala 01
Tel. 3680 5212

Tema 01:▫ Visão geral de sistemas computacionais

Roteiro

- Introdução
- Que é informação?
- Criando o arquivo executável
- Relevância do sistema de compilação
- Organização do hardware
  - Buses
  - Dispositivos I/O
  - Memória principal
  - Processador

Objetivos

- Compreender o conceito de representação da informação e sua dependência do contexto
- Conhecer as etapas do processo de compilação, destacando a relevância dele no desempenho e segurança do programa
- Conhecer os componentes da arquitetura de HW de sistemas computacionais, e a função de cada um destes componentes

Introdução

- **Sistema computacional** = ao conjunto formado pelo **hardware e o sistema operacional** que trabalham em conjunto para executar aplicações
- Implementações específicas de sistemas mudam ao longo do tempo, mais os conceitos subjacentes não
- Todos os sistemas computacionais apresentam componentes de hw e sw semelhantes que desempenham as mesmas funções
- Nosso objetivo é compreender como esses componentes funcionam e como afetam a corretude e desempenho das aplicações que desenvolvemos

Introdução

- Aprenderemos conceitos e habilidades práticas para
  - Evitar erros numéricos incomuns causados pela forma na qual os computadores representam os números
  - Otimizar código C para aproveitar as características de desenho dos processadores e sistemas de memória modernos
  - Evitar as falhas de seguridade associadas com as vulnerabilidades de desbordamento de buffer
  - Reconhecer e evitar os erros desagradáveis que aparecem no processo de linkagem

Introdução

- Em muitos cursos o programa "*Hello world*" é usado para introduzir a linguagem C

```c
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

- Ciclo de vida do programa
  1. Elaboração pelo programador
  2. Conversão em código de maquina
  3. Execução pelo sistema
  4. Impressão da mensagem
  5. Encerramento da execução

- Acompanhamos o ciclo de vida do programa para introduzir os conceitos principais, a terminologia e os componentes de um sistema computacional
- Estas ideias serão aprofundadas ao longo do curso

Informação = Bits + Contexto

- O programa "*Hello world*" inicia seu ciclo de vida como um programa fonte (arquivo fonte)
- O programa é criado usando um editor de textos e armazenado em um arquivo texto (hello.c)
- O arquivo fonte é uma sequência de bits, cada um deles com valor 0 ou 1, organizados em pedaços de 8 bits (byte)
- Cada um destes bytes representa um caractere no programa
- Os sistemas computacionais representam caracteres texto usando a tabela ASCII, cada caractere é associado a um valor inteiro de 8 bits
- Atenção: ASCII não é o único standard. Unicode ou UTF-8

Informação = Bits + Contexto

- Representação ASCII do programa "*Hello world*"

#	i	n	c	l	u	d	e	*SP*	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	*SP*	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	*SP*	*SP*	*SP*	*SP*	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	*SP*	w	o	r	l	d	\	n	"	)	;	\n	*SP*
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	32
SP	*SP*	*SP*	r	e	t	u	r	n	*SP*	0	;	\n	}	\n	
32	32	32	114	101	116	117	114	110	32	48	59	10	125	10	

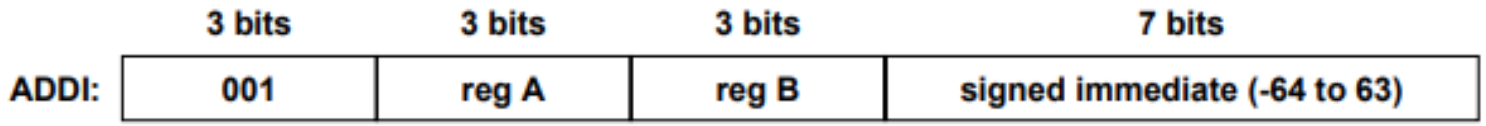
- Arquivos compostos exclusivamente por carácteres ASCII são chamados arquivos texto
- Os outros arquivos são chamados de arquivos binários

Informação = Bits + Contexto

- Toda a informação em um sistema computacional:
  - Arquivos em disco
  - Programas armazenados na memória
  - Dados de usuário em memória
  - Dados transferidos pela rede, ...
- São representados como um conjunto de bits
- Dependendo do contexto, a mesma sequência de bytes pode representar
  - Um número inteiro
  - Um número de ponto flutuante
  - Uma string de caracteres
  - Uma instrução de maquina

Informação = Bits + Contexto

- Exemplo:
  - Binário de 16 bits (2 bytes): 0011 1101 0100 0101
  - **15685** Unsigned Int
  - **2.197936E-41** Float
  - '=**E**' se for uma string em ASCII
  - Instrução de máquina (RISC 16 bits)



addi	Add Immediate RRI-type	001	addi rA, rB, imm	Add contents of **regB** with **imm**, store result in **regA**.

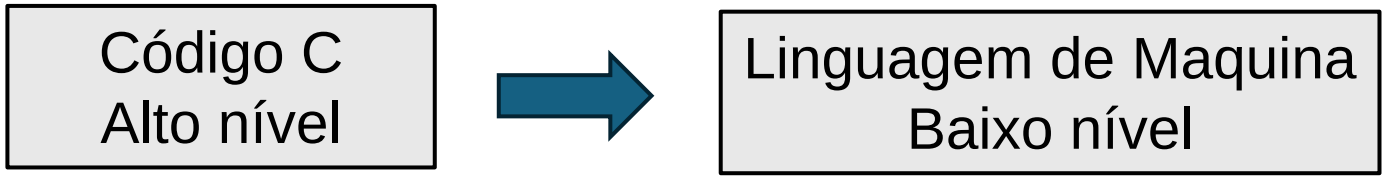
Informação = Bits + Contexto

- Porque devemos conhecer a representação dos números em sistemas computacionais?

