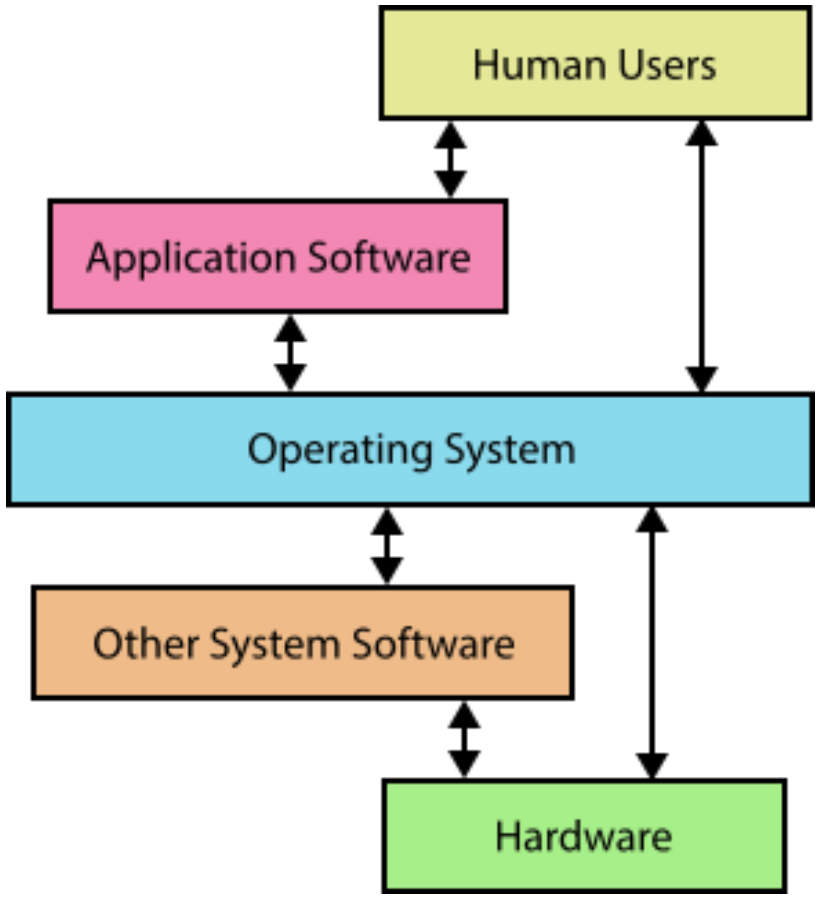
Software × Hardware

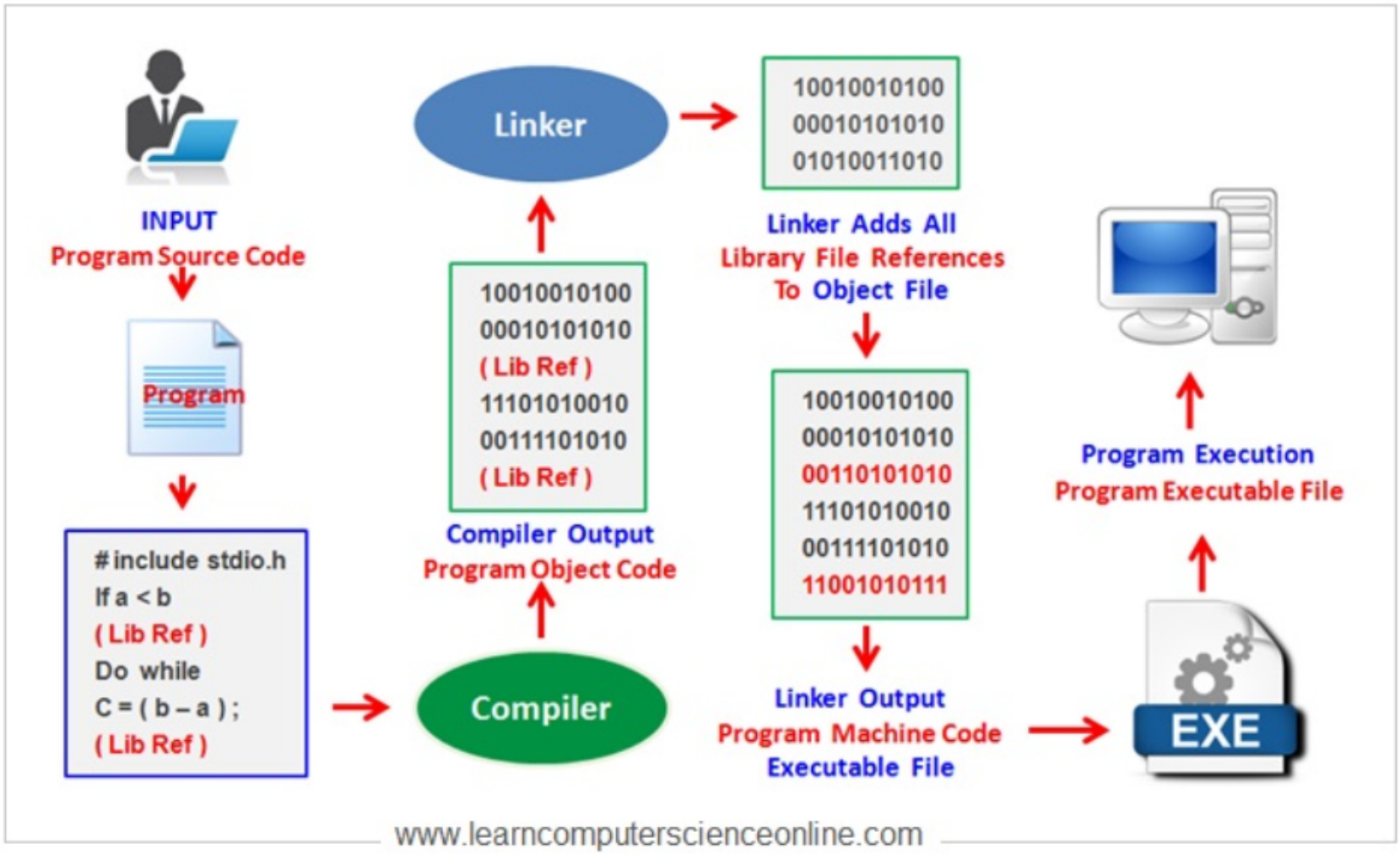


Programas escritos em linguagem de alto nível
