# EA075 - Introdução ao projeto de sistemas embarcados (1S/2016) ::

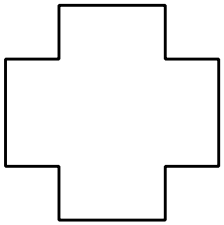
# Processadores de Propósito Geral

Prof. Christian Esteve Rothenberg
chesteve@dca.fee.unicamp.br

# Objetivos do capítulo

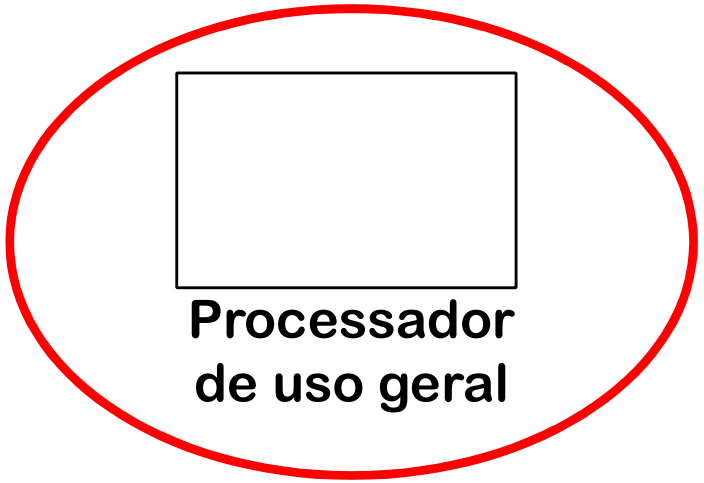
- Processador de uso geral
  - Arquitetura básica (revisão)
  - Operação (revisão)
- Visão do Programador
- Ambiente de Desenvolvimento
- Projeto de processadores de uso geral
- ASIPs
- Escolhendo um Microprocessador

# Tecnologia de processadores



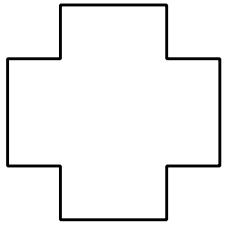
**Funcionalidade desejada**

```
total = 0
for i = 1 to N
loop
     total +=M[i]
end loop
```



**Processador de uso geral**

**Processador de aplicação específica**



**Processador dedicado**

Figura extraída de (Vahid, 2002)

# Introdução

- **Processador genérico:** sistema digital programável projetado para resolver tarefas de computação em uma ampla gama de aplicações.

- Exemplos:
  - ARM 7
  - Motorola 68HC05
  - Intel 8051
  - 8086

# Motivação

Um projetista de sistemas embarcados pode escolher utilizar um processador genérico para implementar parte de uma funcionalidade desejada do sistema e, com isso, obter alguns benefícios:

- O custo (de aquisição) por unidade do processador pode ser baixo – o NRE foi amortizado pela grande quantidade (milhões ou até mesmo bilhões) de unidades vendidas.
- O fabricante pode investir um alto capital em NRE durante a montagem do processador sem que isto aumente de forma significativa o custo da unidade – logo, pode recorrer a tecnologias mais avançadas de IC (e.g., VLSI *layouts*) para componentes críticos.
  - Por isso, processadores genéricos podem oferecer bom desempenho, bem como tamanho e consumo de potência aceitáveis.
- O custo NRE do projetista é relativamente baixo: basta preparar um *software* e utilizar compiladores / montadores adequados.
- Tempo de prototipagem e tempo para o mercado são relativamente baixos.
- Alta flexibilidade.

# Arquitetura básica

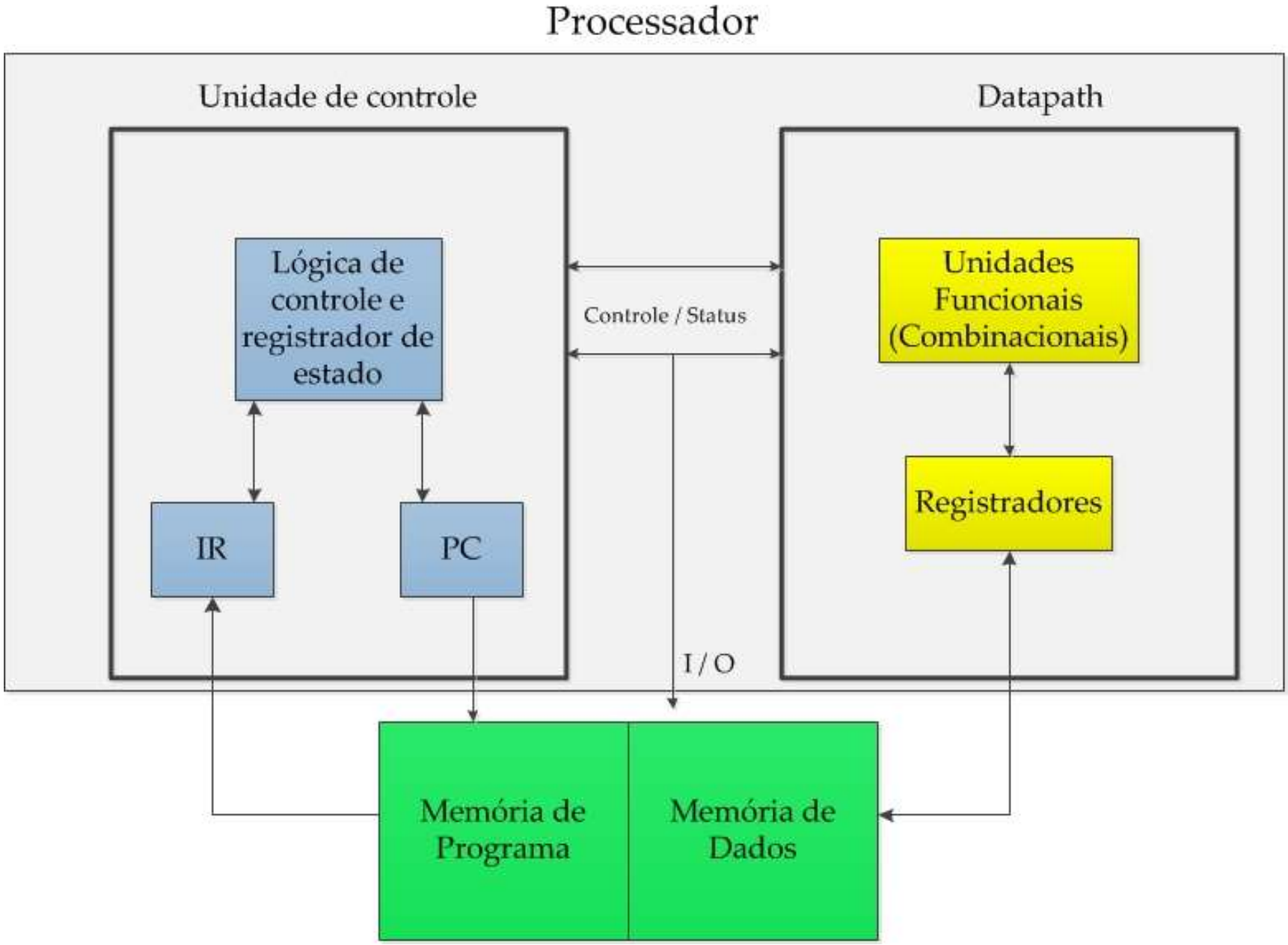
- Processador de propósito geral (CPU, *central processing unit*):
  - *Datapath*
  - Unidade de controle
  - Memória

Semelhante ao processador dedicado, exceto:

(1) pelo fato de o *datapath* ser genérico, oferecendo uma coleção de operações gerais sobre dados;

(2) por ter uma unidade de controle que não realiza uma sequência pré-definida de comandos (precisa ler as instruções armazenadas em uma memória).

# Arquitetura básica



# Arquitetura básica

- ***Datapath*:**

  - Unidade lógico-aritmética (ULA) – oferece um conjunto de transformações sobre os dados, como adição, subtração, AND, OR, inversão e deslocamento.

  - Gera sinais de *status* que indicam condições particulares referentes às operações executadas (por exemplo, estouro aritmético (*overflow*), adição que gera um vai-1 (*carry*)).

  - Registradores para armazenamento temporário de dados.

  - Operação *load*: transfere o conteúdo de uma posição de memória para um registrador.

  - Operação *store*: transfere o conteúdo de um registrador para uma posição de memória.

  - Define o tamanho do processador (regist. de *N* bits, operações executadas sobre operandos de *N* bits, barramentos, interfaces de dados).

# Arquitetura básica

- **Unidade de controle:**

  - Circuito que sequencia a execução de instruções de programa, sendo responsável por mover os dados de, para e através do *datapath* de acordo com estas instruções.

  - **Registrador PC:** contém o endereço da próxima instrução a ser lida.
    - O controlador ajusta o valor do PC para sempre apontar para a próxima instrução. No caso de um desvio ou de uma ramificação, sinais de status do *datapath* podem nortear a definição do próximo valor de PC.
    - Seu tamanho determina o espaço de endereçamento do processador: por exemplo, se PC tem 16 bits, existem 65536 posições de memória endereçáveis.

  - **Registrador IR:** contém a instrução lida.

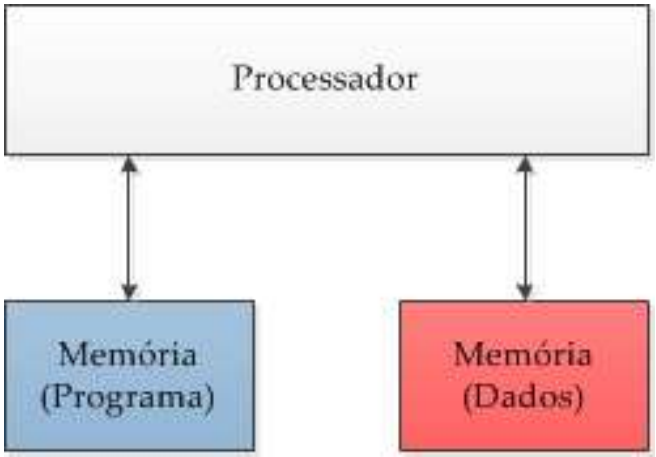
  - Cada instrução exige que o controlador passe por vários estágios, sendo que cada estágio pode durar um ou mais ciclos de relógio.

  - A frequência do processador dá uma noção de sua velocidade.

# Arquitetura básica

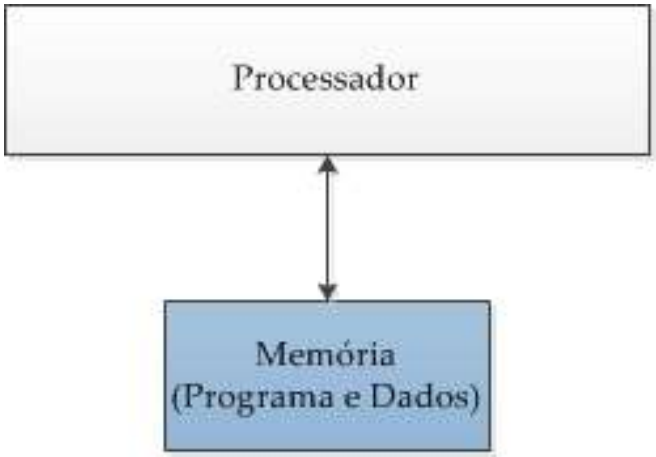
- **Memória:**
  - Armazenamento de dados e instruções para médio e longo prazos.
  - **Duas arquiteturas:**

Harvard




Leitura simultânea de dados e de instruções

Princeton




Menor quantidade de conexões

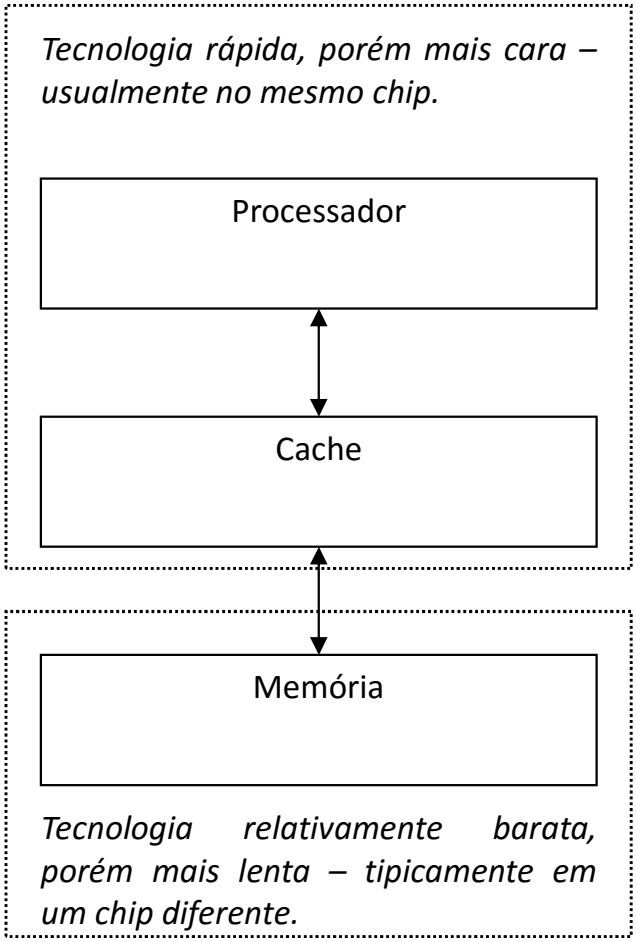
# Arquitetura básica

- **Memória:**
  - Existem diferentes tipos de memória, por exemplo, ROM e RAM.
  - On-chip: a memória está no mesmo IC que o processador.
    - Acesso mais rápido, porém com maiores limitações de capacidade (tamanho).
  - Off-chip: memória está em um IC separado.

# Arquitetura básica

- **Memória:**

Para reduzir o tempo de acesso à memória, uma cópia local (no mesmo chip do processador) de parte da memória é mantida em um pequeno, mas especialmente rápido, dispositivo chamado de *cache*.