  - As representações numéricas NÃO são exatamente números (inteiros ou reais)
  - São representações finitas (espaço de memória limitado)
  - Elas podem ter comportamentos muito diferentes do esperado no campo da matemática

Convertendo código fonte em linguagem de maquina

- Para criarmos um programa executável o programa passa por várias etapas de tradução (transformação)

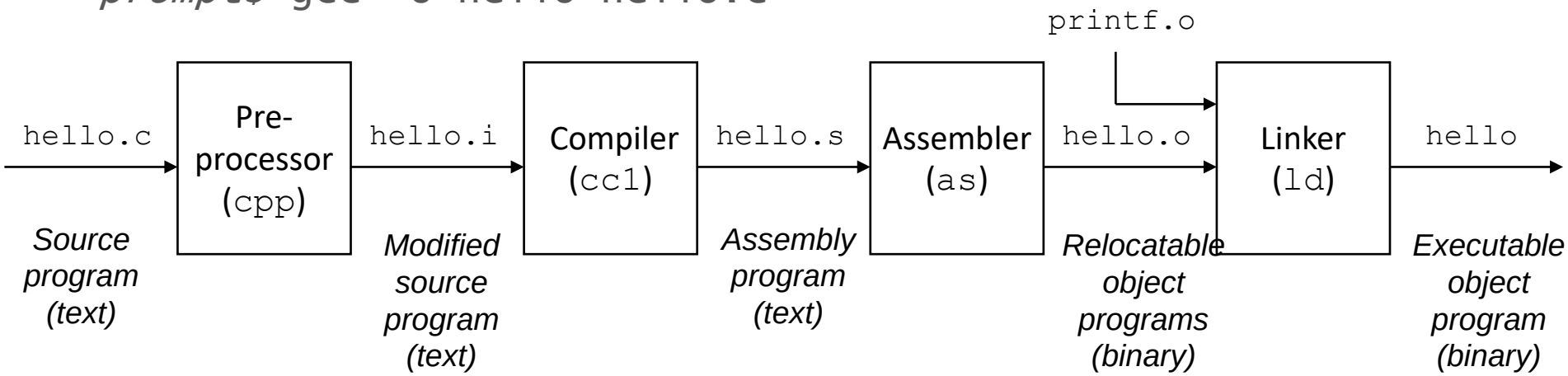


- O programa *hello.c* inicia seu ciclo de vida como um código C de alto nível que pode ser lido e compreendido por humanos*
- Para podermos executar o código as instruções individuais devem ser convertidas (por outros programas) em uma sequência de instruções de maquina de baixo nível
- Estas instruções são empacotadas em um objeto programa executável e armazenadas em disco como um arquivo binário

Convertendo código em linguagem de maquina

- No Linux (Unix-like) a transformação do arquivo fonte ao objeto executável é realizada pelo compilador (controlador do processo)

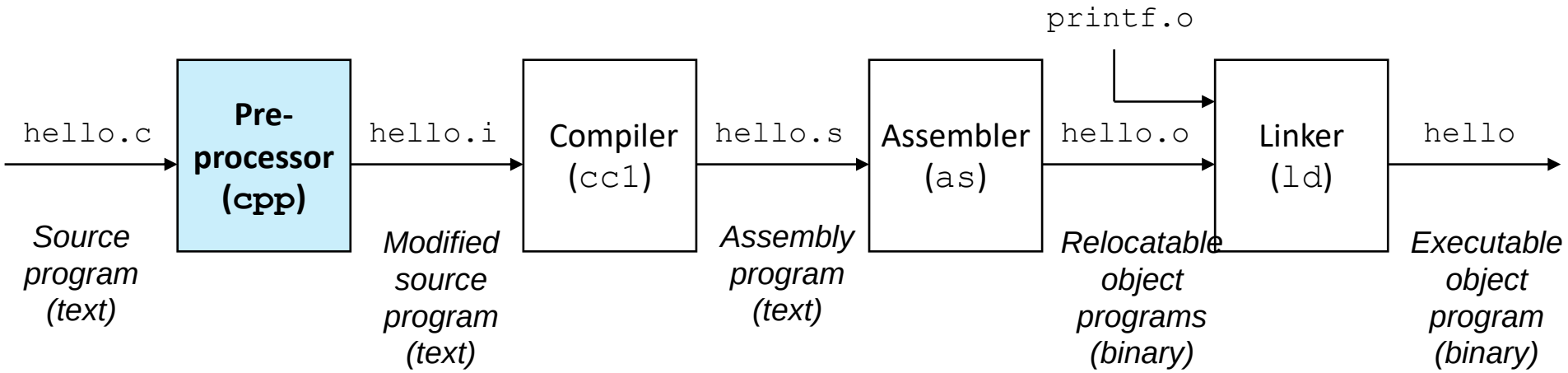
```
prompt$ gcc -o hello hello.c
```



- O processo de compilação ocorre em 4 etapas

1. Pré-processamento
2. Compilação
3. Assembler
4. Linking

Fonte → Executável. Pré-processamento

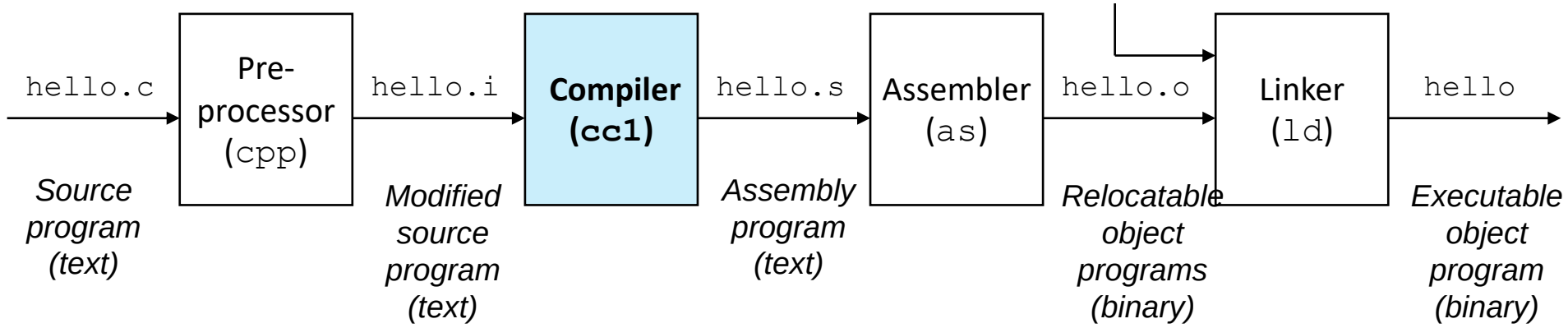


- Modifica o arquivo fonte original de acordo com as diretivas que iniciam com o caractere #

```
#include <stdio.h>
```

- O pré-processador lê o conteúdo do arquivo de cabeçalhos (stdio.h) e o insere no arquivo fonte (hello.c)
- Resultando, em outro arquivo de código, geralmente com a extensão .i

Fonte → Executável. Compilação

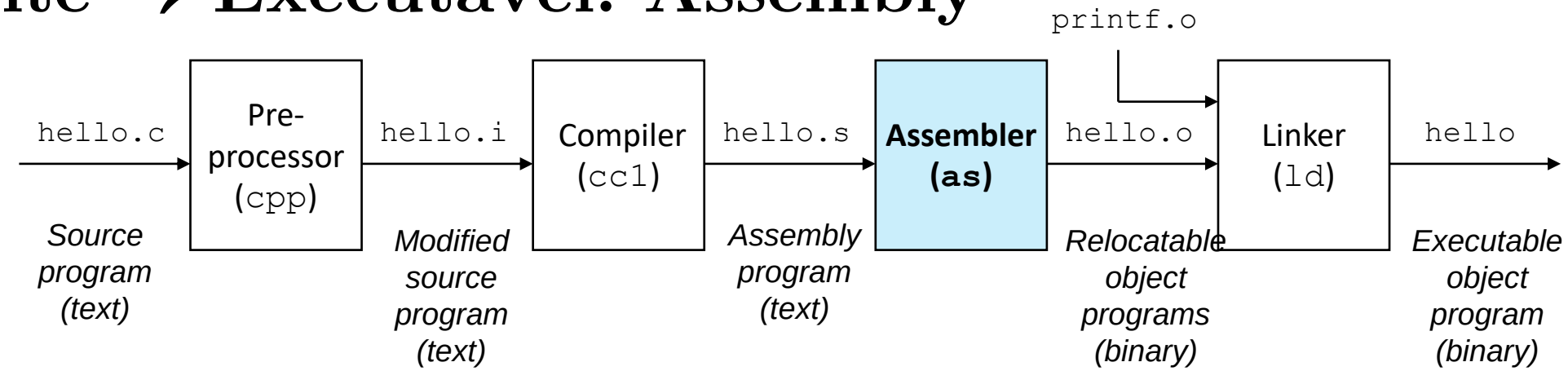


- Converte o arquivo texto hello.i no arquivo texto hello.s