(C, Java, Python, etc)

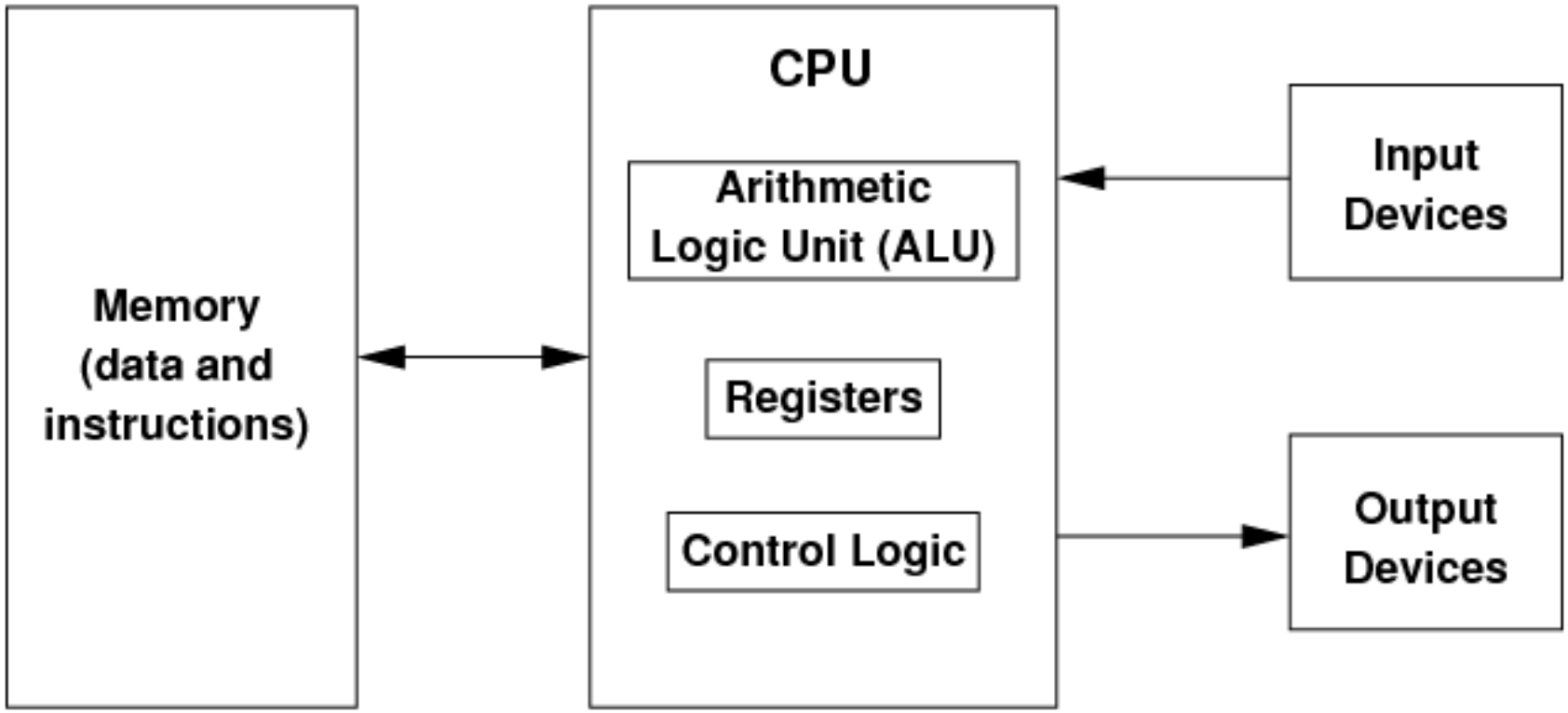
precisam ser convertidos para a linguagem de máquina