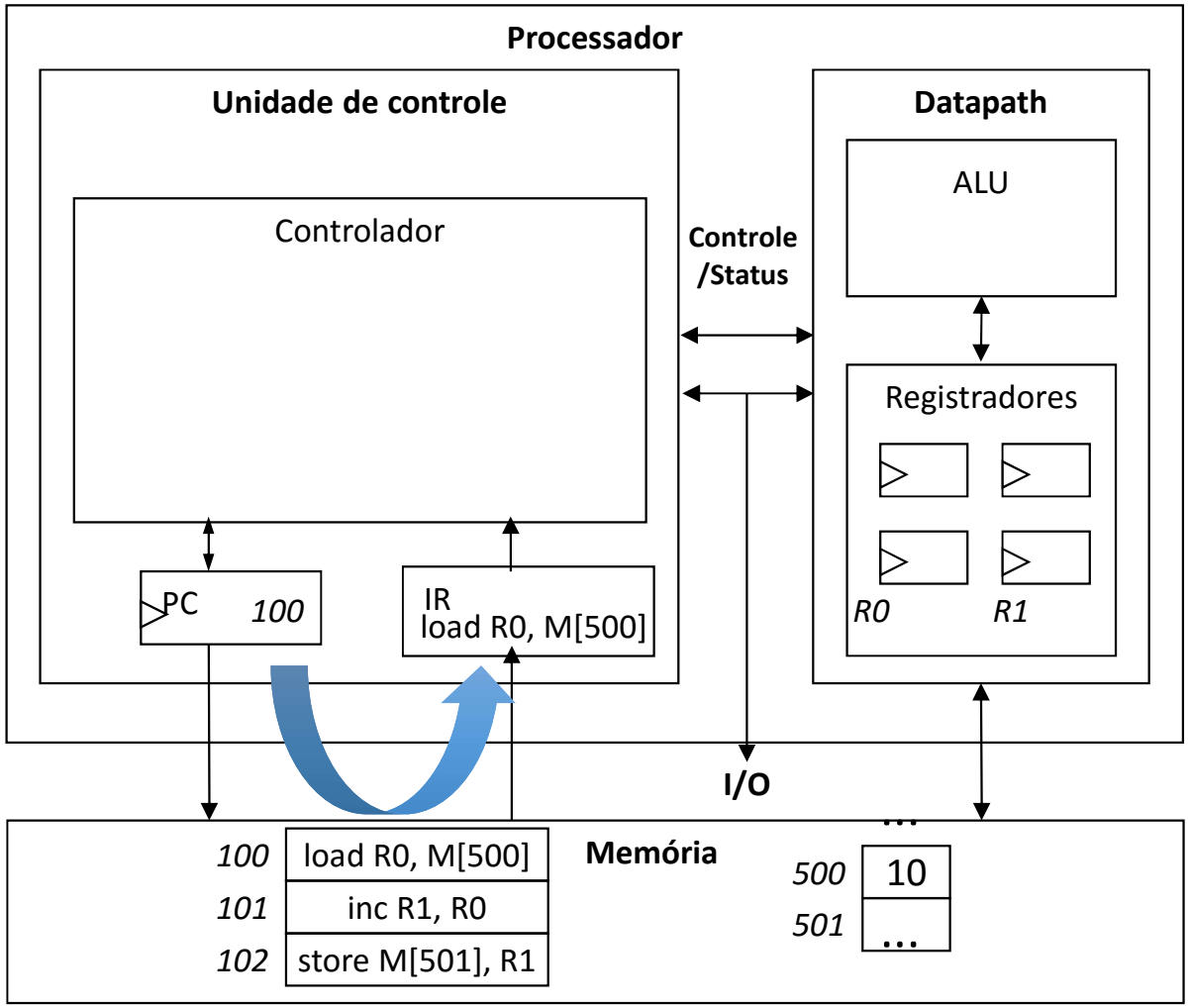
# Operação

- **Execução de uma instrução**

  - Busca (*fetch*) de instrução.
  - Decodificação.
  - Busca de operandos.
  - Execução da operação.
  - Armazenamento de resultados.

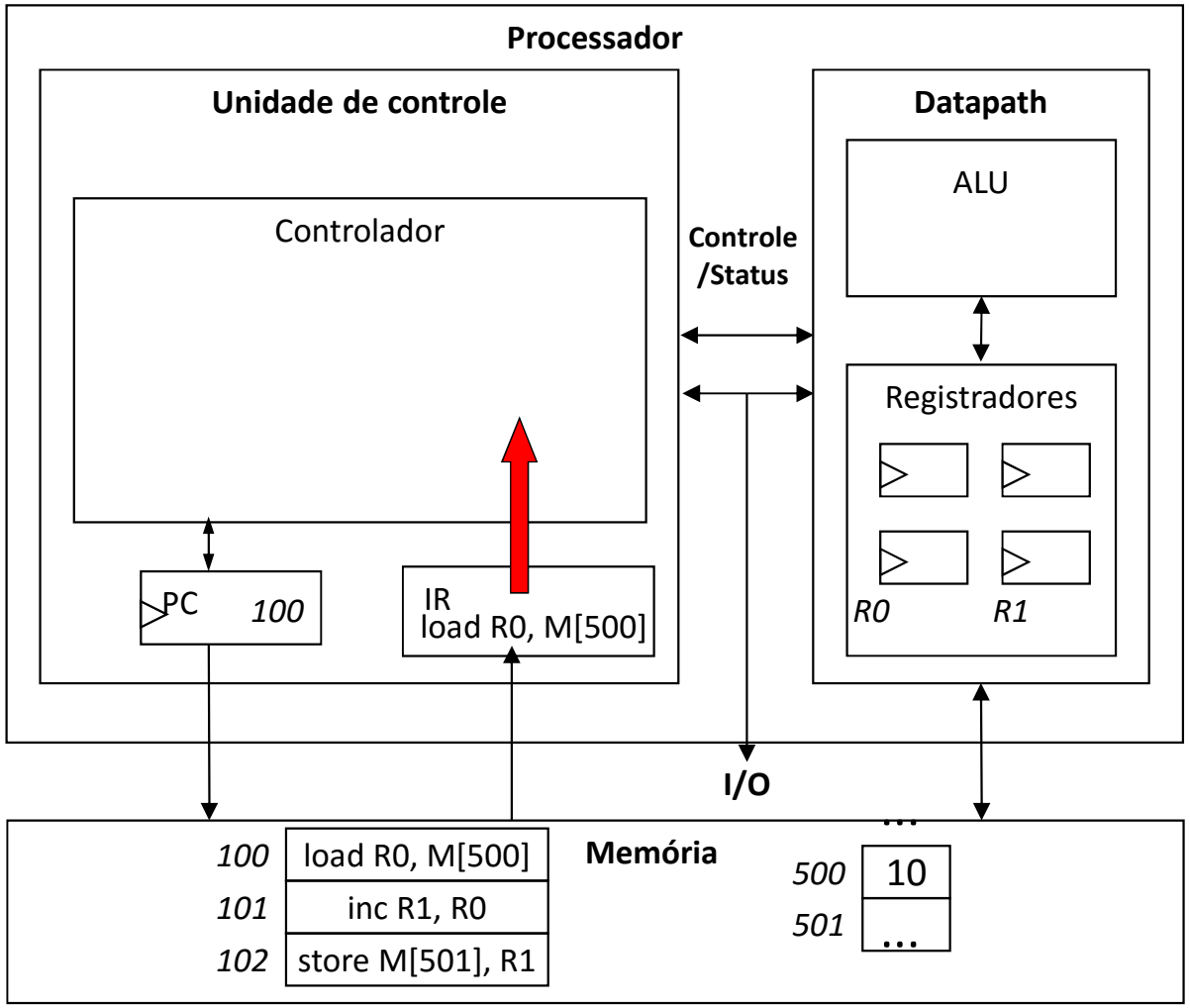
# Operação

- Busca (*fetch*) de instrução – leitura da próxima instrução da memória para o IR.



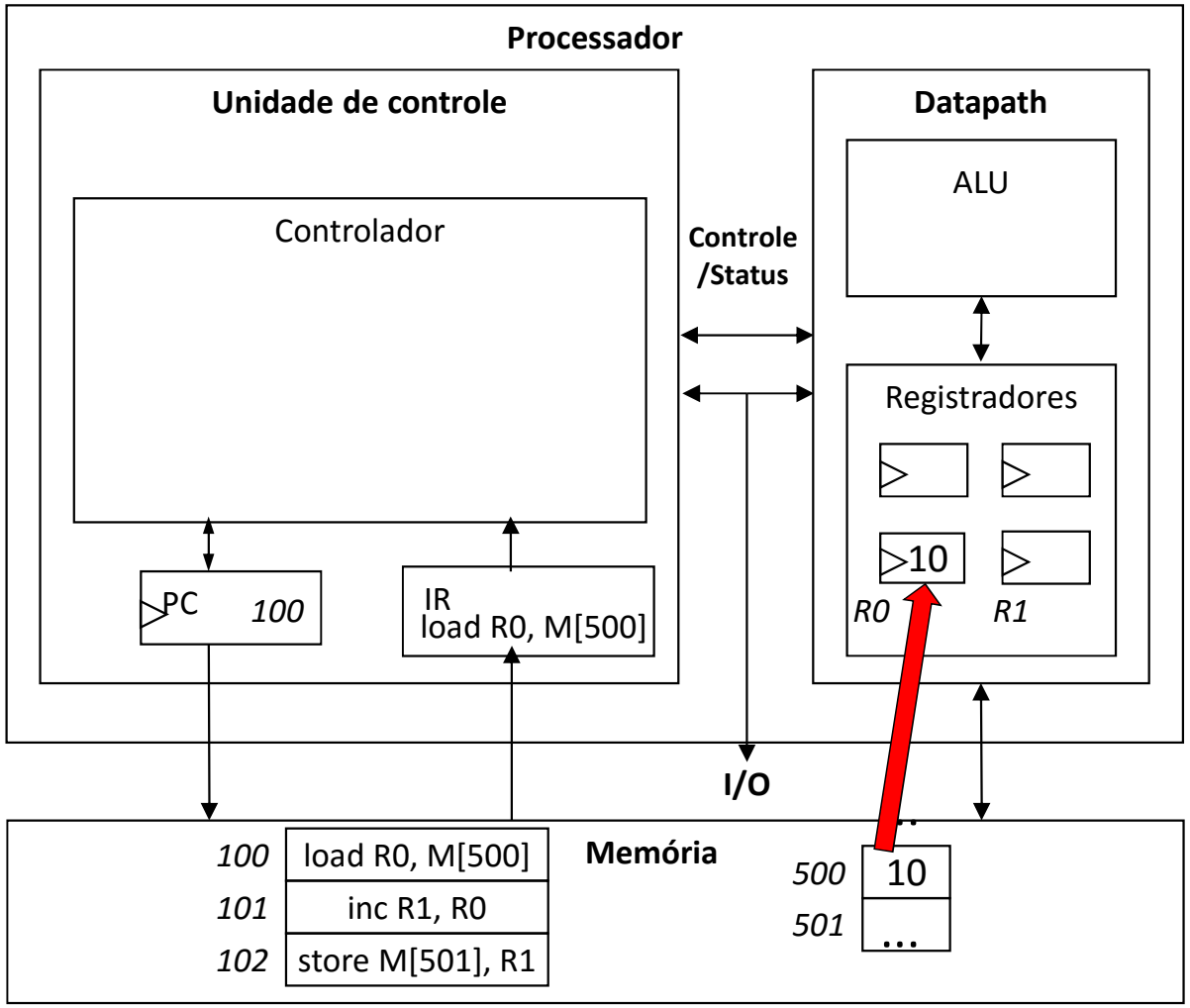
# Operação

- Decodificação – identificação de qual operação está especificada na instrução em IR.



# Operação

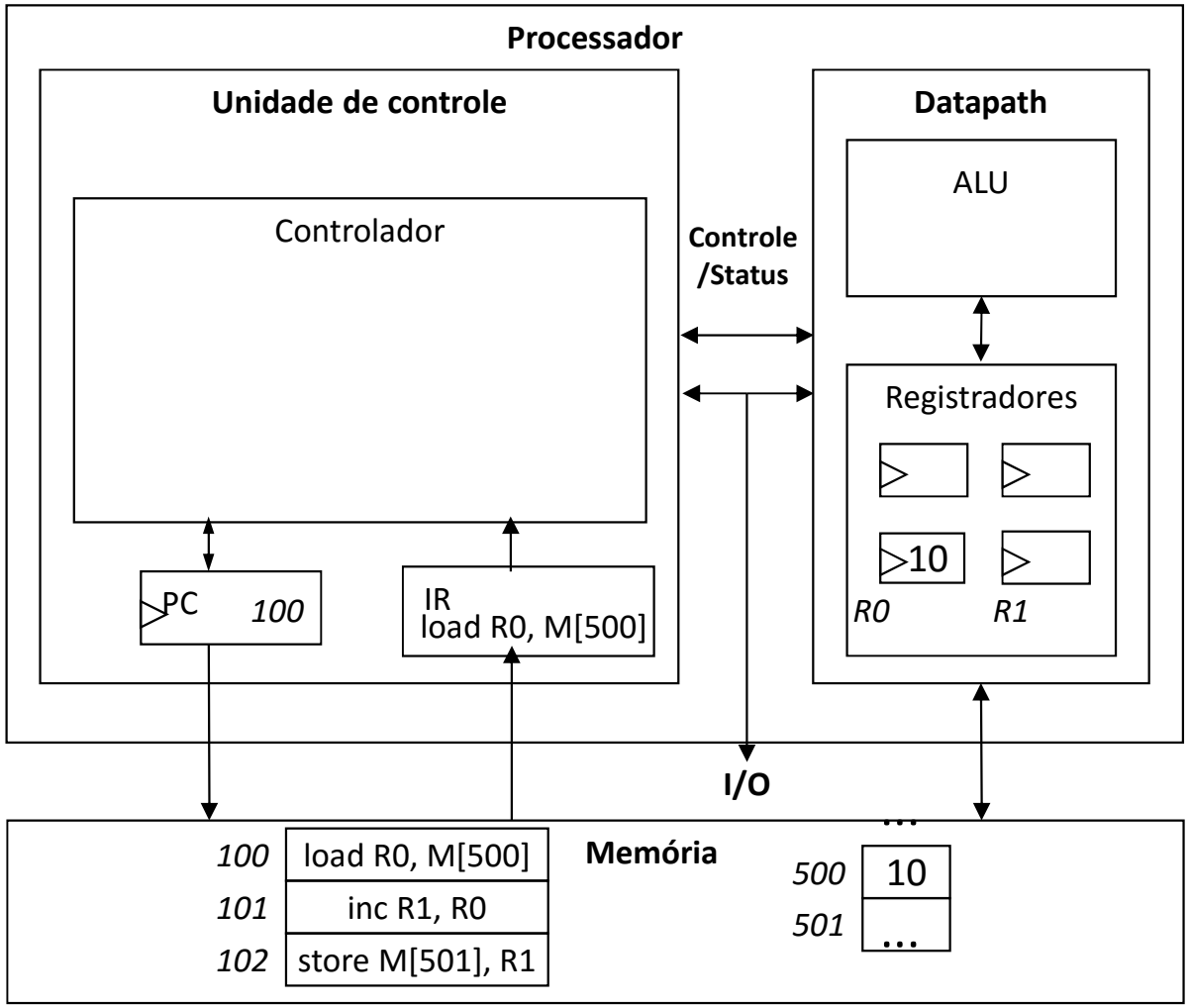
- Busca de operandos – movimentação dos operandos da instrução para os registradores apropriados.



# Operação

- Execução – alimenta os componentes apropriados da ALU que realizarão a operação desejada.

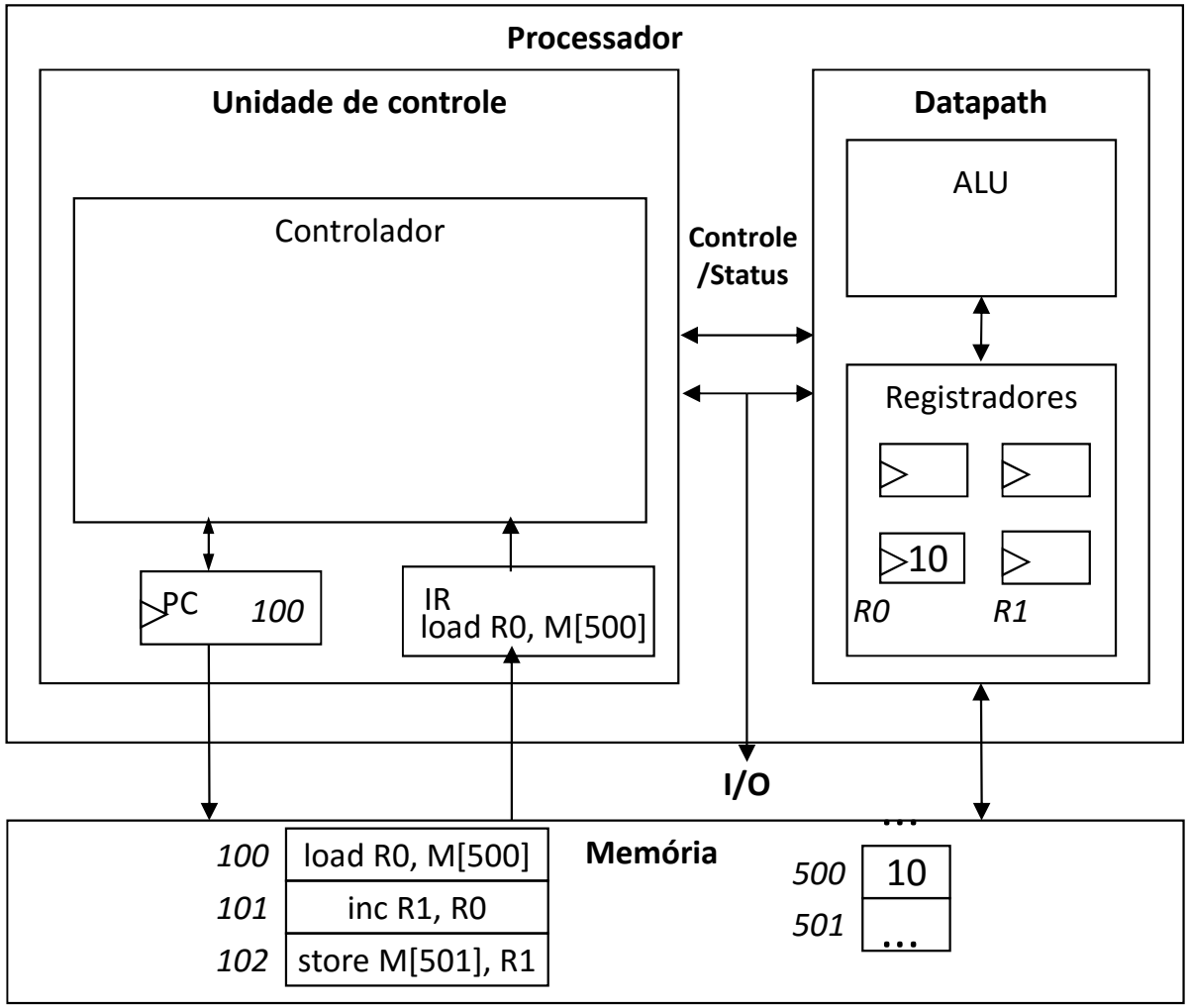
No caso da instrução *load*, não há operação que a ALU precisa realizar.