- hello.s contêm o programa em linguagem *assembly*
- Cada linha descreve uma instrução de máquina de forma textual

```
1   main:
2       subq    $8, %rsp
3       movl    $.LC0, %edi
4       call    puts
5       movl    $0, %eax
6       addq    $8, %rsp
7       ret
```

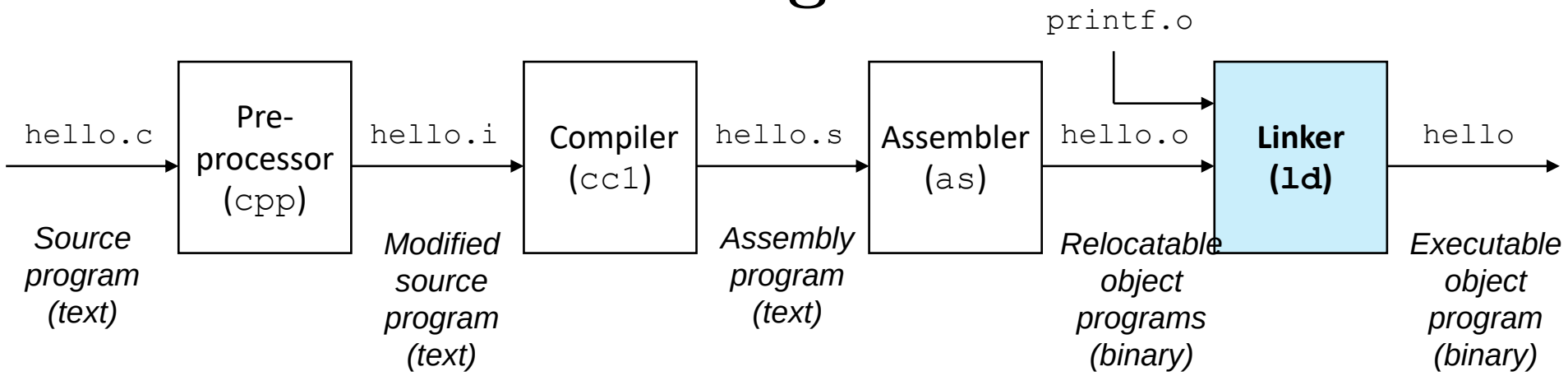
- A linguagem *assembly* fornece uma linguagem de saída comum de diferentes compiladores para diferentes linguagens de alto nível

Fonte → Executável. Assembly



- O ensamblador converte o arquivo hello.s em instruções de linguagem de máquina
- Empacota estas instruções em um programa objeto relocalizável (relocatable object program)
- Armazena o programa objeto resultante no arquivo hello.o (arquivo binário)

Fonte → Executável. Linking



- O programa hello chama a função printf, a qual faz parte da biblioteca standard de C oferecida por todos os compiladores
- A função printf está armazenada em arquivo objeto precompilado chamado printf.o
- O linker (ld) manipula a combinação (merge) entre os arquivos hello.o e printf.o
- O resultado é um programa executável que está pronto para ser carregado em memória e executado pelo sistema

Importância do sistema de compilação

- Para programas simples com baixo consumo de recursos podemos confiar que o sistema de compilação produzirá código de máquina correto e eficiente
- Por outro lado, existe um conjunto importante de razões pelas quais o programador deve conhecer o funcionamento do sistema de compilação
  1. Otimização do desempenho do programa
  2. Entender e poder solucionar os erros em tempo de linkagem
  3. Evitar problemas de segurança

Importância do sistema de compilação

- Otimização do desempenho do programa