para serem "executados" pelo processador

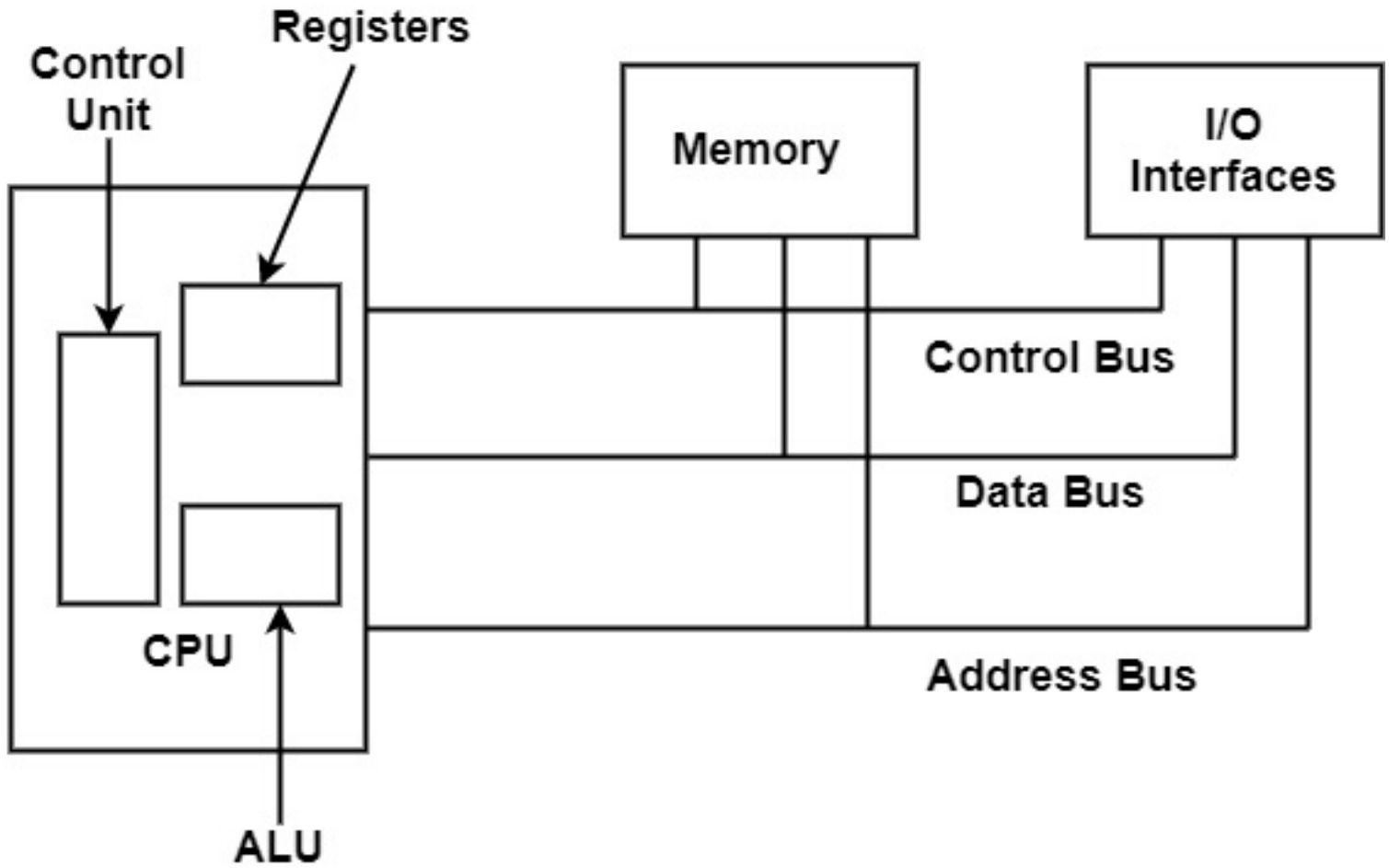
Computer Program Compilation Process

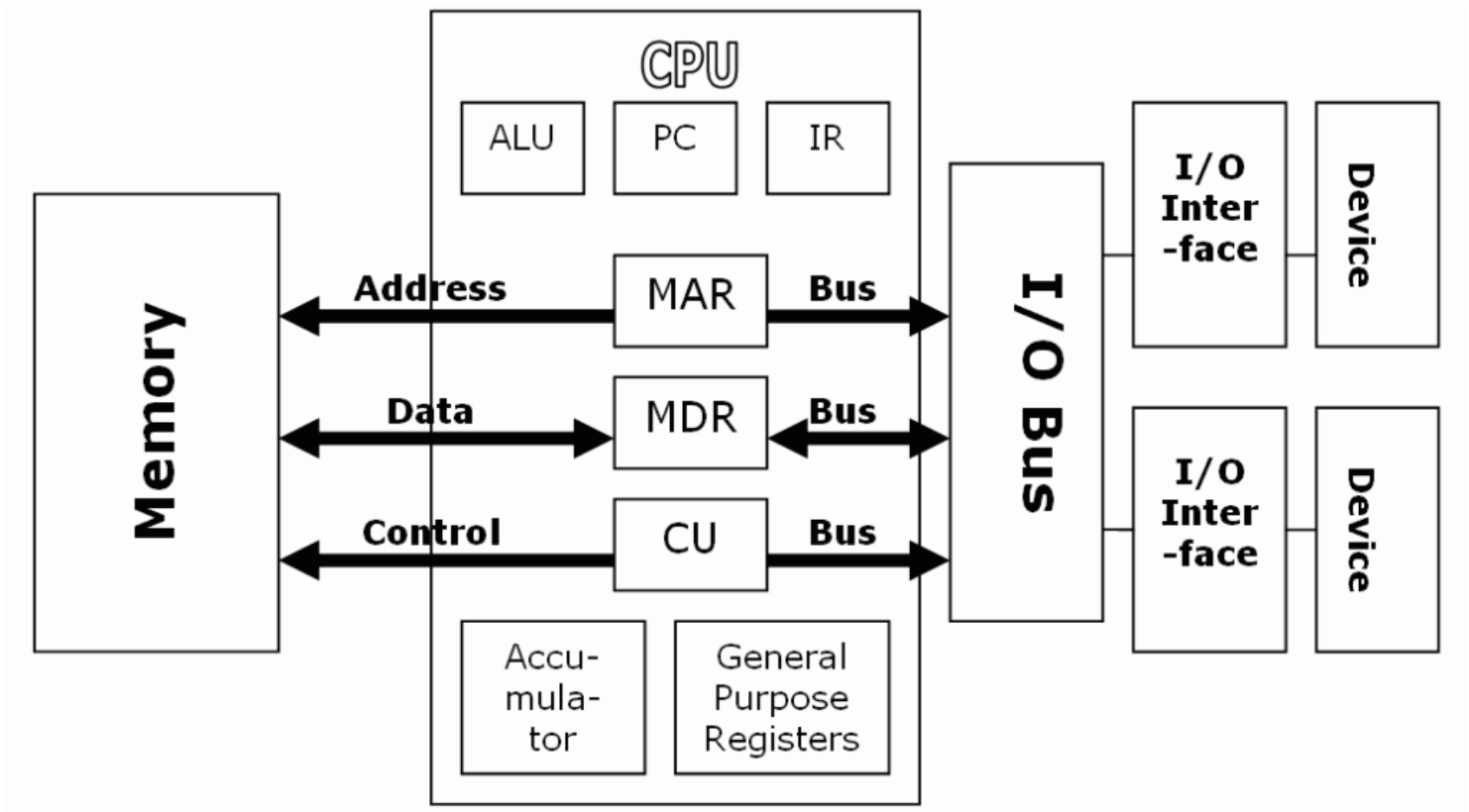


Modelo de Von-Neumann – a ideia de ter instruções (programa) e dados na memória



Modelo de Von-Neumann (outro diagrama) – destacando os componentes principais

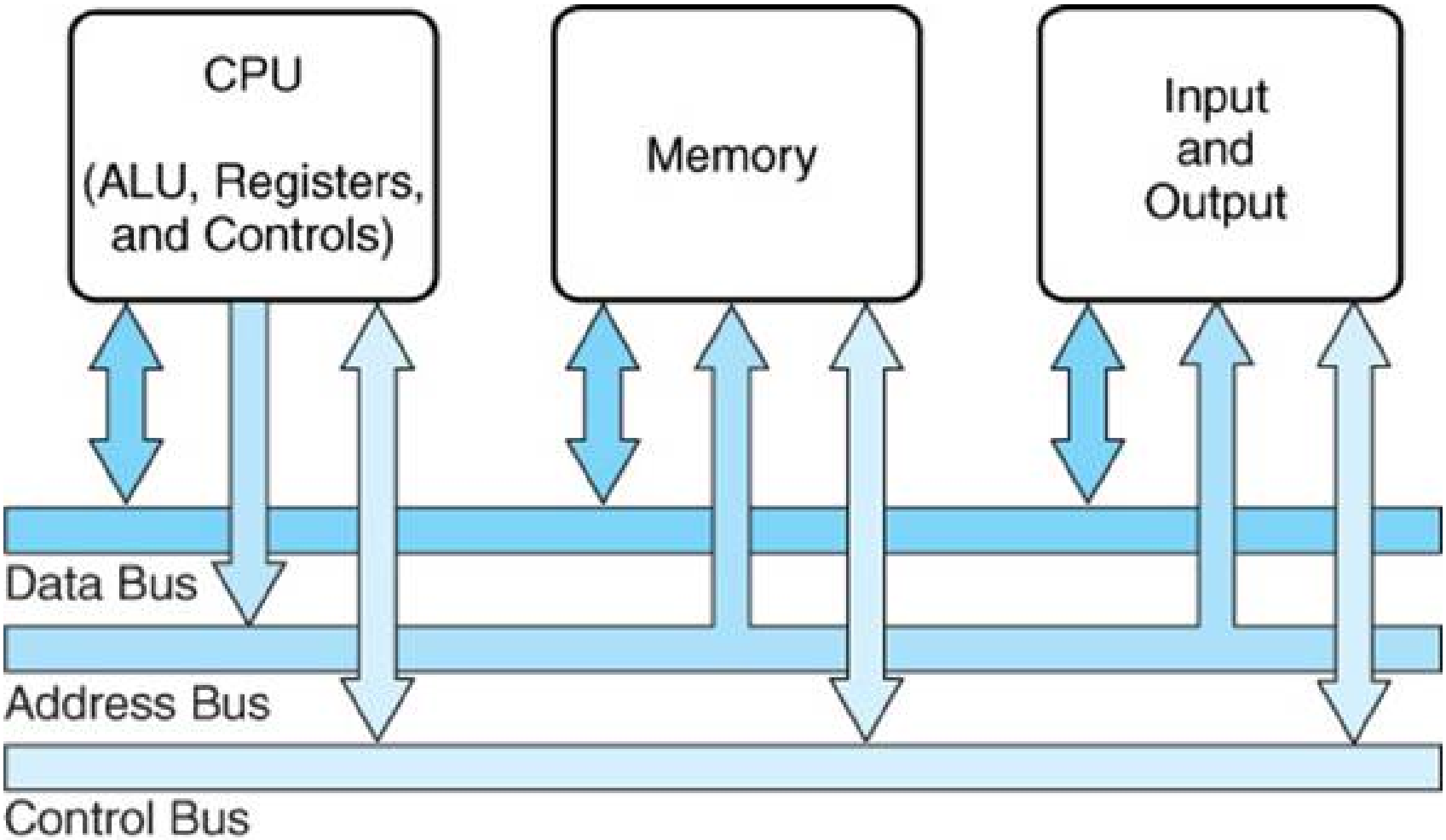


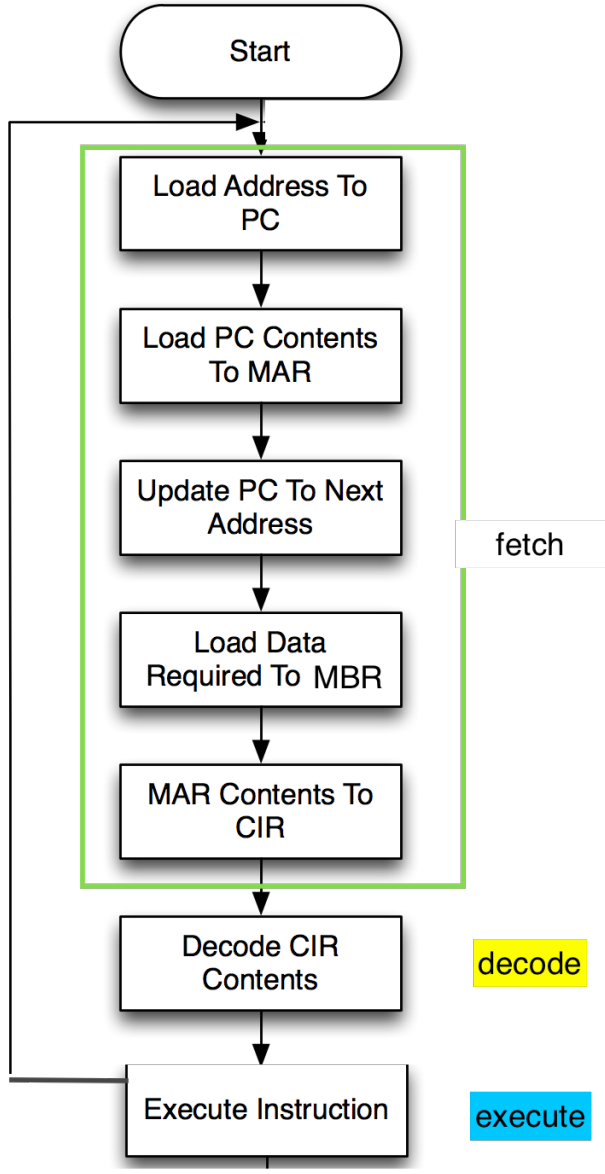


No nosso caso, não usaremos MAR (registrador para endereço) nem MDR (registrador para dados) da RAM

No nosso caso:

- Teremos RAM, ULA e três registradores (PC, IR e AC)
  PC: registrador-contador, aponta para a próxima instrução a ser executada
  IR: é para onde uma instrução que está na RAM é copiada, para em seguida alimentar o decodificador de instruções
  AC: é um registrador auxiliar usado em operações que envolvem a ULA
- A primeira instrução do programa a ser executado sempre estará na posição ZERO da RAM
- A unidade de controle é um circuito combinacional. Sua parte central é o decodificador de instruções que irá gerar os sinais de controle adequados para cada instrução a ser executada. Os sinais de controle devem garantir, por exemplo, que o modo de leitura/escrita da RAM está correto, que o sentido para o qual os dados fluem está correto, etc


CPU
(ALU, Registers, and Controls)
Memory
Input and Output
Data Bus
Address Bus
Control Bus


Start
Load Address To PC
Load PC Contents To MAR
Update PC To Next Address
Load Data Required To MBR
MAR Contents To CIR
Decode CIR Contents
Execute Instruction
fetch
decode
execute

Ciclo de instrução (FDX)

Consiste de 3 passos: FETCH, DECODE e EXECUTE

No nosso caso, o FETCH é mais simples do que está no slide anterior (já que não usamos os registradores MAR e MDR) para acesso à memória RAM)

Ciclo de instrução (FDX) – no nosso caso

1. Fetch: copiar o conteúdo na posição de memória apondata por PC para o IR e incrementar o valor de PC

   No nosso caso, como não usamos MAR e MDR, a instrução é copiada diretamente da RAM para o IR

2. Decode: preparar os sinais para a execução de acordo com a instrução em IR

   Este passo é realizado por um circuito combinacional; assim que a instrução é carregada no IR, a decodificação "acontece"