# Operação

- Armazenar resultados – escrita de resultados armazenados em registradores de volta à memória.

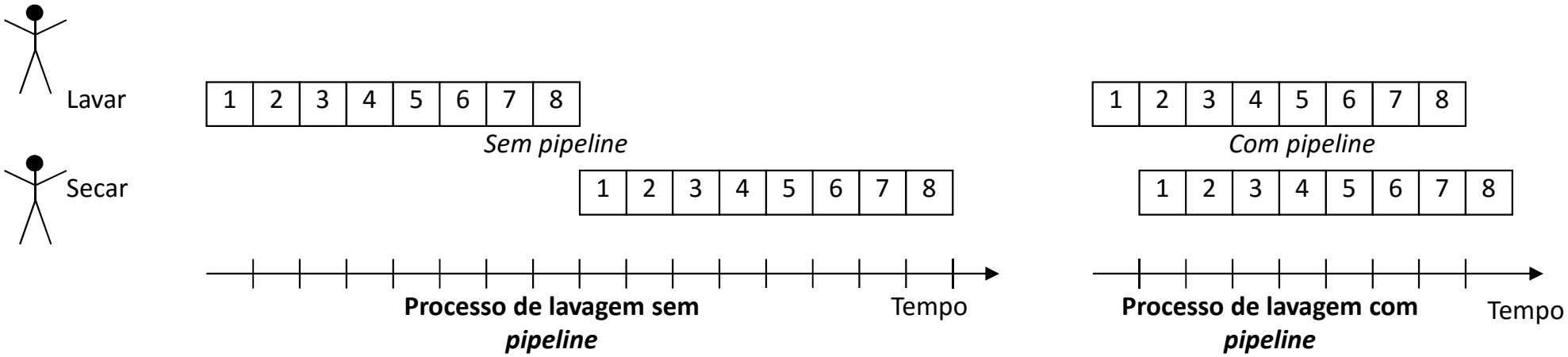
No caso da instrução *load*, não há esta etapa.



# Operação

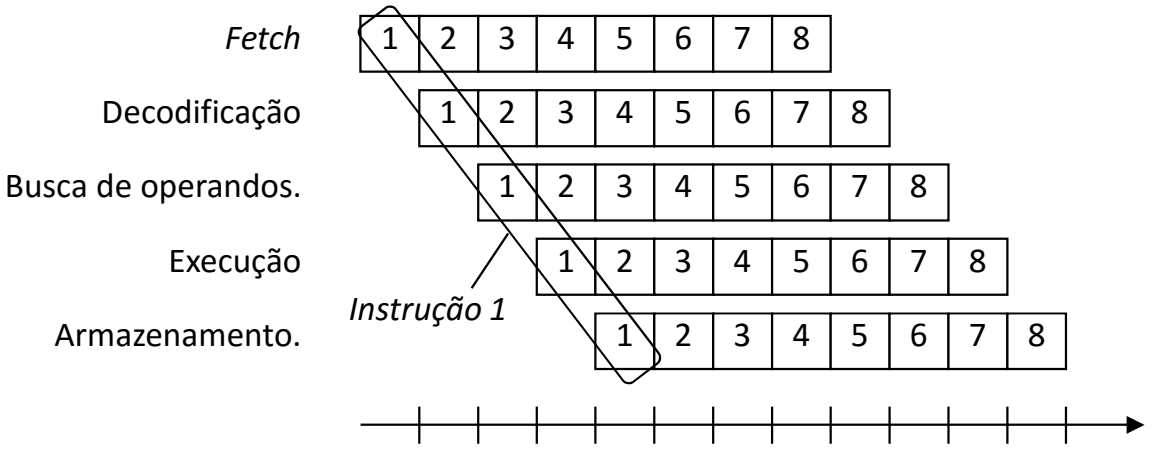
- ***Pipeline*:** ao utilizar uma unidade separada para cada estágio de instrução, é possível ter múltiplas instruções de máquina em processamento ao mesmo tempo.

- Ilustração: lavar louças.



# Operação

- Exemplo: depois que a unidade de busca de instrução (*fetch*) faz a leitura da primeira instrução, a unidade de decodificação é acionada para interpretá-la, enquanto a unidade de busca pode carregar a próxima instrução em IR.




**Execução de instruções em *pipeline*.**

# Introdução

- **Comparação preliminar:** uniciclo vs. *pipeline*
  - 8 instruções – *lw*, *sw*, *add*, *sub*, *and*, *or*, *slt* e *beq*.
  - Tempo de operação:
    - 200ps para acesso à memória;
    - 200ps para operação da ALU;
    - 100ps para leitura ou escrita no arquivo de registradores.
  - Tempo de execução das instruções:

| Instruction class | Instruction fetch | Register read | ALU operation | Data access | Register write | Total time |
|---|---|---|---|---|---|---|
| Load word (`lw`) | 200 ps | 100 ps | 200 ps | 200 ps | 100 ps | 800 ps |
| Store word (`sw`) | 200 ps | 100 ps | 200 ps | 200 ps | | 700 ps |
| R-format (`add`, `sub`, `AND`, `OR`, `slt`) | 200 ps | 100 ps | 200 ps | | 100 ps | 600 ps |
| Branch (`beq`) | 200 ps | 100 ps | 200 ps | | | 500 ps |

# Introdução

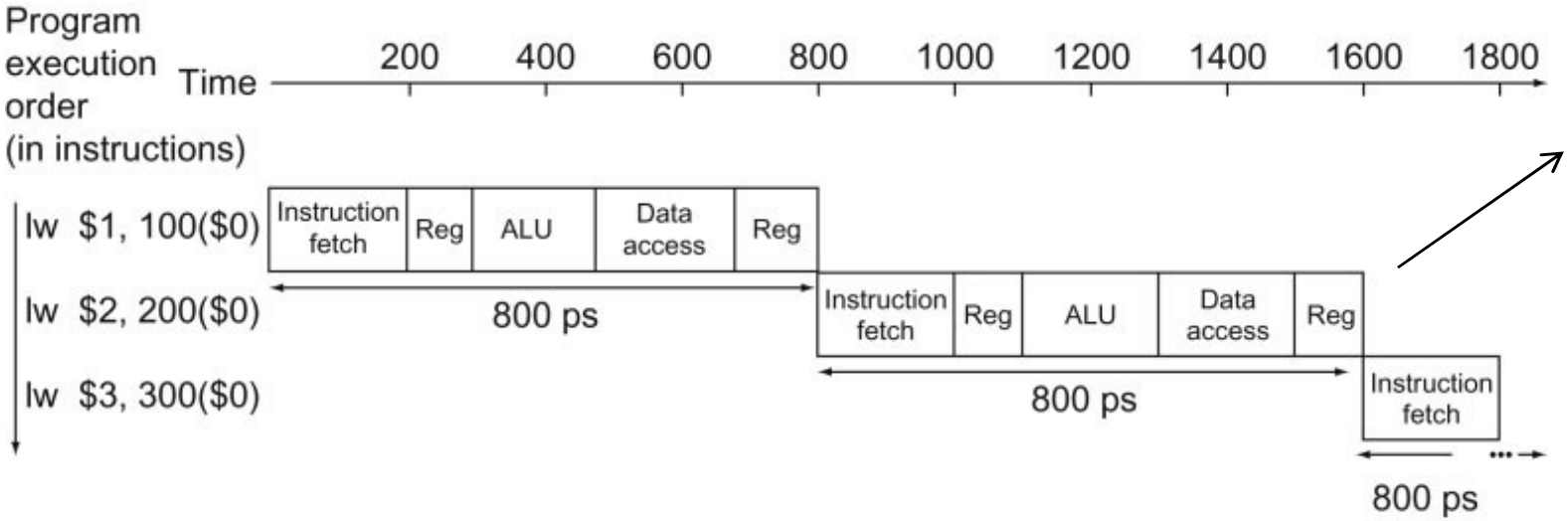
- **Comparação preliminar:** uniciclo vs. *pipeline*

  - **Uniciclo:** o mínimo ciclo de relógio será 800ps.

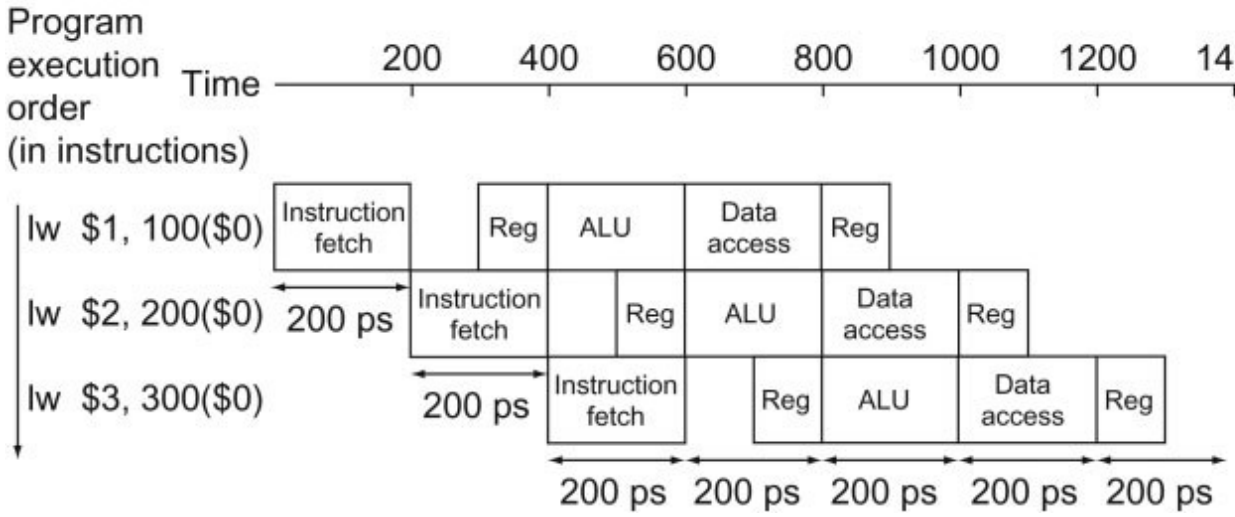
  - ***Pipeline*:** todos os estágios da *pipeline* consomem um ciclo de relógio.
    - Logo, o período do relógio deve ser longo o suficiente para acomodar a operação mais longa – 200ps.

# Introdução

- **Comparação preliminar:** uniciclo vs. *pipeline*




O tempo entre a primeira instrução *load* e a quarta será de 3 x 800 ps = 2400 ps




Com *pipeline*, a quarta instrução começará a ser executada 600 ps após a primeira.

# Operação

- **Cuidados – *pipeline*:**

  - As instruções devem ser passíveis de decomposição em estágios de aproximadamente o mesmo tamanho.

  - As instruções devem tomar o mesmo número de ciclos de relógio.

  - Instruções de desvio / ramificação são um problema para *pipelines*, pois não se sabe qual será a próxima instrução até que a atual atinja o estágio de execução.

# *Hazards*

- Há situações especiais em que a próxima instrução do programa não pode ser executada no ciclo seguinte. Estes eventos são chamados de ***hazards***.

- Veremos três tipos:
  A. **Estruturais**
  B. **Dados**
  C. **Controle**

# *Hazards*

- ***Hazard* estrutural:**
  - Situação de conflito pelo uso (simultâneo) de um mesmo recurso de *hardware*.
  - Ocorre quando duas instruções precisam utilizar o mesmo componente de *hardware* – para fins distintos ou com dados diferentes – no mesmo ciclo de relógio.
  - **Exemplo:** única memória sendo acessada em dois momentos distintos (e.g., para busca de instrução e para carregamento de um valor em um registrador).

# *Hazards*

- ***Hazard* de dados:**
  - Ocorrem quando uma instrução depende da conclusão de uma instrução prévia que ainda esteja na *pipeline* para realizar sua operação e/ou acessar um dado.

  - **Exemplo:**

    add $s0, $t0, $t1

    sub $t2, $s0, $t3

    - A instrução *add* somente escreve seu resultado no final do 5º estágio da *pipeline*.
    - Logo, teríamos que desperdiçar três ciclos de relógio aguardando até que o resultado correto ($s0) pudesse ser lido pela instrução *sub*.

# *Hazards*

- ***Hazard* de controle:**
  - Surge por causa da necessidade de tomar uma decisão baseada em resultados de uma instrução enquanto outras estão em execução.
  - Ou seja, está ligado a instruções de desvio.
  - **Problema:**
    - A *pipeline* inicia a busca da instrução subsequente ao *branch* no próximo ciclo de relógio.
    - Porém, não há como a *pipeline* saber qual é a instrução correta a ser buscada, uma vez que acabou de receber o próprio *branch* da memória.

# Operação

- O desempenho de um processador pode ser aperfeiçoado através de :
  - Sinais de relógio mais rápidos (contudo existe um limite).
  - *Pipeline*: divisão da execução das instruções em estágios.
  - Múltiplas ALUs para suportar mais de uma sequência de instruções.

- **Superescalar:** é capaz de realizar duas ou mais operações escalares em paralelo, fazendo uso de duas ou mais ALUs.
  - Estática – a ordem das operações tem que ser definida em tempo de compilação.
  - Dinâmica – reordenam as instruções em tempo de execução para fazer uso de ALUs adicionais.

- **VLIW** (*very long instruction word*): arquitetura superescalar estática na qual cada palavra na memória possui múltiplas operações independentes.

# Visão do programador

- O programador nem sempre precisa conhecer todos os detalhes da arquitetura, podendo trabalhar em diferentes níveis de abstração:
  - Nível Assembly – linguagem comumente ligada às características do processador.
  - Linguagens estruturadas – C, C++, Java, etc.

- A maior parte do desenvolvimento hoje é realizado utilizando-se linguagens estruturadas.
  - Programação em linguagem *Assembly* pode ser necessária.
  - **Drivers:** partes do programa que comunicam com e/ou controlam (dirigem) outros dispositivos.
    - Frequentemente é preciso levar em conta aspectos temporais, manipulação de bits. Nestes casos, *Assembly* pode ser a melhor opção.