1. Sempre é mais eficiente utilizar uma instrução switch que uma sequência de instruções if-else?
2. Qual é o custo da sobrecarga provocada por uma chamada à função?
3. O acesso aos elementos de um vetor é mais eficiente usando ponteiros ou subscritos?
4. Porque um laço roda muito mais rápido se somamos em uma variável local em vez de um argumento que é passado por referência?
5. Como uma função pode ser executada mais rapidamente ao reorganizarmos os parênteses de uma operação aritmética?

Importância do sistema de compilação

- Entendendo os erros de linkagem

1. Os erros de programação mais sorprendentes estão relacionados com a operação de linkagem, especialmente quando construímos sistemas grandes
2. Que significa quando o linker informa que não consegue resolver uma referência?
3. Qual é a diferença entre uma variável estática e uma variável global?
4. O que acontece se você definir duas variáveis globais em diferentes arquivos C com o mesmo nome?
5. Qual é a diferença entre uma biblioteca estática e uma biblioteca dinâmica?
6. Por que é importante a ordem em que listamos as bibliotecas na linha de comando?
7. Por que alguns erros relacionados ao linker não aparecem até o tempo de execução?

Importância do sistema de compilação

- Evitando problemas de segurança

1. Vulnerabilidades de *buffer overflow* foram responsáveis por muitas das falhas de segurança em servidores de rede e de Internet