3. Execute: a instrução é executada

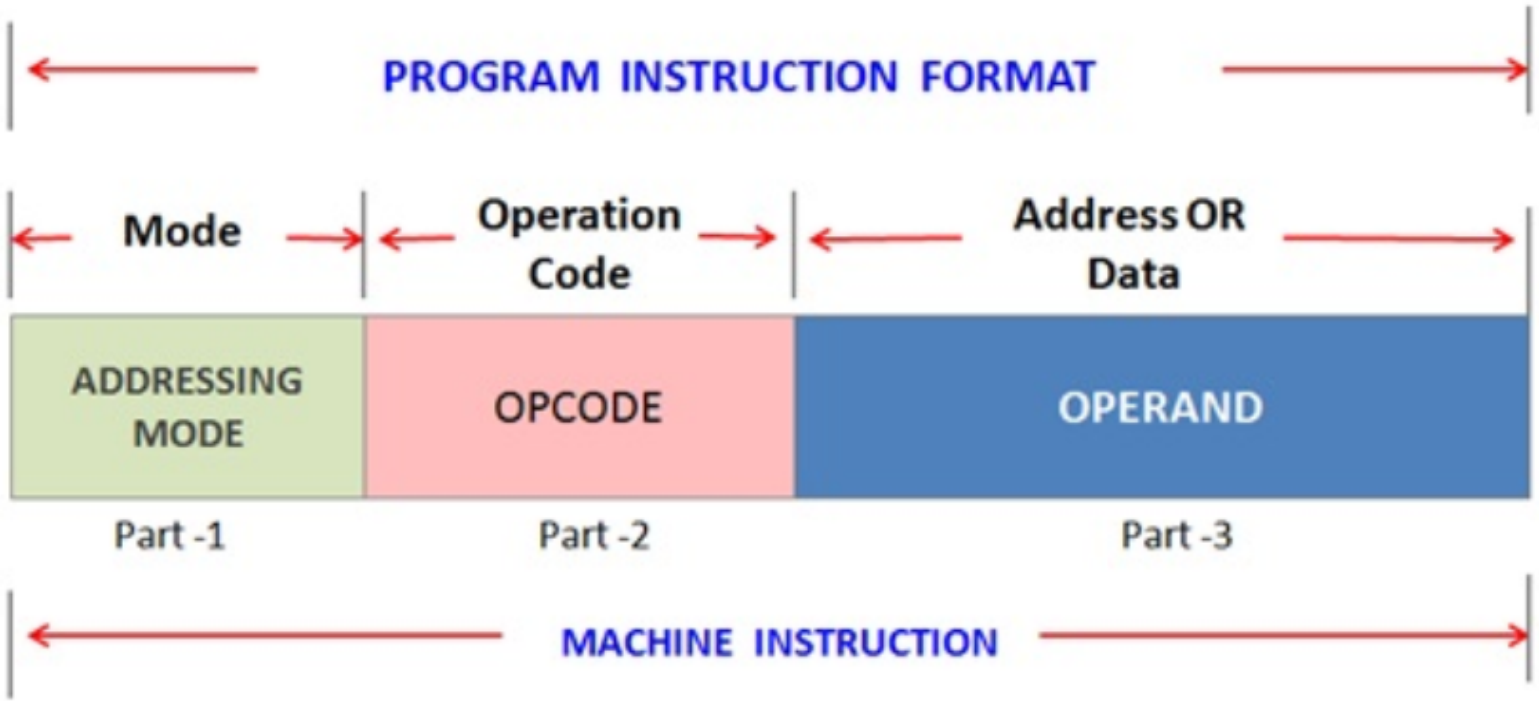
   Este passo em geral envolve alteração do valor na RAM ou de algum registrador

   Após a execução, o ciclo deve ser resetado (ficar em estado pronto para o ciclo referente à próxima instrução)

Os passos 1 e 3 requerem um pulso de clock; o passo 2, não.

Logo, NO NOSSO MODELO SIMPLICADO, dois pulsos de clock são suficientes para um ciclo de instrução

Estrutura geral de uma instrução simples



PART 1 – Addressing Mode - Rule For Operand - Data Or Address

PART 2 – OPCODE - For Control Unit - Which Operation To Perform.

PART 3 – OPERAND - For ALU - On Data Operation To Be Performed.

www.learncomputerscienceonline.com

No nosso caso, uma instrução consiste de 16 bits

8 bits mais significativos: código da instrução

8 bits menos significativos: endereço de memória, se pertinente

Instruções de máquina do nosso processador

Código	Descrição da instrução
00	NOP (no operation)
01	Copie [EE] para o AC
02	Copie [AC] para a posição de endereço EE
03	Some [EE] com [AC] e guarde o resultado em AC
04	Subtraia [EE] de [AC] e guarde o resultado em AC
07	Leia um número e guarde-o na posição de endereço EE
08	Imprima [EE]
09	Pare
0A	Desvie para EE (desvio incondicional)
0B	Desvie para EE se [AC] $>$ 0
0D	Desvie para EE se [AC] $=$ 0
0F	Desvie para EE se [AC] $<$ 0