# Assembly

- Difícil trabalhar com código de máquina (0s e 1s)
- Assembly – Usa mnemônicos
  - ***Load*** instruction—**MOV Ra, d**
  - ***Store*** instruction—**MOV d, Ra**
  - ***Add*** instruction—**ADD Ra, Rb, Rc**

```
0: RF[0]=D[0]
1: RF[1]=D[1]
2: RF[2]=RF[0]+RF[1]
3: D[9]=RF[2]
```

```
0: 0000 0000 00000000
1: 0000 0001 00000001
2: 0010 0010 0000 0001
3: 0001 0010 00001001
```

machine code

```
0: MOV R0, 0
1: MOV R1, 1
2: ADD R2, R0, R1
3: MOV 9, R2
```

assembly code

# Visão do programador

- **Conjunto de instruções:** corresponde ao repertório de operações elementares que o programador pode invocar.

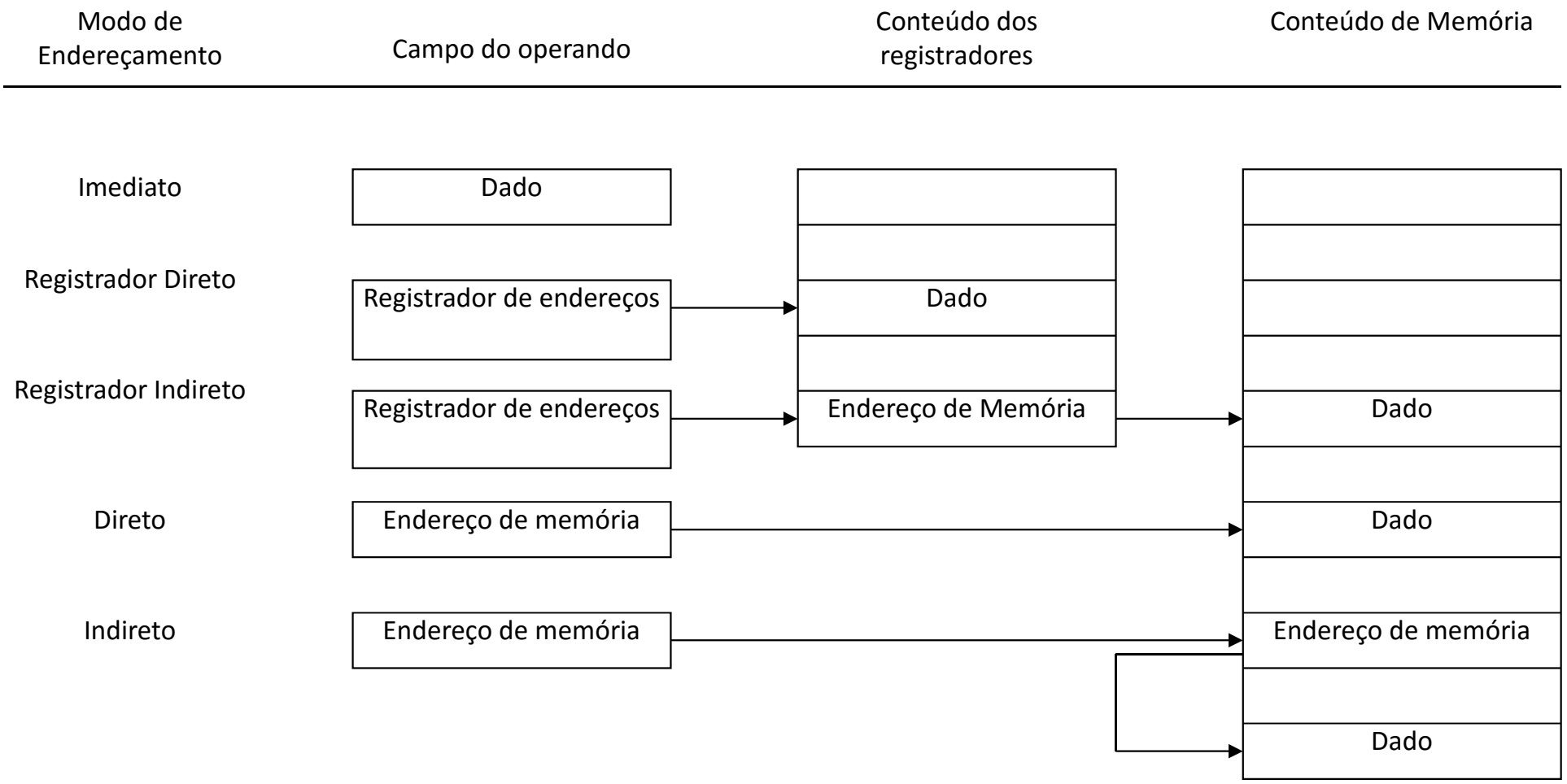
| Opcode | 1º Operando | 2º Operando |
|---|---|---|

  - Opcode: código binário que define um identificador único para cada instrução.

- **Tipos básicos de instruções:**
  - Transferência de dados: memória/registrador, registrador/registrador, I/O, etc.
  - Lógico-aritmética: usa registradores como entradas para a ALU e armazena resultados em registradores.
  - Desvio (*branch*): determina o valor do PC quando deseja-se realizar um salto para outro ponto do código, em vez da próxima instrução imediata.

# Visão do programador

- **Modos de endereçamento:**



# Visão do programador

- **Espaço para programas e dados**
  - Processadores em sistemas embarcados costumam ser muito limitados: por exemplo, 64 Kbytes de programa, 256 bytes de RAM (expansível).

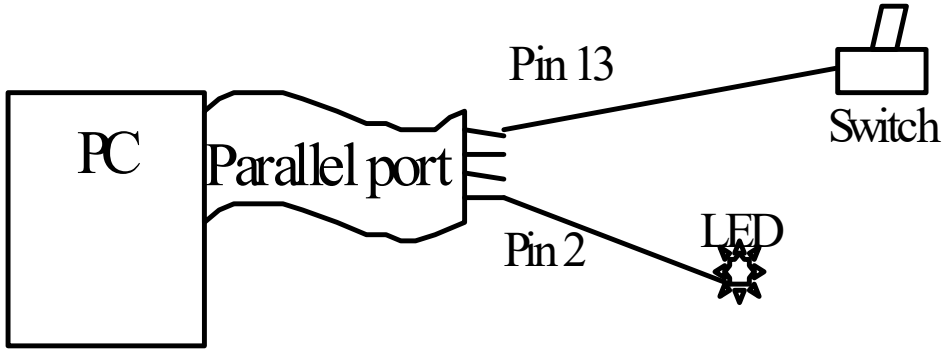
- **Registradores:** quantos estão disponíveis? Existem registradores com funções especiais?

- **I/O**
  - Como é realizada a comunicação com os sinais externos (portas)?

- **Interrupções**
  - Onde deve ser armazenada a rotina de tratamento de interrupção? Quantos pinos do processador são destinados a sinais de interrupção?

# Visão do programador

**Exemplo:** porta paralela monitora a chave de entrada e acende ou apaga o LED de acordo com a posição da chave.



# Visão do programador

- Um conjunto especial de três registradores é utilizado para leitura/escrita de valores nos pinos da porta paralela.

| Pino de conexão (LPT) | Direção (I/O) | Posição / Registrador |
|---|---|---|
| 1 | Saída | Bit 0 do regist. #2 |
| 2-9 | Saída | Bits 0-7 do regist. #0 |
| 10, 11, 12, 13, 15 | Entrada | Bits 6, 7, 5, 4, 3 do regist. #1 |
| 14, 16, 17 | Saída | Bits 1-3 do regist. #2 |

- Endereço base do banco de registradores: $3BC_h$.
  - Logo, o regist. #2 está no endereço 3BC + 2 = $3BE_h$.

# Visão do programador

- Programa em Assembly

```
CheckPort proc
     push ax              ; save the content
     push dx              ; save the content
     mov  dx, 3BCh + 1    ; base + 1 for register #1
     in   al, dx          ; read register #1
     and  al, 10h         ; mask out all but bit # 4
     cmp  al, 0           ; is it 0?
     jne  SwitchOn        ; if not, we turn the LED on

SwitchOff:
     mov  dx, 3BCh + 0    ; base + 0 for register #0
     in   al, dx          ; read the current state of the port
     and  al, FEh         ; clear first bit (masking)
     out  dx, al          ; write it out to the port
     jmp  Done            ; we are done

SwitchOn:
     mov  dx, 3BCh + 0    ; base + 0 for register #0
     in   al, dx          ; read the current state of the port
     or   al, 01h         ; set first bit (masking)
     out  dx, al          ; write it out to the port

Done:
     pop  dx              ; restore the content
     pop  ax              ; restore the content
CheckPort endp
```

# Visão do programador

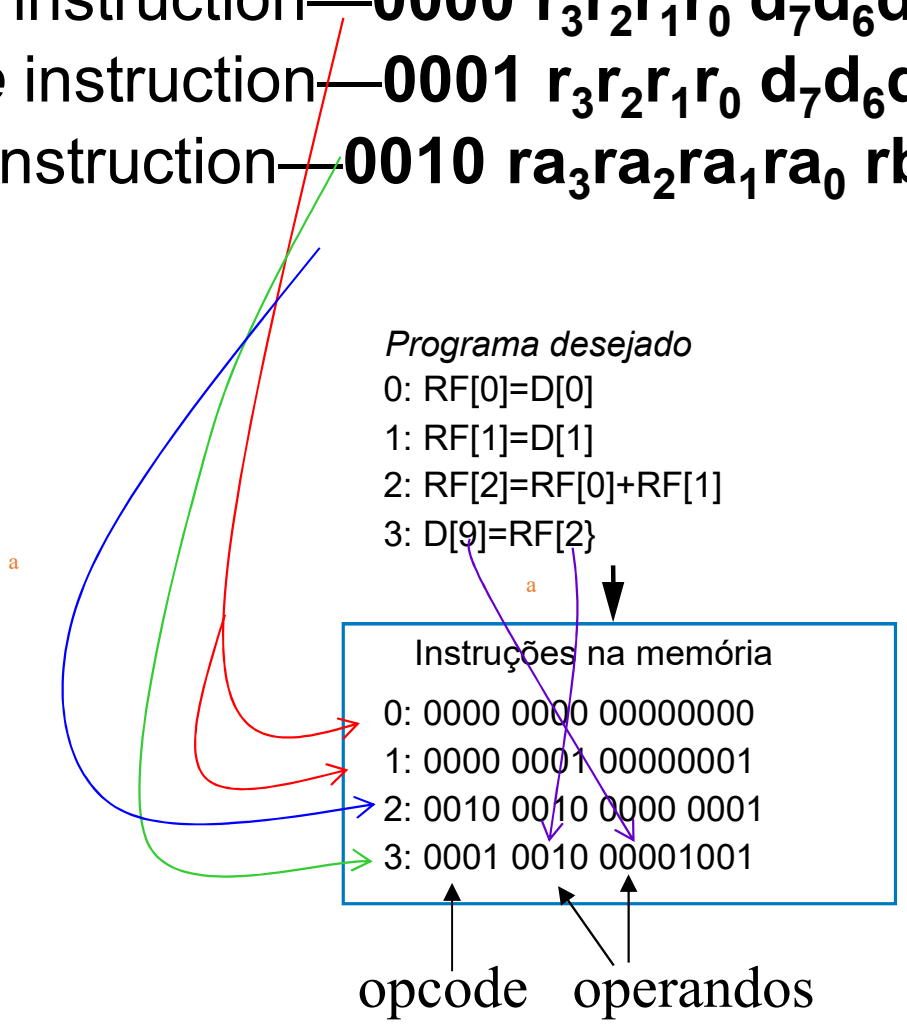
Sistema operacional:

- Esconde alguns detalhes do *hardware* e provê à camada de aplicação (que é onde o programador atua) uma interface para o *hardware* por meio do mecanismo de chamadas de sistema.
- Administração de arquivos, acesso à memória.
- Interface com teclado / display.
- Sequenciamento da execução de múltiplos programas (divisão do tempo de uso da CPU).

# Processador de uso geral de 3 instruções

Conjunto de instruções – Lista de instruções permitidas e suas representações na memória

- ***Load*** instruction—**0000 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$**
- ***Store*** instruction—**0001 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$**
- ***Add*** instruction—**0010 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**




Instruções em 0s e 1s – *código de máquina*

# Processador de uso geral de 6 instruções

Conjunto de instruções – Lista de instruções permitidas e suas representações na memória

- ***Load constant*** instruction—**0011 $r_3r_2r_1r_0$ $c_7c_6c_5c_4c_3c_2c_1c_0$**
- ***Subtract*** instruction—**0100 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**
- ***Jump-if-zero*** instruction—**0111 $ra_3ra_2ra_1ra_0$ $o_7o_6o_5o_4o_3o_2o_1o_0$**

**TABLE 8.1 Six-instruction instruction set..**

| Instruction | Meaning |
|---|---|
| MOV Ra, d | RF[a] = D[d] |
| MOV d, Ra | D[d] = RF[a] |
| ADD Ra, Rb, Rc | RF[a] = RF[b]+RF[c] |
| MOV Ra, #C | RF[a] = C |
| SUB Ra, Rb, Rc | RF[a] = RF[b]-RF[c] |
| JMPZ Ra, offset | PC=PC+offset if RF[a]=0 |

**TABLE 8.2 Instruction opcodes.**

| Instruction | Opcode |
|---|---|
| MOV Ra, d | 0000 |
| MOV d, Ra | 0001 |
| ADD Ra, Rb, Rc | 0010 |
| MOV Ra, #C | 0011 |
| SUB Ra, Rb, Rc | 0100 |
| JMPZ Ra, offset | 0101 |

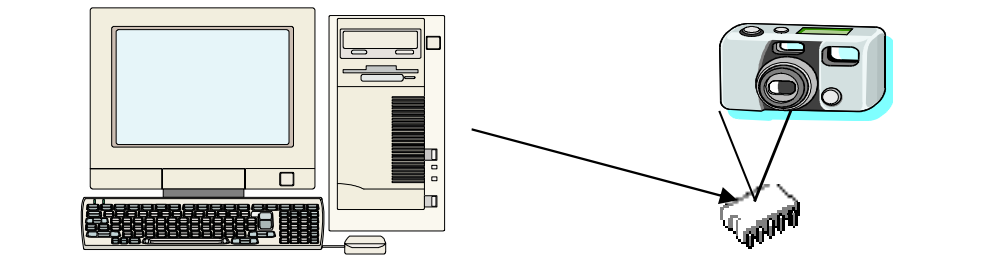
# Ambiente de Desenvolvimento

## Processador de desenvolvimento

- Corresponde ao processador no qual escrevemos e depuramos o programa.
- Usualmente presente em um PC.

## Processador alvo

- Corresponde ao processador no qual o programa final será carregado e que irá efetivamente fazer parte da implementação do sistema embarcado.
- Frequentemente, é diferente do processador de desenvolvimento.