2. Apenas poucos programadores entendem a necessidade de restringir cuidadosamente a quantidade e a forma dos dados que aceitamos de fontes não confiáveis
3. É importante compreender as consequências da forma como os dados e as informações de controle são armazenados na pilha do programa (Stack)
4. Também devemos conhecer quais métodos que podem ser usados pelo programador, compilador e sistema operacional para reduzir a ameaça de um ataque.

Executando nosso programa

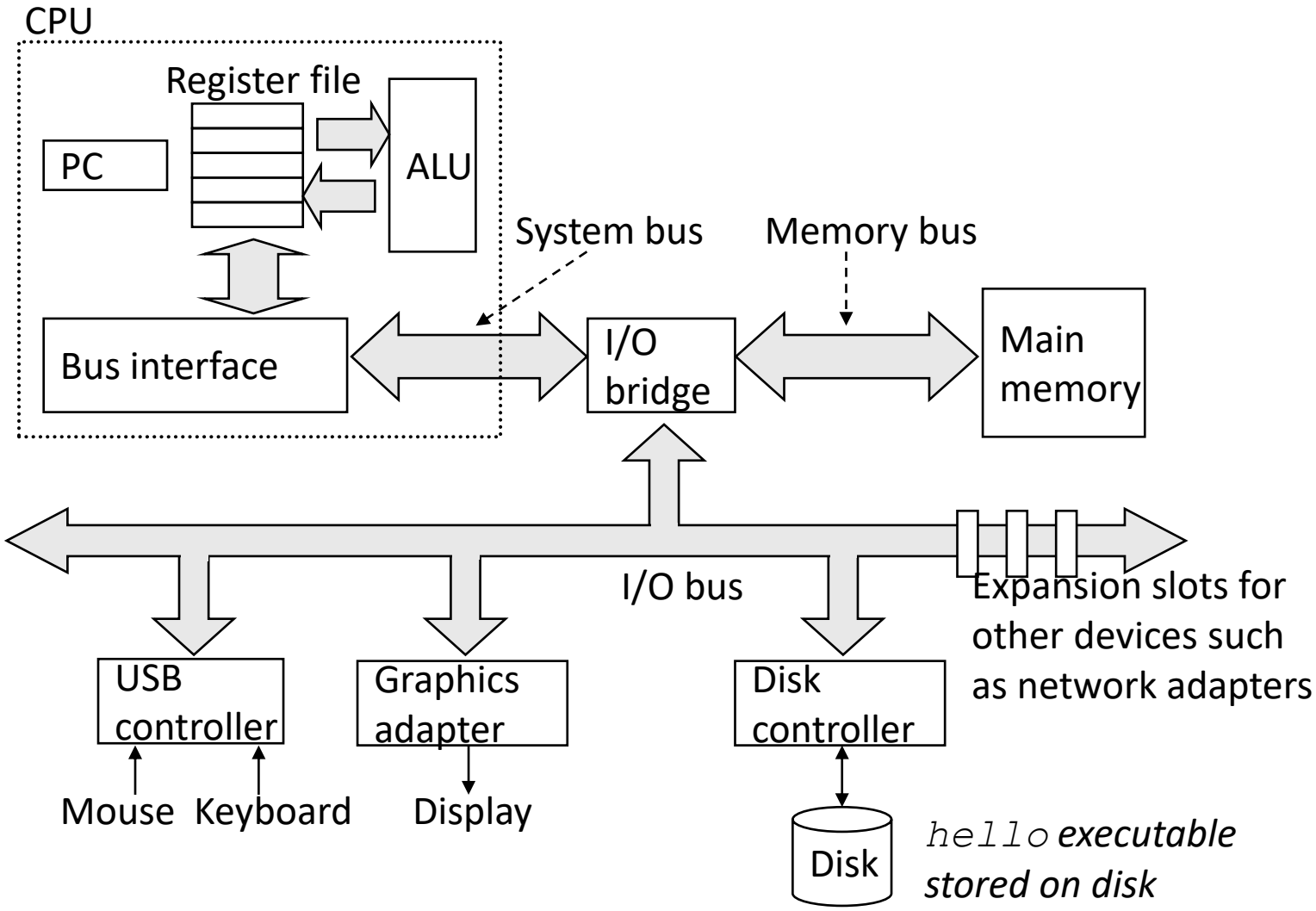
- Agora nosso código fonte é um programa executável, chamado hello e armazenado em nosso disco
- Para executar nosso programa

```
prompt$ ./hello
prompt$ hello, world
```

- O shell de Linux é um interpretador de linhas de comando (CLI, Command Line Interpreter)
  - Recebe um linha de comando
  - Executa o comando
  - Tipos de comando
    - Built-in commands
    - Programas executáveis
  - Programas executáveis são carregados e executados
  - Aguarda até a execução do programa finalizar

Organização do hardware

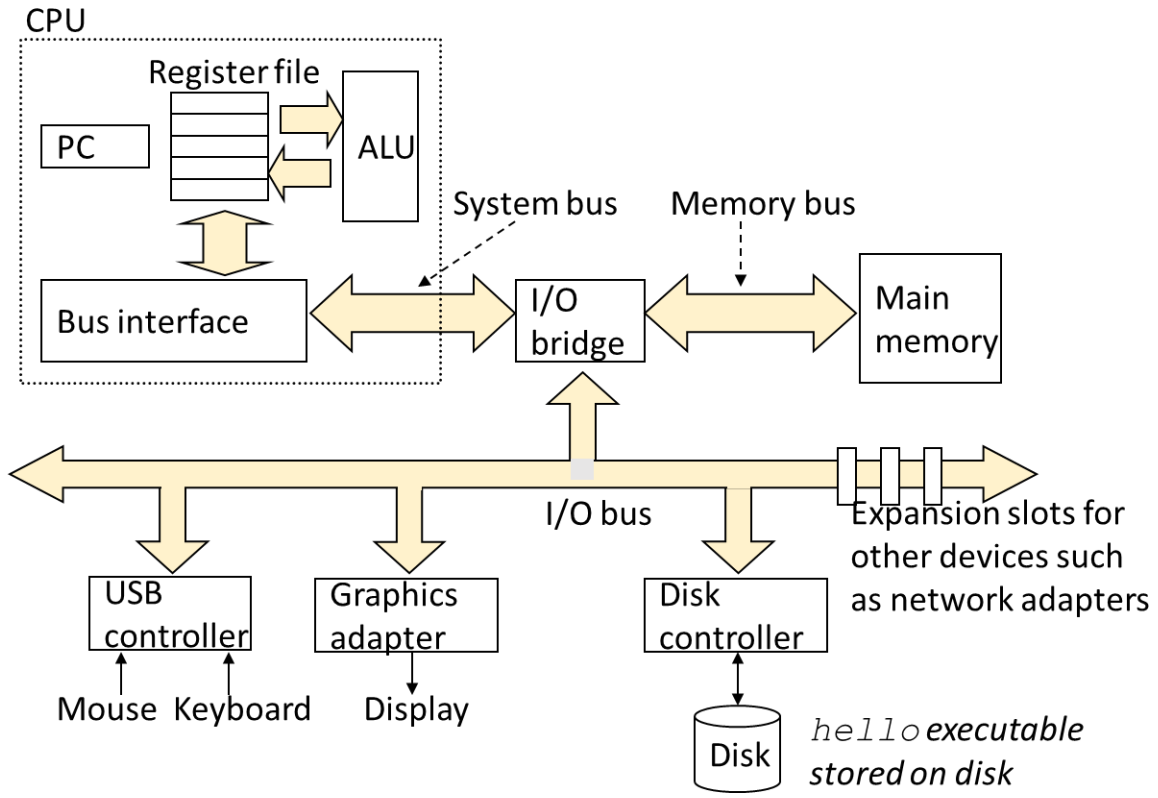
- Para compreendermos o que ocorre quando executamos um programa, precisamos entender a organização de hw de um sistema típico



- Modelo baseado na arquitetura Intel
- Todos os sistemas são semelhantes

Buses

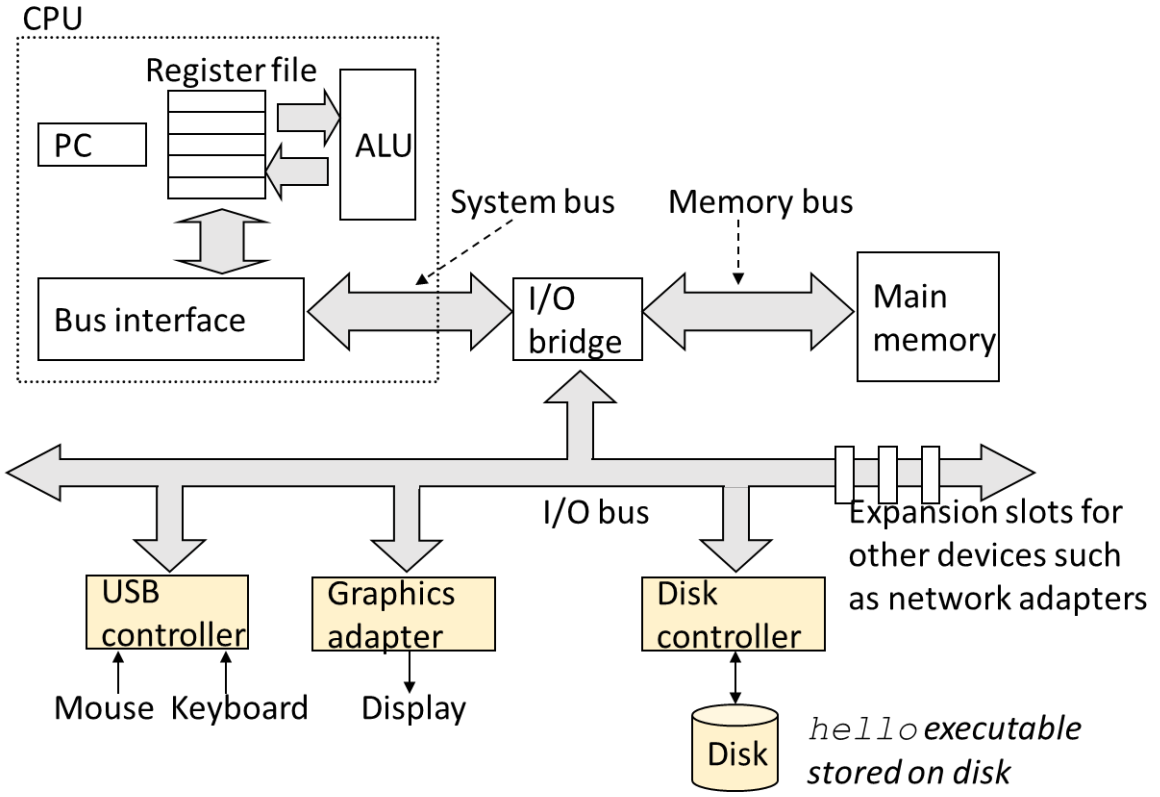