Exemplo de um programa

```
00: 0110 -- Copie [10] para o AC
01: 0211 -- Copie [AC] para a posição de endereço 11
02: 0811 -- Imprima [11]
03: 0712 -- Leia um número e guarde-o na posição de endereço 12
04: 0112 -- Copie [12] para o AC
05: 0D0A -- Desvie para 0A se [AC] = 0
06: 0311 -- Some [11] com [AC] e guarde o resultado em AC
07: 0211 -- Copie [AC] para a posição de endereço 11
08: 0811 -- Imprima [11]
09: 0A03 -- Desvie para 03
0A: 0900 -- Pare
    ...
0F: ...
10: 0000 -- Zero
11:      -- Soma
12:      -- Num
```

```
Zero = 0
Soma = Zero
print(Soma)
Num = int(input())
while Num!=0 :
    Soma = Soma + Num
    print(Soma)
    Num = int(input())
```

Em linguagem de máquina, tanto o `if/else` como o `while` envolvem operações de DESVIO

No `while`, quando a condição não é satisfeita, a execução do programa é desviada para a instrução (comando) que vem logo depois do final do laço. Por outro lado, caso as instruções do laço sejam executadas, ao final do laço ocorre necessariamente um desvio para o início do laço, para verificar a condição novamente.

Esses desvios, no slide anterior, correspondem às instruções na posição 05 e 09, respectivamente.

Suponha que a próxima instrução a ser executada é

Copie o conteúdo do endereço 10 para AC

A instrução é 0110
(01 é o código da instrução e 10 é o endereço referenciado)

Suponha que ela está na posição 02 na RAM

(preencha) Antes de iniciar o ciclo de instrução:

- o valor do PC é
- à porta de endereço da RAM está chegando o conteúdo do
- a RAM encontra-se em modo de
- a saída da RAM está sendo direcionada para o

Suponha que a próxima instrução a ser executada é

Copie o conteúdo do endereço 10 para AC

A instrução é 0110
(01 é o código da instrução e 10 é o endereço referenciado)

Suponha que ela está na posição 02 na RAM

(preencha) Antes de iniciar o ciclo de instrução:

- o valor do PC é 02
- à porta de endereço da RAM está chegando o conteúdo do PC
- a RAM encontra-se em modo de "Leitura"
- a saída da RAM está sendo direcionada para o IR

Ciclo de instrução (FDX) – o que acontece a cada passo?

Antes do FETCH, o estado está como descrito no slide anterior

FETCH: com um pulso do clock,

- a saída da RAM (que é o conteúdo na posição 02 da RAM, ou seja, a instrução 0110, é "carregada" no IR)

- o PC é incrementado em 1

Ciclo de instrução (FDX) – o que acontece a cada passo?

Após o FETCH, o IR contém a instrução a ser executada

DECODE: o circuito combinacional

- tem como entrada os 8 bits mais significativos do IR (código da instrução)

- acerta alguns sinais de controle

— direciona os 8 bits menos significativos (endereço referenciado pela instrução) para a porta de endereço da RAM (note que precisamos de um mecanismo para selecionar endereço do PC ou da instrução para alimentar a porta de endereço da RAM)

— o modo de operação da RAM continua em "Leitura"

— o dado da porta de dados da RAM deve ser direcionado para o AC (note que aqui precisamos de um mecanismo para direcionar o dado da saída da RAM ou para o IR ou para o AC)

Ciclo de instrução (FDX) – o que acontece a cada passo?

Após o DECODE, o circuito está preparado para executar a instrução

EXECUTE: com um pulso do clock,

- o dado na porta de dados da RAM está chegando em AC e será carregado nele

- o estado do circuito deve voltar ao de início de ciclo

— acertar o endereço enviado para a porta de endereços da RAM

— colocar o modo de operação da RAM para leitura

— direcionar a saída de dados da RAM para o IR

Código assembly – exemplo

prog1.c

```
int main() {
  int x=5;
  int y=7;

  return 0;
}
```

```
===> Para compilar prog1.c, usando o gcc:

   gcc prog1.c -o prog1

O gcc gerará o executável prog1 (se não ocorrer erro de compilação)


===> Para gerar o código assembly, usando o gcc:

   gcc -S prog1.c

O gcc gerará o arquivo prog1.s, contendo o código
assembly correspondente ao programa em prog1.c
```

Código assembly

```
        .file   "prog1.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $5, -8(%rbp)
        movl    $7, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
        .section        .note.GNU-stack,"",@progbits
```

Código assembly “enxuto”

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $5, -8(%rbp)
        movl    $7, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        ret
```

```
%rbp  é o registrador de base da pilha de execução
      (base point)
%rsp  é o registrador de topo da pilha de execução
      (stack point)
%eax  é um registrador de dados genérico
      (no caso, usado para armazenar o valor
       retornado pela função)
```