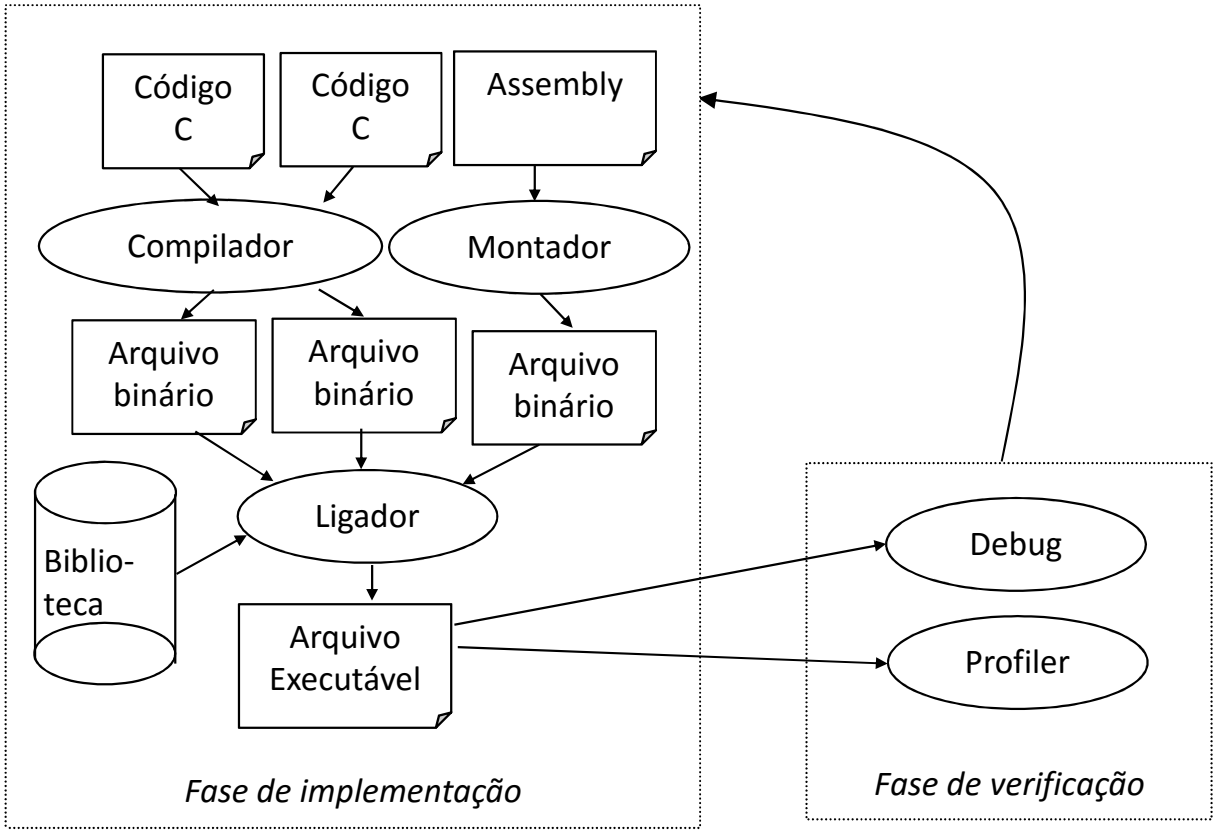


Processador de desenvolvimento Processador Alvo

# Ambiente de Desenvolvimento

- A programação de um processador inserido no sistema embarcado apresenta algumas diferenças sutis, porém importantes, em relação ao projeto de *software* em um *desktop*.

- Em um desktop:



# Ambiente de Desenvolvimento

Em um sistema embarcado, o processador alvo comumente é diferente do processador de desenvolvimento. Logo, embora a programação seja feita no processador de desenvolvimento, o código gerado precisa ser compatível com o formato de instrução utilizado pelo processador alvo.

- **Compiladores:** traduzem programas escritos em linguagens estruturadas em instruções de máquina (ou Assembly), possivelmente realizando algumas otimizações no código.
  - **Compilador-cruzado (*cross compiler*):** é executado em um processador (desenvolvimento), mas gera código para outro processador (alvo).

- **Montadores:** traduzem instruções mnemônicas (Assembly) em instruções de máquina (binário), fazendo também a tradução de endereços (no lugar dos rótulos).
  - **Cross-assembler.**

# Ambiente de Desenvolvimento

- **Teste e depuração:**
  - Depurar um programa que roda em um sistema embarcado requer que tenhamos controle sobre o tempo, bem como controle sobre o ambiente no qual está inserido o sistema, e também a habilidade de acompanhar a execução do programa a fim de detectar erros – é um processo mais complexo que aquele realizado em *desktop*.

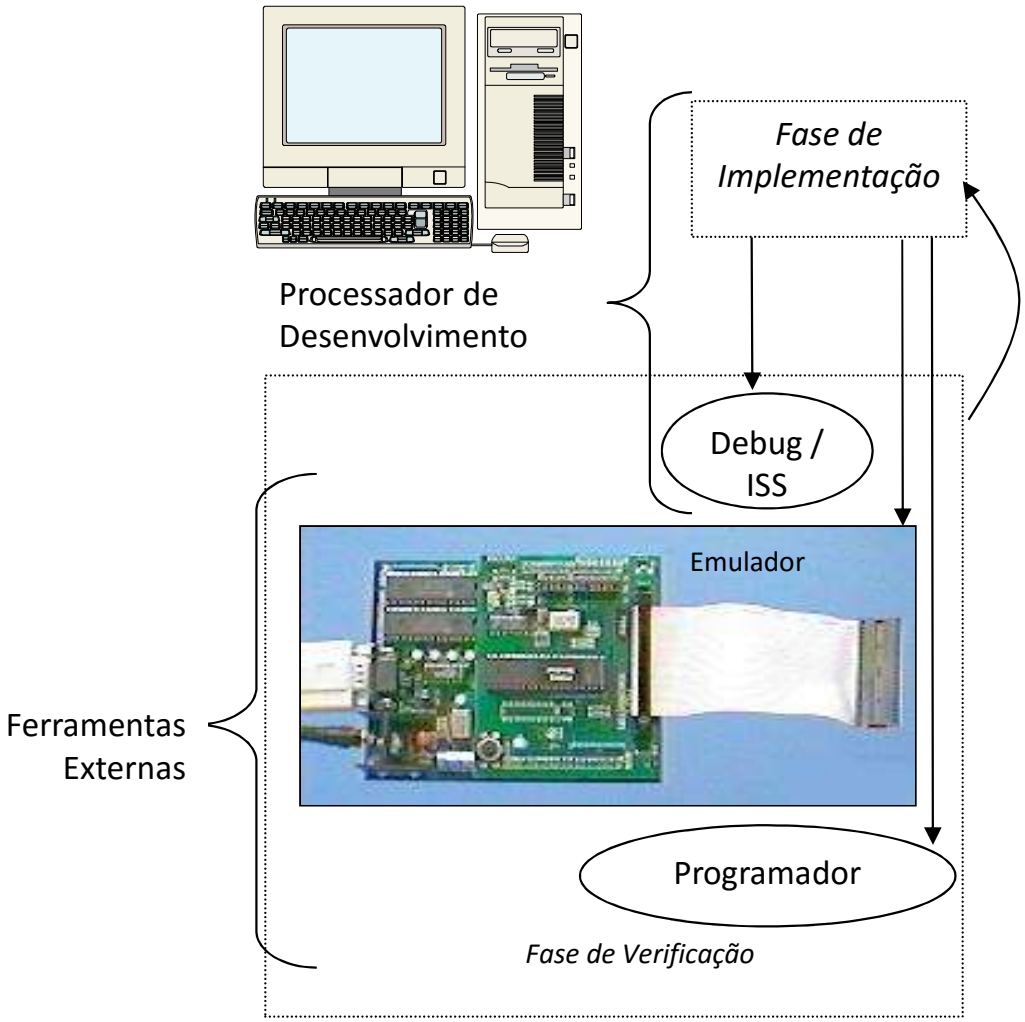
- **ISS (Instruction set simulator):** roda no processador de desenvolvimento, mas executa código projetado para o processador alvo – imita ou simula a função do processador alvo (também chamado de máquina virtual).

- **Emulador:** suporta a depuração do programa enquanto ele é executado no processador alvo. Normalmente, consiste de um depurador acoplado a uma placa conectada ao desktop e que contém o processador alvo e um circuito adicional de suporte.

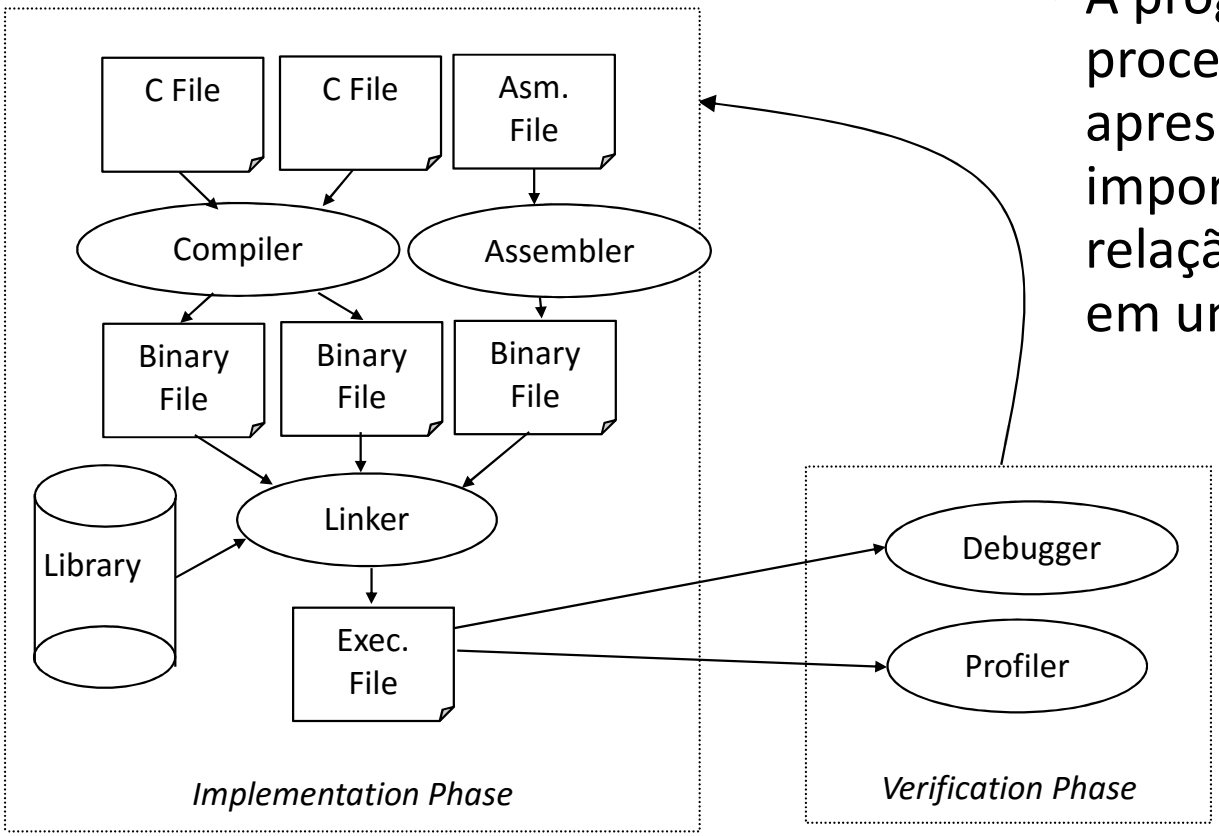
- **Programadores de dispositivo:** Carregam um programa da memória do processador de desenvolvimento para o processador alvo.

# Ambiente de Desenvolvimento

- Em um sistema embarcado:

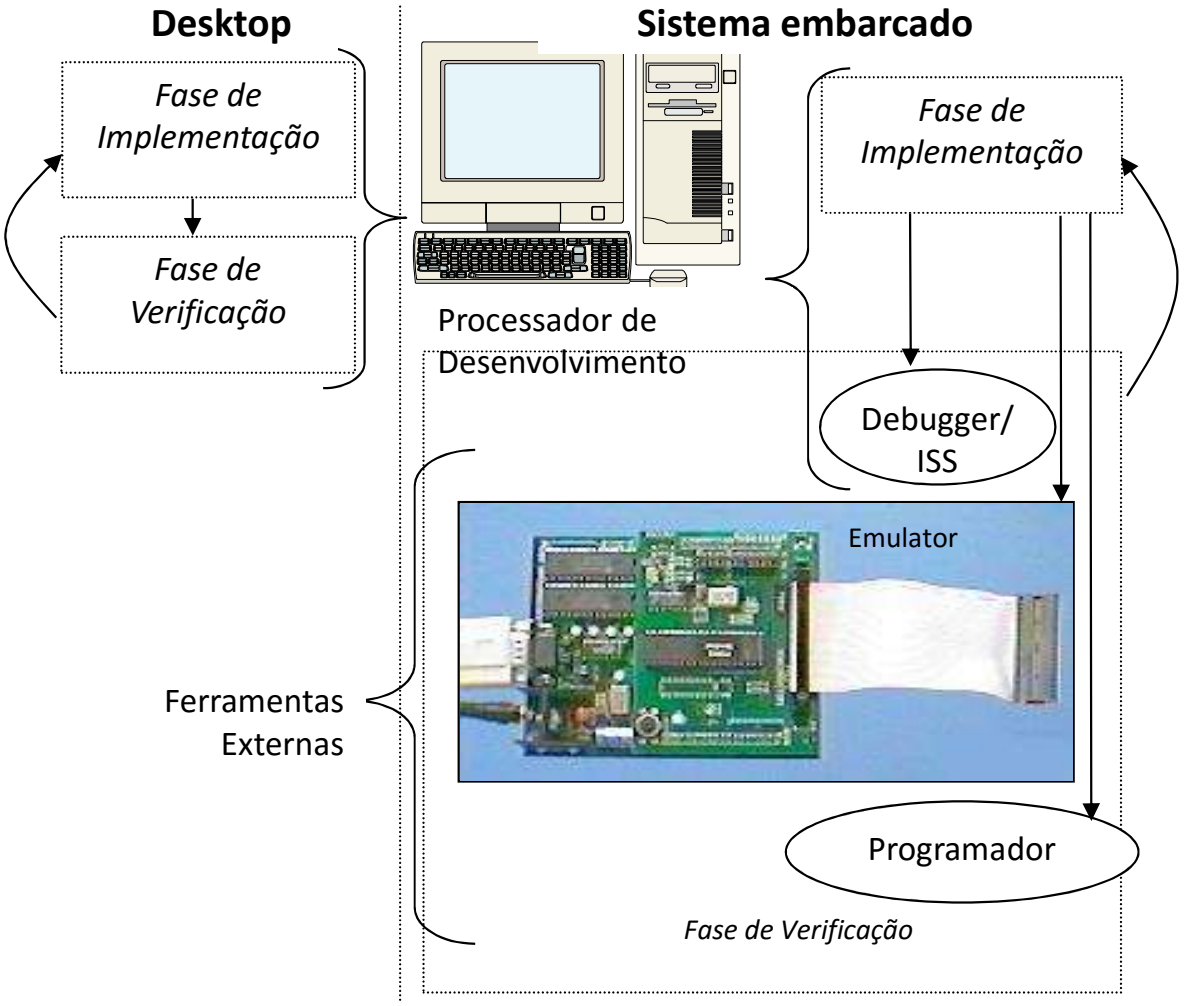


# Desenvolvimento do *Software*

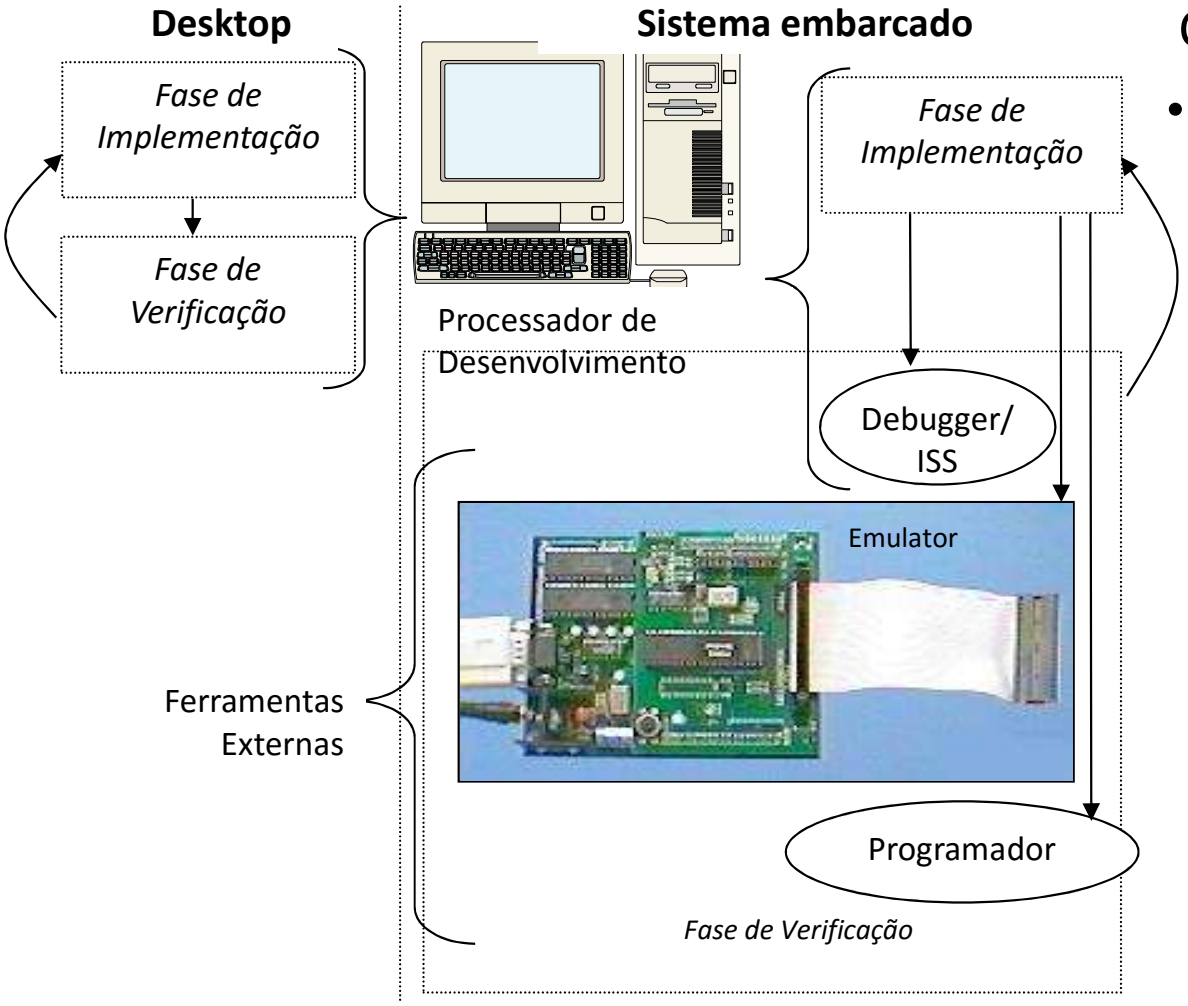



- A programação de um processador inserido em um SE apresenta algumas sutis, porém importantes diferenças em relação ao projeto de software em um desktop

# Desenvolvimento do *Software*



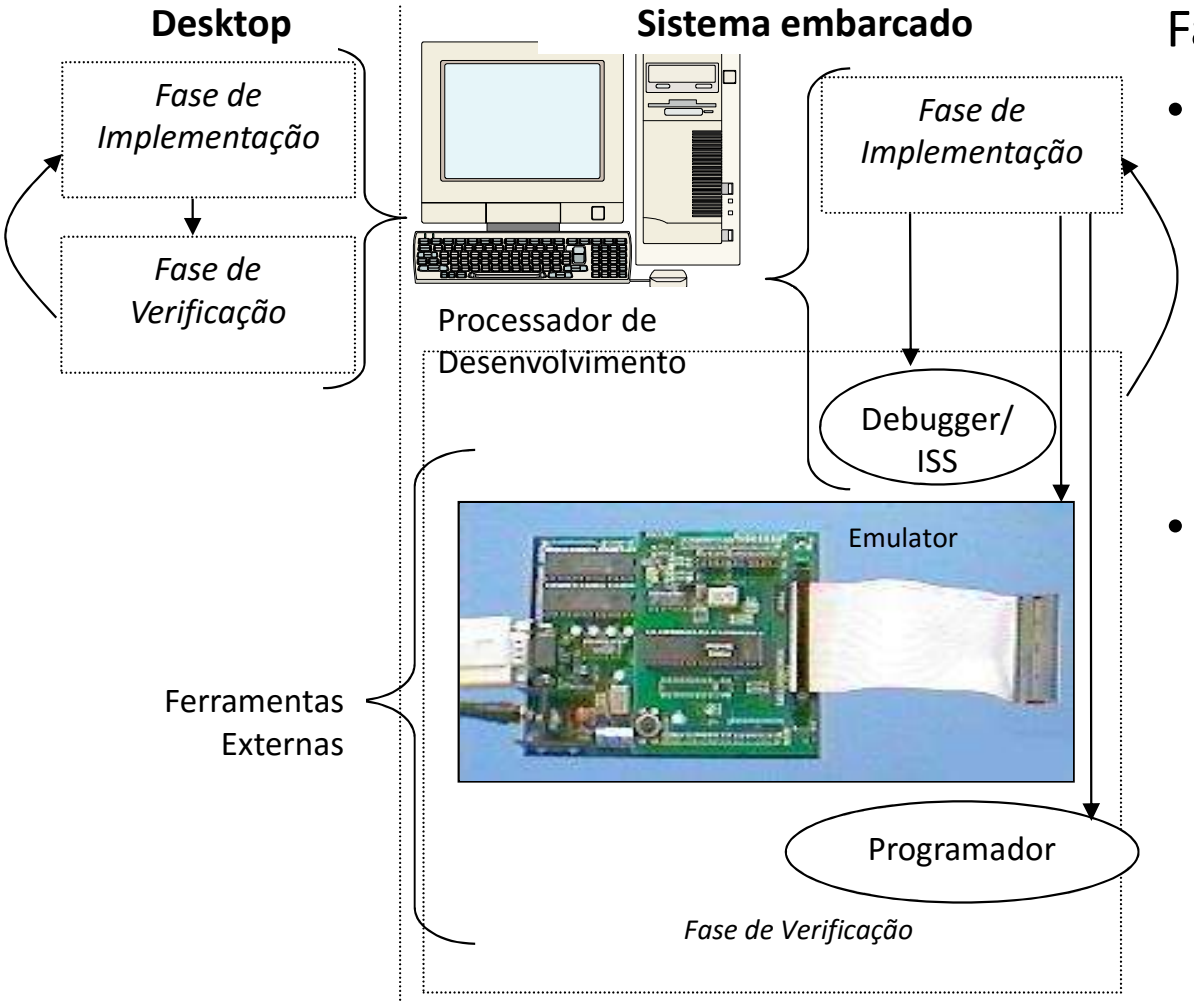
# Desenvolvimento do *Software*




## Cuidados na Fase de implementação

- Em um sistema embarcado, o processador alvo comumente é diferente do processador de desenvolvimento. Logo, embora a programação seja feita no processador de desenvolvimento, o código gerado precisa ser compatível com o formato de instrução utilizado pelo processador alvo.
  -

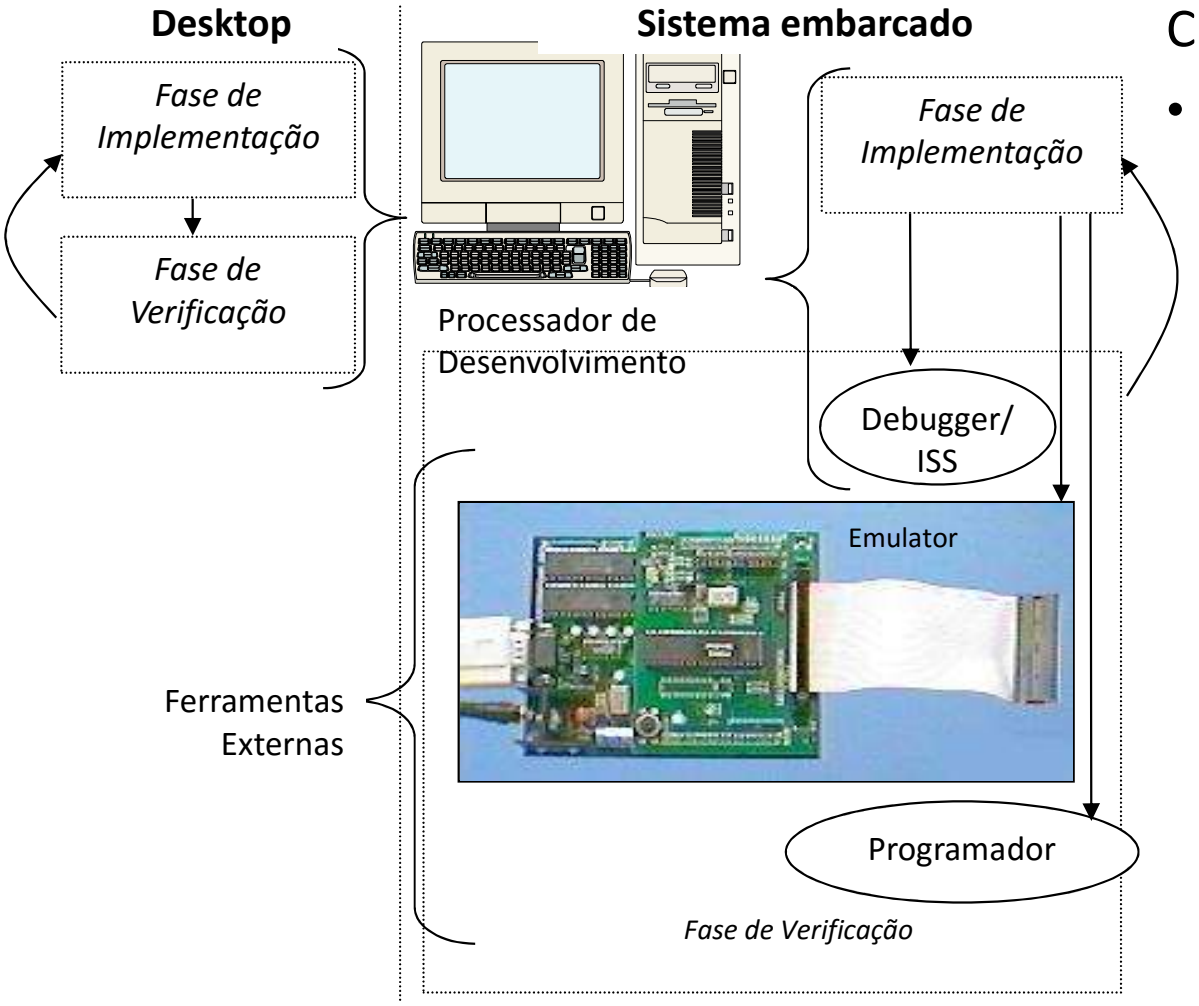
# Desenvolvimento do *Software*




## Fase de implementação

- Compilador: traduz programa em linguagem estruturada para linguagem de máquina (Assembly), realizando otimizações no código
  - Cross-compiler: é executado em um processador (desenvolv.), mas gera código para outro processador (alvo)
- Montador: traduzem instruções mnemônicas (Assembly) em instruções de máquina (binário), fazendo também a tradução de endereços (no lugar dos rótulos)
  - Cross-assembler