- Aparecem por todo o sistema
- São uma coleção de condutos elétricos que transportam bytes de informação entre os componentes
- São desenhados para transferir fatias de informação de tamanho fixo (palavra de memória)



- O tamanho de uma palavra de memória é um parâmetro fundamental do sistema
- Tamanhos de palavra típicos
  - 4 bits (arquitetura de 32 bits)
  - 8 bits (arquitetura de 64 bits)

Dispositivos I/O

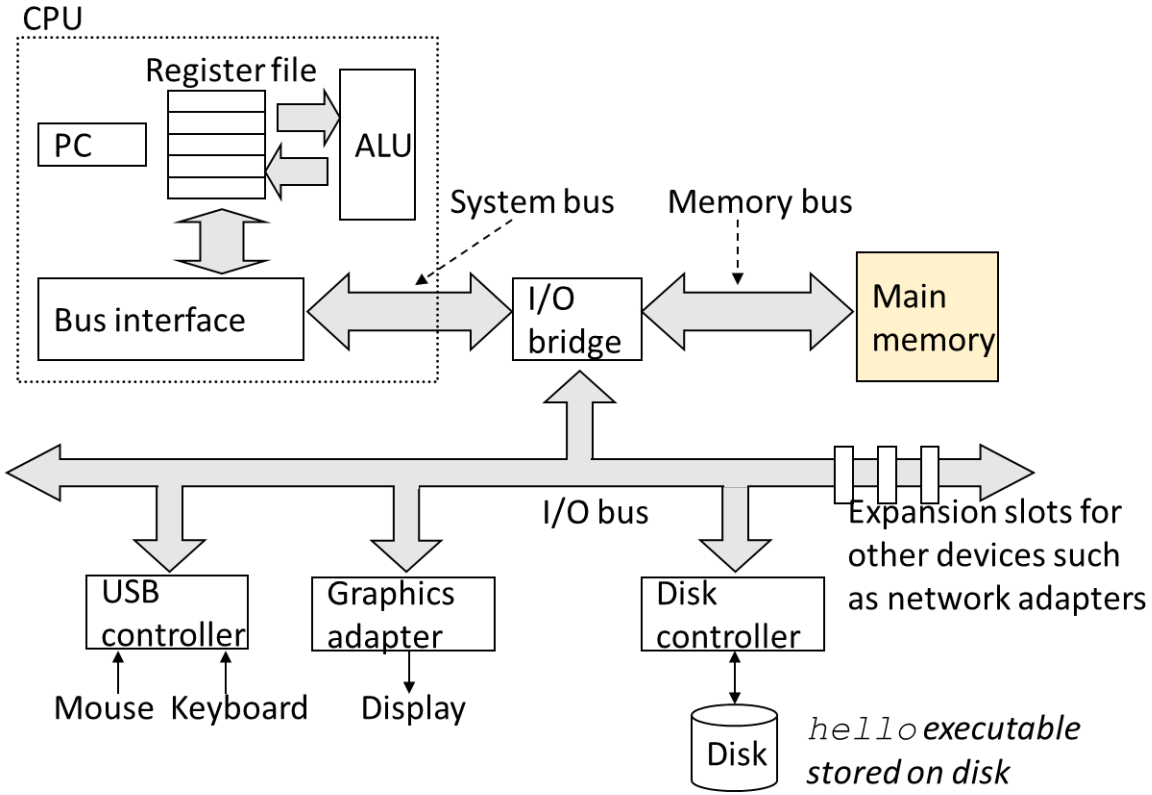
- São as ligações do sistema com o mundo exterior
- Exemplos:
  - Teclado e mouse para as entradas do usuário
  - Display, saída do usuário
  - Discos, para armazenamento de longo prazo de dados e programas



- Cada dispositivo I/O é conectado ao bus I/O através de um controlador ou adaptador
- Controlador: chip no próprio dispositivo ou soldado na placa mãe
- Adaptador: placa que é conectada a um slot na placa mãe
- O proposito destes (controlador/adaptador é transferir informação entre o dispositivo e o bus

Memória principal

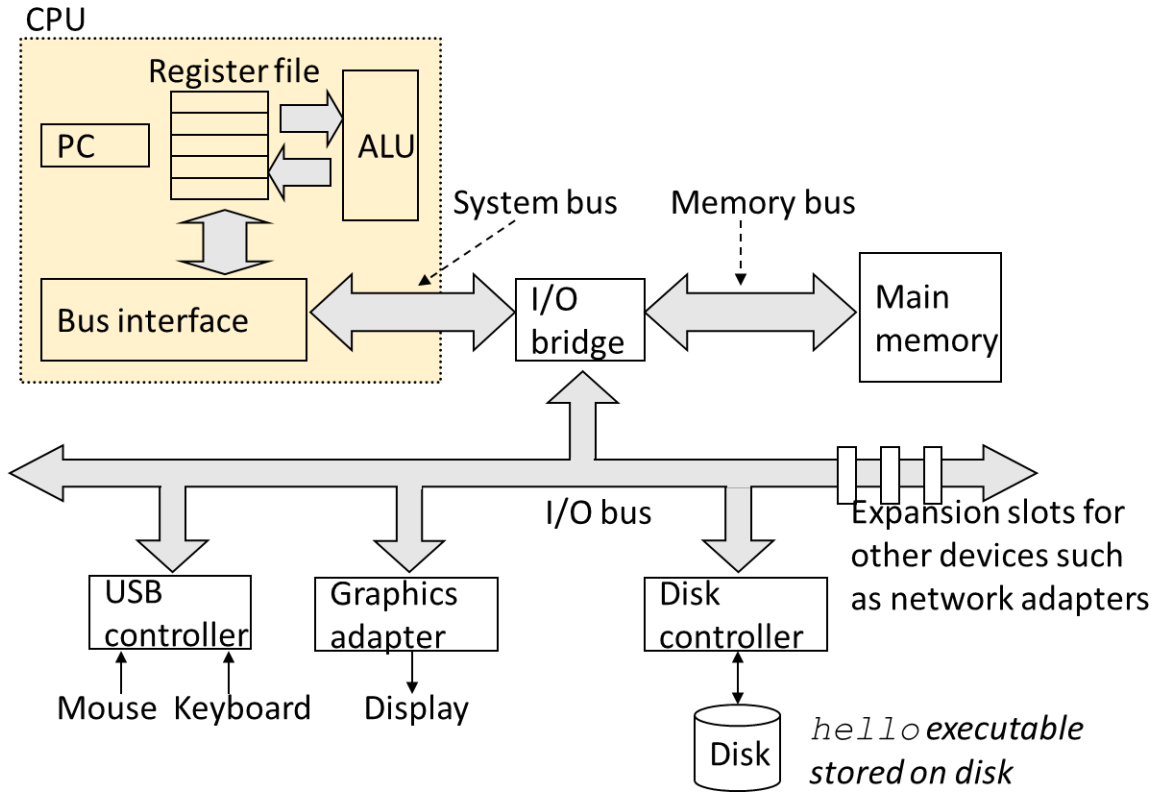
- Dispositivo de armazenamento temporário
- Armazena o programa e os dados a serem manipulados enquanto o processador executa o programa



- Fisicamente a memória principal é uma coleção de chips de memória de acesso randómico (DRAM, Dynamic Random Access Memory)