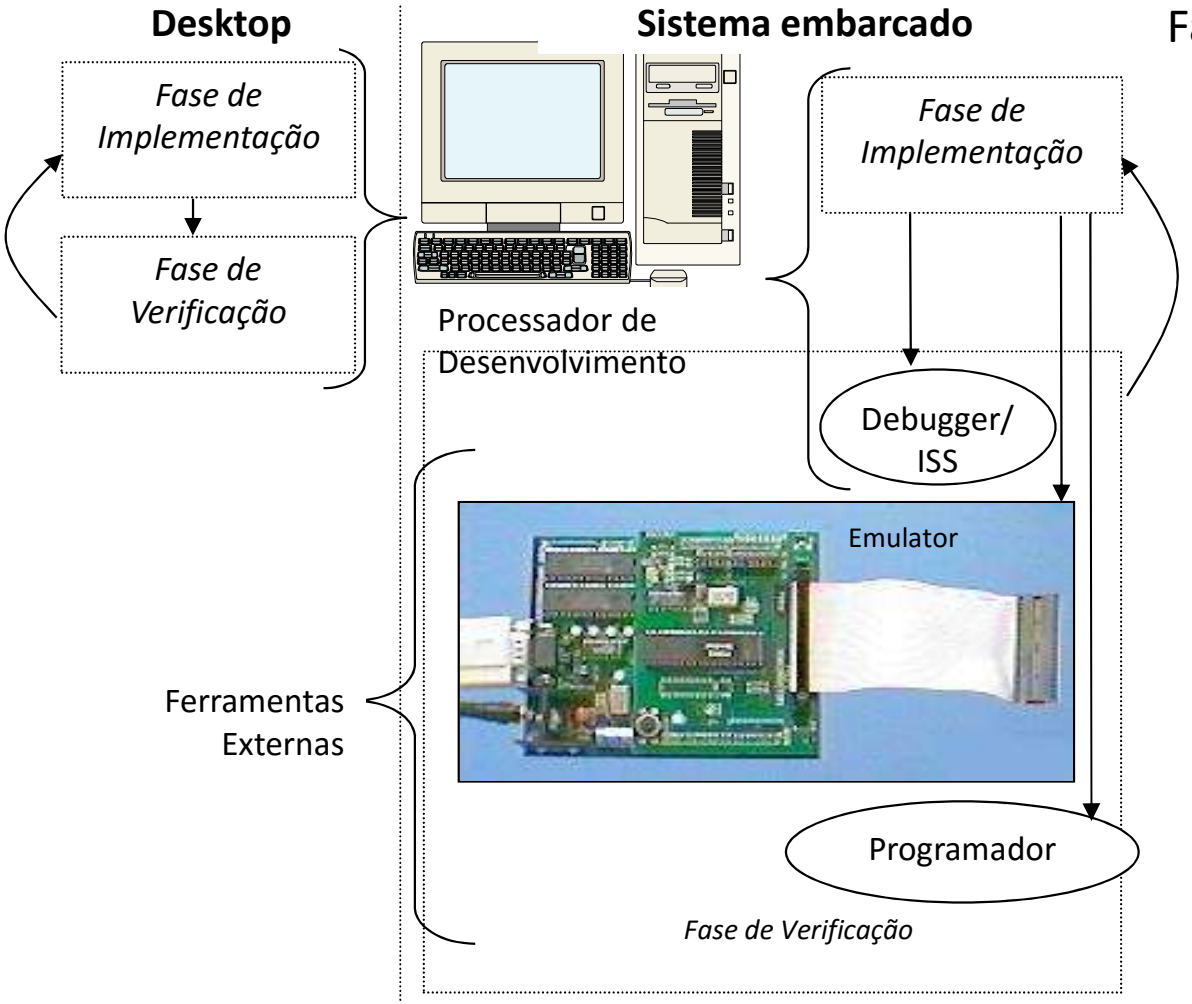
# Desenvolvimento do *Software*



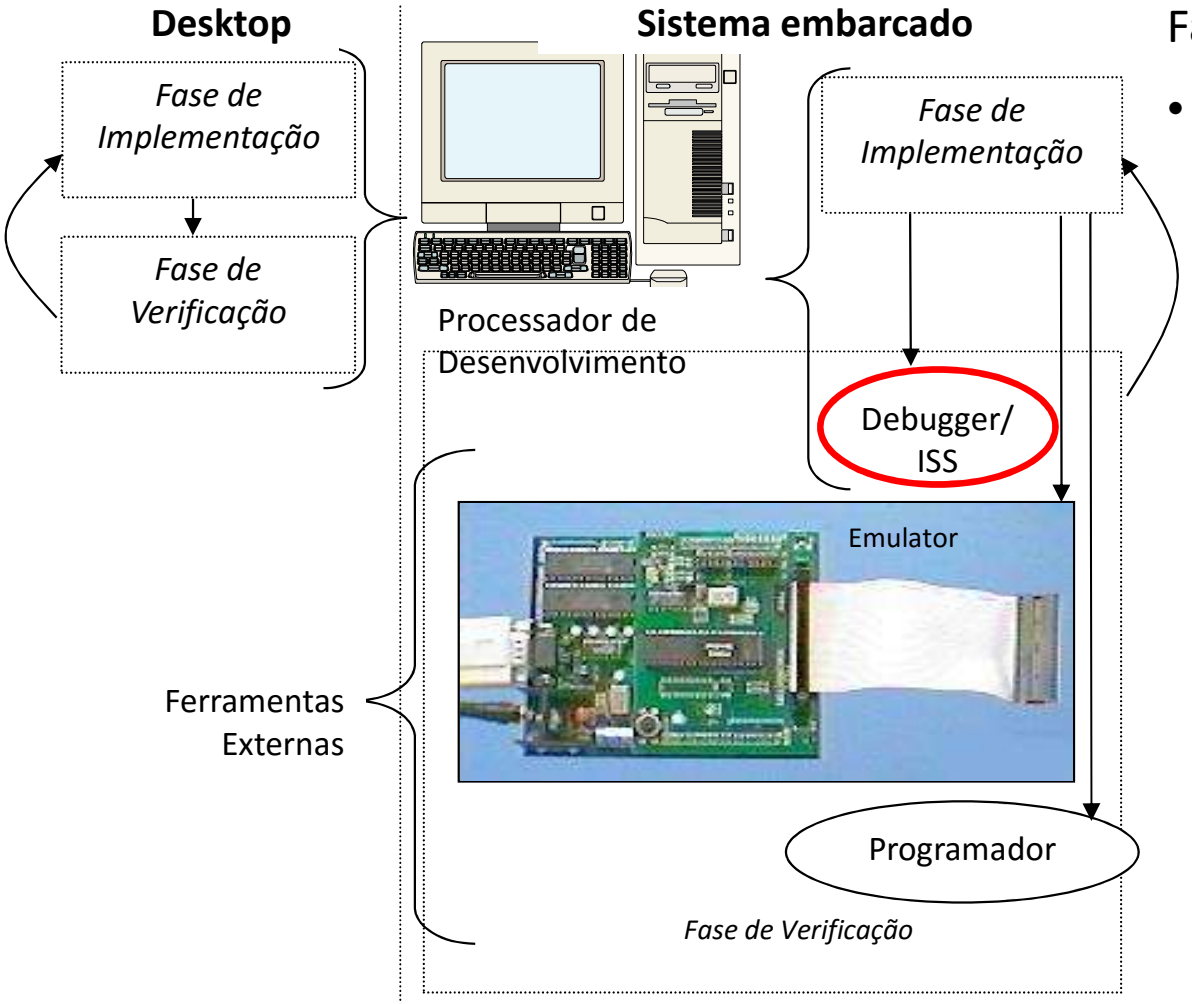

## Cuidados na Fase de verificação

- Depurar um programa que roda em um sistema embarcado requer que tenhamos controle sobre o tempo, bem como controle sobre o ambiente no qual está inserido o sistema, e também a habilidade de acompanhar a execução do programa a fim de detectar erros – é um processo mais complexo que aquele realizado em desktop.
  - SE: resposta certa no tempo certo
  - Desktop: resposta certa

# Testando e Depurando



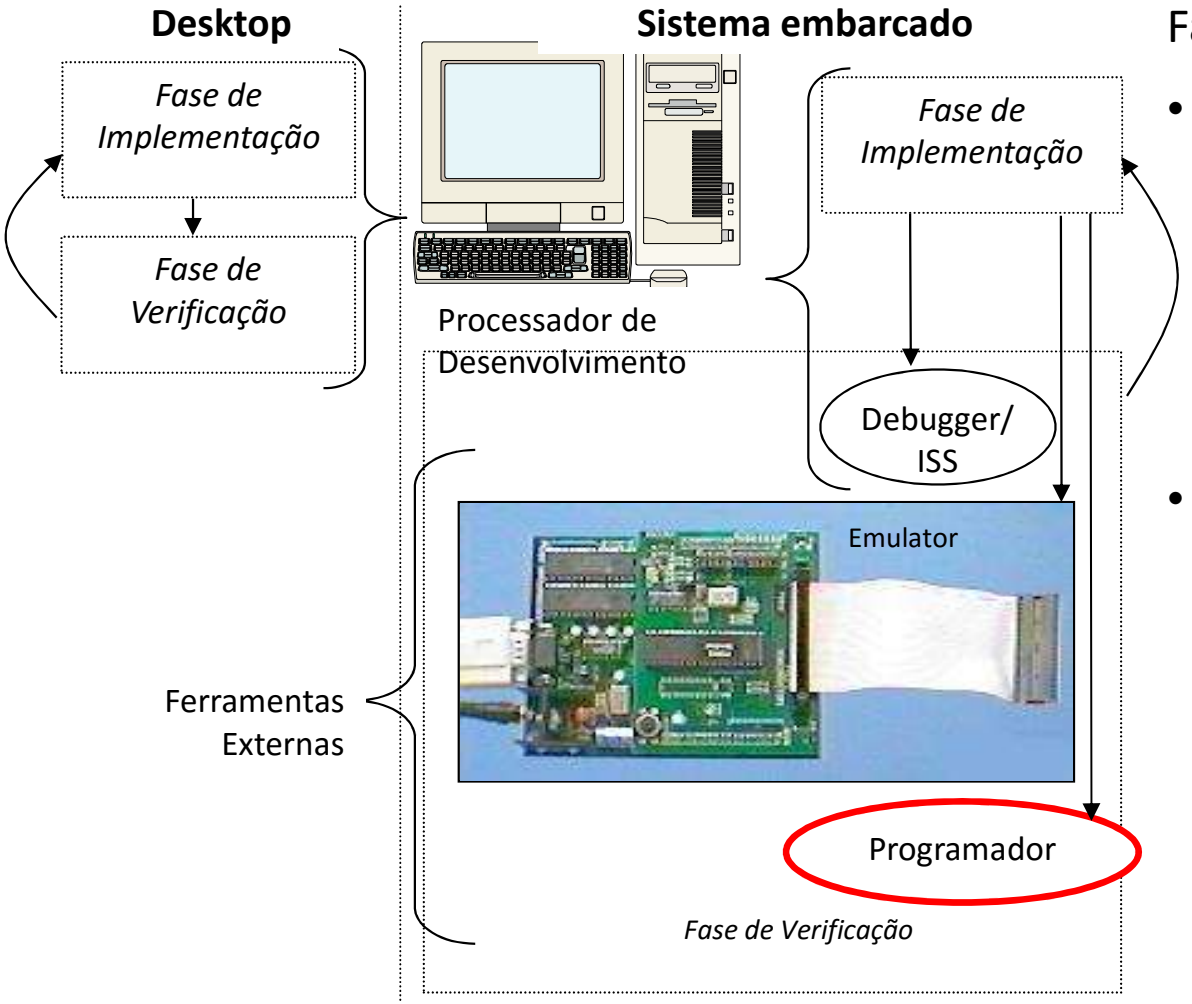
# Testando e Depurando




## Fase de verificação

- ISS
  - Controle durante a simulação: habilita *breakpoints*, procura por valores de registradores, habilita valores, execução passo-a-passo, etc.
  - Mas não interage com o ambiente real

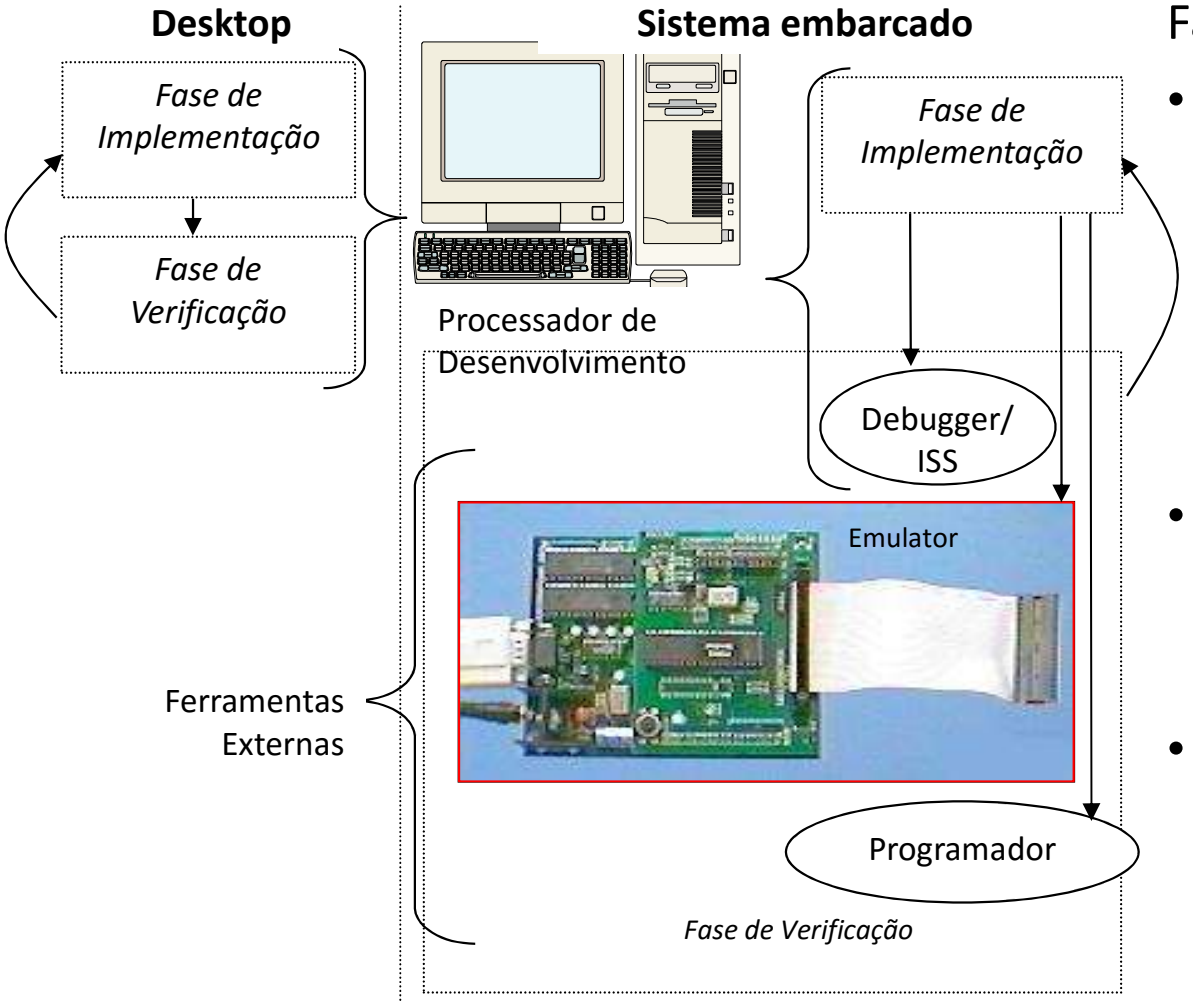
# Testando e Depurando




## Fase de verificação

- ISS
  - Controle durante a simulação: habilita *breakpoints*, procura por valores de registradores, habilita valores, execução passo-a-passo, etc.
  - Mas não interage com o ambiente real
- Programador de dispositivos
  - *Download* para a placa teste
  - Executado em ambiente real porém sem controle detalhado

# Testando e Depurando




## Fase de verificação

- ISS
  - Controle durante a simulação: habilita *breakpoints*, procura por valores de registradores, habilita valores, execução passo-a-passo, etc.
  - Mas não interage com o ambiente real
- Programador de dispositivos
  - *Download* para a placa teste
  - Executado em ambiente real porém sem controle detalhado
- Emulador
  - Execução em ambiente real, em tempo real ou próximo dele
  - Possui alguma possibilidade de controle a partir do PC

# Ambiente de Desenvolvimento

- Três maneiras de testar o sistema embarcado:

  - Depuração usando ISS – **menos realista e impreciso na observação do comportamento / abordagem mais rápida e simples**.
  - Emulação usando um emulador.
  - Teste de campo através do carregamento do programa diretamente na memória do processador alvo – **mais realista / abordagem mais lenta**.

# ASIP

## Application Specific Instruction set Processor

# ASIPs

- Processadores de Propósito Geral (GPP): muito geral para ser efetivo em aplicações exigentes
  - processamento de vídeo – exige *buffers* gigantescos e opera com vetores de dados extremamente longos, ineficiente para um GPP
- Processadores dedicados: possuem alto custo NRE e não são programáveis
- ASIPs – voltados a um campo particular de aplicação
  - Programável – alta flexibilidade, baixo NRE, baixo time-to-market
  - Desempenho, tamanho e consumo satisfatórios
  - Exemplos: controle de processo, processamento digital de sinais, processamento de vídeos, processamento de redes, telecomunicações, etc.

# Ex. Microcontrolador

- Para aplicações de controle
  - Leitura de sensores, ajustando atuadores
  - Na maioria das vezes lida com eventos (bits): o dado está presente, mas em pequenas quantidades
  - Ex: VCR, dispositivo de discos, máquina de lavar, forno de microondas
- Características do Microcontrolador
  - Periféricos no chip
    - Temporizadores, conversores A/D, comunicação serial, etc.
  - Memória de Programa e dados *on-chip*
  - Programador com acesso direto a pinos do chip
  - Instruções especializadas para operações de controle típicas (manipulação de bits e outras operações em nível lógico)

# Ex. Digital Signal Processor

- Para aplicações de processamento de sinais
  - Grande quantidade de dados digitalizados
  - Transformação de dados deve ser realizada rapidamente
  - Ex: filtro de voz para telefone celulares, TV digital, sintetizador de música
- Características do DSP
  - Vários registradores, blocos de memória, multiplicadores e outras unidades aritméticas
  - Multiplica-e-soma implementada em uma única instrução
  - Operações vetoriais eficientes – e.g., soma de dois vetores
  - Permite execução em paralelo de algumas funções
  - Incorpora periféricos ao chip: conversores AD e DA, DMA, timers, etc.

# Tendência: ASIPs ainda mais customizados

- Oportunidade de adicionar ao *datapath* umas poucas instruções, ou apagar algumas instruções
- Potenciais impactos significativos no desempenho, tamanho e potência consumida
- Problema: necessidade do compilador/depurador para o ASIP customizado
  - Lembre-se: a maioria dos desenvolvedores utiliza linguagem estrutural
  - Uma solução: geração automática de compiladores/depuradores
    - e.g., www.tensillica.com
  - Outra solução: retargettable compilers
    - e.g., www.improvsys.com (arquiteturas VLIW customizadas)

# Projeto do processador genérico

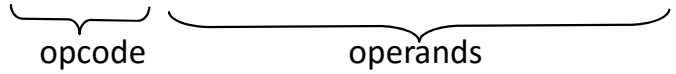
- Processador genérico = processador dedicado cujo propósito é processar instruções armazenadas em uma memória de programa.

- É possível utilizar a técnica de projeto de processadores dedicados, para construir um processador genérico.

# Projeto do processador genérico

## Conjunto de instruções

| Instrução Assembly | Primeiro byte | Segundo byte | Operação |
|---|---|---|---|
| MOV Rn, direct | 0000 Rn | direct | Rn = M(direct) |
| MOV direct, Rn | 0001 Rn | direct | M(direct) = Rn |
| MOV @Rn, Rm | 0010 Rn | Rm | M(Rn) = Rm |
| MOV Rn, #immed. | 0011 Rn | immediate | Rn = immediate |
| ADD Rn, Rm | 0100 Rn | Rm | Rn = Rn + Rm |
| SUB Rn, Rm | 0101 Rn | Rm | Rn = Rn - Rm |
| JZ Rn, relative | 0110 Rn | relative | PC = PC+ relative (somente se Rn for 0) |

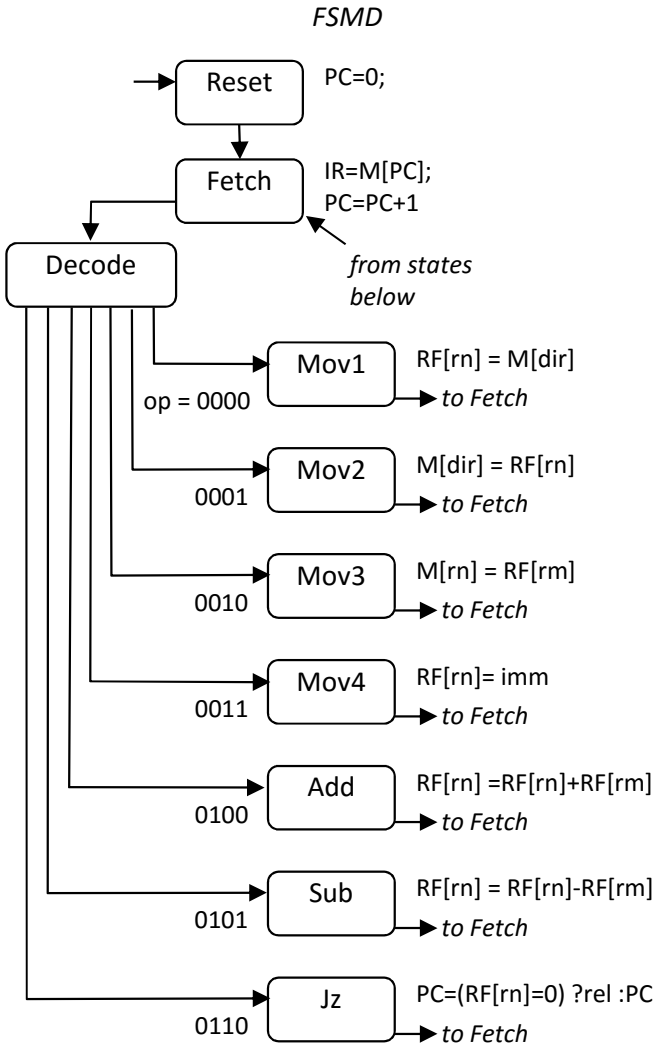


# Projeto do processador genérico

Finite State Machine with Datapath (FSMD)

- PC – 16 bits;
- IR – 16 bits;
- Memória M: 64k x 16;
- Arquivo de regist. (RF): 16 x 16.

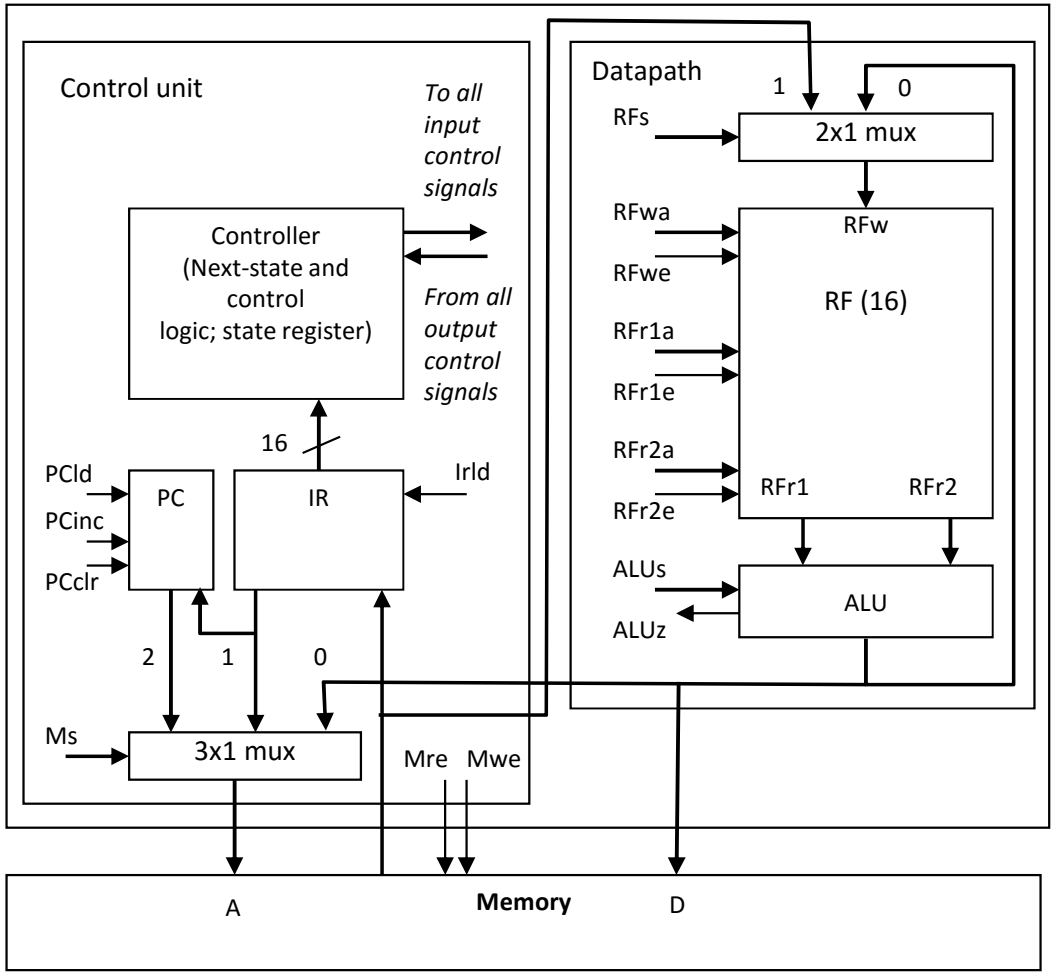
- Opcode: IR[15…12]
- rn (regist. destino): IR[11…8]
- rm (regist. origem): IR[7…4]
- dir: IR[7…0]
- imm : IR[7…0]
- rel : IR[7…0]



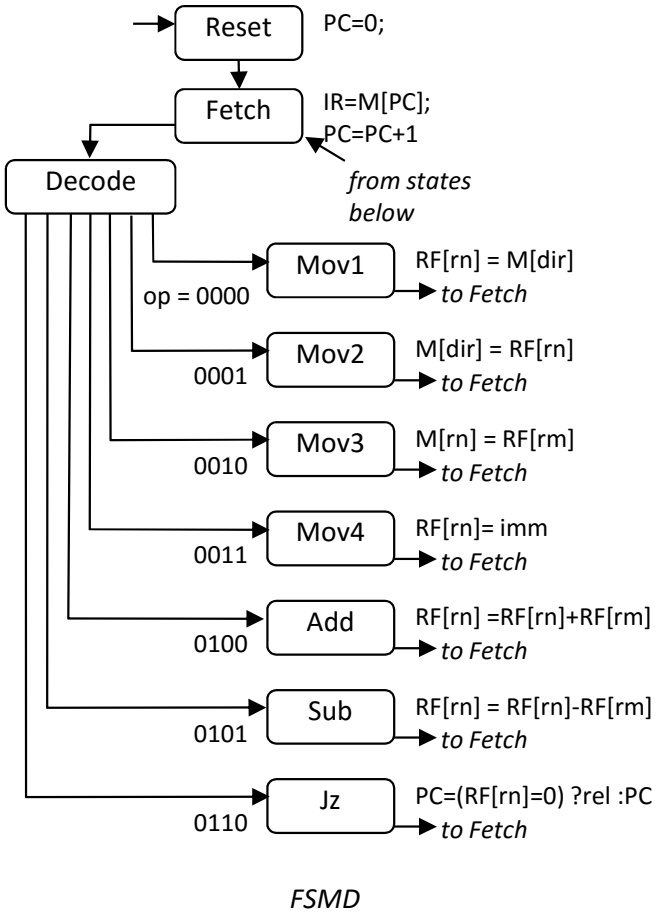
# Projeto do processador genérico

## *Datapath*

- Para cada varíavel declarada, crio um dispositivo de armazenamento (regist. PC e IR, memória M e arquivo de regist. RF).
- Unidades funcionais para executar as operações – uso de uma única ALU.
- Adiciono conexões entre as portas dos componentes como exigido pela FSMD, acrescentando multiplexadores quando há mais de uma conexão sendo colocada em alguma entrada.
- Crio identificadores únicos para todos os sinais de controle.



# Projeto do processador genérico




*FSMD*

PCclr=1;

MS=10;
Irld=1;
Mre=1;
PCinc=1;

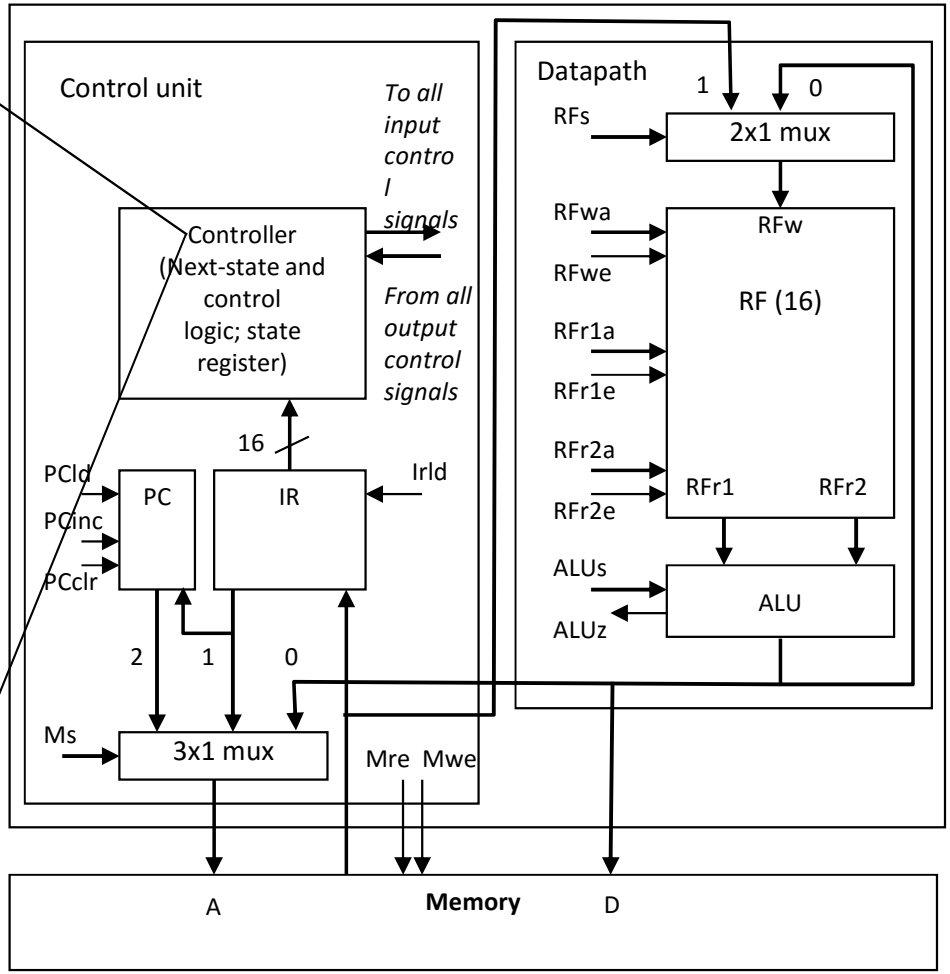
RFwa=rn; RFwe=1; RFs=01;
Ms=01; Mre=1;

RFr1a=rn; RFr1e=1;
Ms=01; Mwe=1;

RFr1a=rn; RFr1e=1;
Ms=10; Mwe=1;

RFwa=rn; RFwe=1; RFs=10;

RFwa=rn; RFwe=1; RFs=00;
RFr1a=rn; RFr1e=1;
RFr2a=rm; RFr2e=1; ALUs=00
RFwa=rn; RFwe=1; RFs=00;
RFr1a=rn; RFr1e=1;
RFr2a=rm; RFr2e=1; ALUs=01
PCld= ALUz;
RFrla=rn;
RFrle=1;



# Processador genérico x dedicado

- A diferença é que o processador dedicado põe o “programa” dentro de sua lógica de controle, enquanto um processador genérico o mantém em uma memória externa.

- Uma segunda diferença é que o *datapath* de um processador genérico é projetado sem o conhecimento de qual programa será colocado na memória, enquanto tal conjunto de comandos (programa) é conhecido no caso de um processador dedicado.

# Como escolher um processador?

**Critérios:**

- Técnicos: velocidade, potência consumida, tamanho, custo.
- Outros: ambiente de desenvolvimento, familiaridade, autorização para uso, etc.

**Velocidade**

- Aspecto relativamente difícil de ser medido e comparado.
- Tentativas:
  - Velocidade do relógio – mas o número de instruções por ciclo de relógio pode ser diferente.
  - Instruções por segundo – mas o trabalho realizado (ou a complexidade) das instruções pode ser diferente.

# Como escolher um processador?

- **Benchmarks:** tentativa de criar um mecanismo para comparação "justa" entre diferentes processadores.

  - Dhrystone: *Synthetic Benchmark* – conjunto de programas sintéticos de avaliação, desenvolvido em 1984 – medida em Dhrystones/segundo.

  - MIPS: 1 MIPS = 1757 Dhrystones/segundo (baseado no VAX 11/780 de Digital).
    - Amplamente utilizado hoje em dia.
    - Então, 750 MIPS = 750*1757 = 1.317.750 Dhrystones/segundos.

# Como escolher um processador?

| Processor | Clock speed | Periph. | Bus Width | MIPS | Power | Trans. | Price |
|---|---|---|---|---|---|---|---|
| General Purpose Processors | | | | | | | |
| Intel PIII | 1GHz | 2x16 K L1, 256K L2, MMX | 32 | ~900 | 97W | ~7M | $900 |
| IBM PowerPC 750X | 550 MHz | 2x32 K L1, 256K L2 | 32/64 | ~1300 | 5W | ~7M | $900 |
| MIPS R5000 | 250 MHz | 2x32 K 2 way set assoc. | 32/64 | NA | NA | 3.6M | NA |
| StrongARM SA-110 | 233 MHz | None | 32 | 268 | 1W | 2.1M | NA |
| Microcontroller | | | | | | | |
| Intel 8051 | 12 MHz | 4K ROM, 128 RAM, 32 I/O, Timer, UART | 8 | ~1 | ~0.2W | ~10K | $7 |
| Motorola 68HC811 | 3 MHz | 4K ROM, 192 RAM, 32 I/O, Timer, WDT, SPI | 8 | ~.5 | ~0.1W | ~10K | $5 |
| Digital Signal Processors | | | | | | | |
| TI C5416 | 160 MHz | 128K, SRAM, 3 T1 Ports, DMA, 13 ADC, 9 DAC | 16/32 | ~600 | NA | NA | $34 |
| Lucent DSP32C | 80 MHz | 16K Inst., 2K Data, Serial Ports, DMA | 32 | 40 | NA | NA | $75 |

# Resumo do Capítulo

- Processadores de Propósito Geral
  - bom desempenho, baixo NRE, flexíveis (várias aplicações)
  - consistem de UC, *datapath* e memórias (dados e programa)
  - prevalece o uso de linguagem estruturadas, mas programação em linguagem assembly ainda pode ser necessária
  - grande disponibilidade de ferramentas de desenvolvimento, incluindo simuladores de séries de instruções e emuladores de circuitos
- ASIPs
  - microcontroladores, DSP, processadores de rede, outros ASIPs customizados
- a escolha entre os vários modelos de processadores disponíveis é um passo importante dentro do projeto

# Créditos

## Fontes: Material de oferecimentos anteriores:

- Profa. Leticia
- Prof. Levy
- Prof. Leo Pini
- Profa. Alice
- Prof. Cristiano Araújo, CIN / UFPE
- …
- Algumas figuras foram extraídas de F. Vahid e T. Givargis, "*Embedded System Design: A Unified Hardware/Software Introduction*", John Wiley & Sons, 2001.


Source: http://www.gettyimages.com/detail/illustration/we-thank-you-royalty-free-illustration/174177500