- Logicamente, a memória é organizada em um arranjo linear de bytes, cada um destes com seu próprio endereço exclusivo (index) iniciando em zero

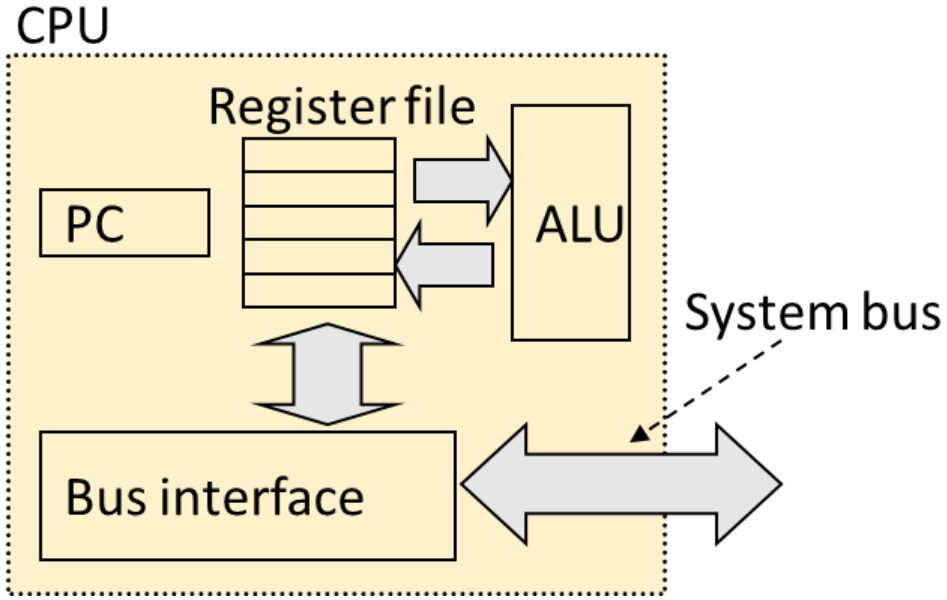
Processador

- Central processing unit (CPU)
- É uma máquina que interpreta (executa) instruções armazenadas na memória principal
- Um de seus componentes essências é uma registrador (dispositivo de armazenamento) chamado program counter (PC)



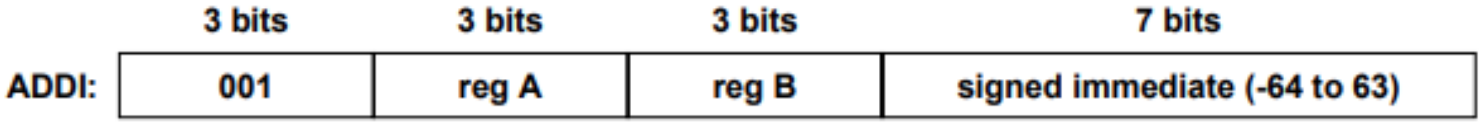
- O processador executa repetidamente a instrução apontada pelo PC
- E atualiza continuamente o contador de programa para apontar para a próxima instrução
- **Aparentemente** o processador opera segundo um modelo de instrução simples (uma instrução por vez)

Processador



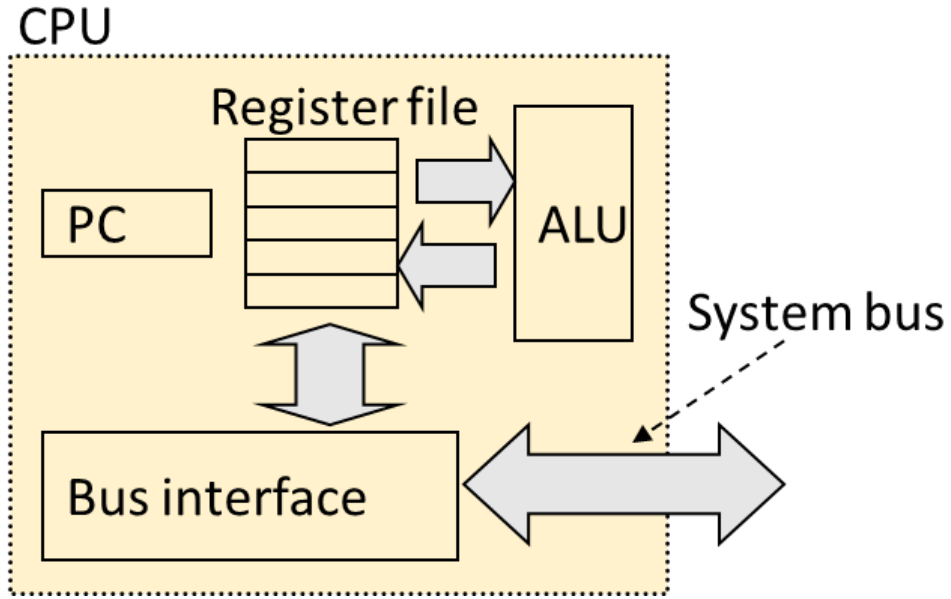
- A execução de uma instrução envolve algumas etapas
  1. O processador lê da memória principal a instrução apontada pelo PC

     0011 1101 0100 0101



  2. Interpreta os bits da instrução, o prefixo da instrução caracteriza o tipo e os demais bits são os operandos
  3. Executa alguma operação simples associada com a instrução
  4. Atualiza o PC para apontar para a próxima instrução
     - Pode ou não ser o endereço contíguo a instrução que acabou de ser executada

Processador



- Existem umas poucas instruções simples
- Elas envolvem a memória principal, o arquivo de registro e ALU (Unidade de Aritmética e Lógica)

- Componentes da CPU
  - Arquivo de registro
    - Pequeno dispositivo de armazenamento
    - Uma coleção de registradores com o tamanho de uma palavra
    - Cada registrador têm seu próprio identificador exclusivo
  - ALU calcula os novos dados e os novos endereços de memória

Processador

- Exemplos de operações (instruções do processador)
  - **Load**: Copia um byte o palavra da memória principal a um registrador, sobrescrevendo o conteúdo prévio do registrador
  - **Store**: Copia um byte o palavra de um registrador à um endereço da memória principal, sobrescrevendo o conteúdo prévio da memória
  - **Operate**: Copia o conteúdo de dois registradores na ALU, executa a operação aritmética envolvendo os dois dados, e armazena o resultado em um registro, sobrescrevendo o conteúdo prévio do registrador
  - **Jump**: Extrai uma palavra da própria instrução e copia esse valor no PC, sobrescrevendo o valor anterior do PC

Processador

- O processador pode ser abstraído como uma implementação de sua arquitetura do conjunto de instruções
- **Arquitetura do conjunto de instruções** = define o efeito de cada instrução do código de máquina (semântica)
- **Microarquitetura** = detalha como o processador é realmente implementado

- Processadores modernos usam mecanismos complexos para acelerar a execução do programa, executando múltiplas instruções "simultaneamente"