Introdução ao MIPS
- Operações Lógicas -

Arquitetura de Computadores 2024/2025

Operações Bitwise

- Até agora fizemos operações aritméticas (`add`, `sub`, `addi`), acessos a memória (`lw` e `sw`), "branches" (`beq, bne,` …) e saltos (`j`).

- Em todos estes casos o registo é visto como um todo, representando um número com ou sem sinal.

- Nova Perspectiva: Ver o registo como um conjunto de 32 bits não relacionados, em vez de um número único representado por 32 bits.

- Neste contexto podemos querer aceder a bits individuais (ou grupos de bits).

- Para isso vamos precisar de duas novas classes de operações:
  - Operações lógicas
  - Shifts/Deslocamentos

Operações Lógicas

- As duas operações lógicas fundamentais são:
  - AND: saída 1 se, e só se, ambas as entradas são 1
  - OR: saída 0 se, e só se, ambas as entradas forem 0

- Sintaxe semelhante ao `add`, `addi`, etc.
  - **`OP`** `$destino, $fonte1, $fonte2/imediato`

- Nome das instruções:
  - `and`, `or`: Neste caso o terceiro argumento é um registo
  - `andi`, `ori`: Neste caso o terceiro argumento é um imediato

- Os operadores lógicos do MIPS são sempre bitwise, significando que o bit na posição 0 da saída depende dos bits 0's das entradas, o bit 1 dos bits 1's, etc.
  - C: Bitwise AND é `&` (e.g., `z = x & y;`)
  - C: Bitwise OR é `|` (e.g., `z = x | y;`)

Utilidade das Operações Lógicas (1/3)

- Note que fazer o `and` de um bit desconhecido com 0 produz sempre 0. Por outro lado, o resultado do `and` com 1 produz sempre o bit original.

- Isto é extremamente útil para criar máscaras (permitem isolar grupos de bits)
  - Exemplo:

    1011 0110 1010 0100 0011 1101 1001 1010

    mask: 0000 0000 0000 0000 0000 1111 1111 1111
  - O resultado deste AND é:

    0000 0000 0000 0000 0000 1101 1001 1010

isolar os últimos 12 bits

Utilidade das Operações Lógicas (2/3)

- A segunda sequência de bits do exemplo é designada máscara e serve para isolar os últimos 12 bits da direita, mascarando o resto da "bitstring" original.

- Usando a instrução `andi` e assumindo que a sequência original estava no registo `$t0`, teríamos:

```
andi    $t0,$t0,0xFFF
```

- De forma semelhante repare que fazer o `Or` de um bit desconhecido com 1 produz sempre 1, e com 0 produz o bit original.

- Esta propriedade pode ser utilizada para forçar (mascarar) certos bits a 1' s.
  - Se `$t0` contém `0x12345678`, então depois da instrução:

```
ori $t0, $t0, 0xFFFF
```

  - … `$t0` contém `0x1234FFFF`.

Utilidade das Operações Lógicas (3/3)

- Assumindo que $t0 = `0x12345678`

```
ori  $t0, $t0, 0xFFF
```

- Isto é extremamente útil para forçar certos conjuntos de bits a 1
  - Exemplo:

0001 0010 0011 0100 0101 0110 0111 1000
mask: 0000 0000 0000 0000 0000 1111 1111 1111

  - O resultado deste OR é:

0001 0010 0011 0100 0101 1111 1111 1111

Resultado:

`$t0 = 0x12345FFF`

Forçar os últimos 12 bits a 1

Instruções de Deslocamento (1/3)

- Sintaxe

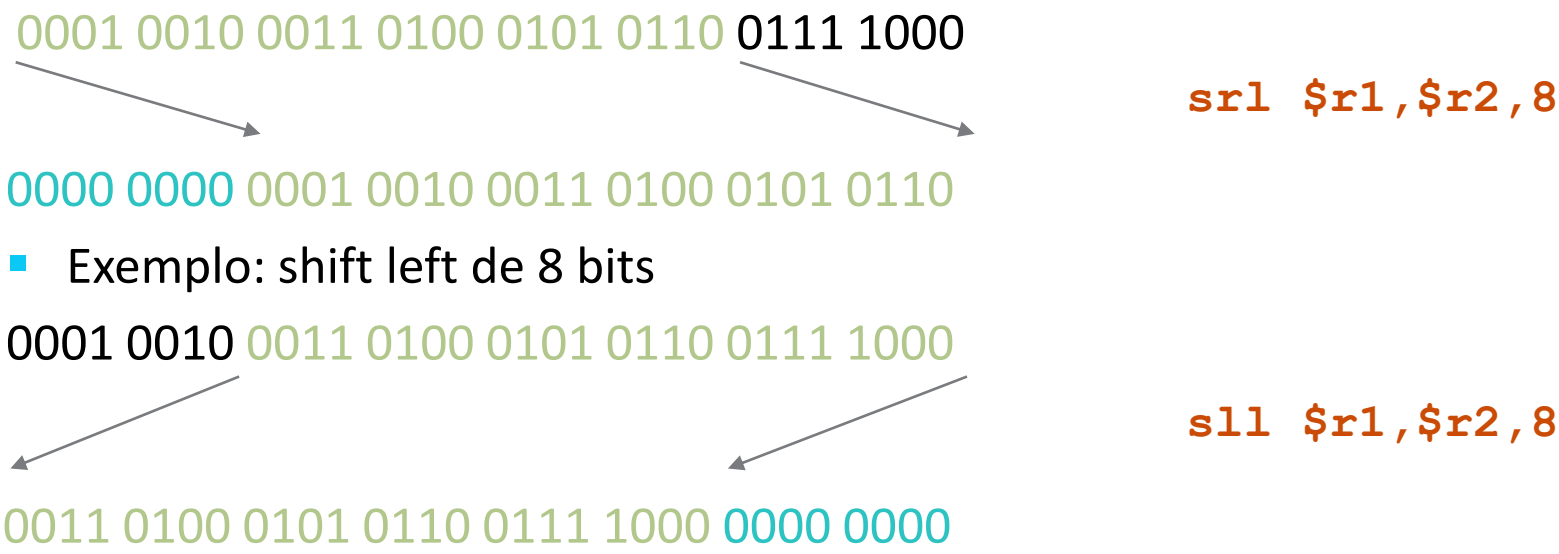
```
OP $destino, $fonte, imediato
```

- O valor imediato especifica o número de bits que são deslocados (<32)

- MIPS shift instructions:
  - `sll` (shift left logical): desloca para a esquerda e preenche os bits vazios com 0's
  - `srl` (shift right logical): desloca para a direita e preenche os bits vazios com 0's
  - `sra` (shift right arithmetic): desloca para a direita e preenche os bits vazios com a extensão de sinal
  - `sla` (shift left arithmetic) é idêntico ao `sll`

Instruções de Deslocamento (2/3)

- Deslocamentos lógicos para a esquerda e direita
  - Exemplo: shift right de 8 bits



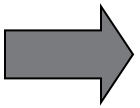
  - Exemplo: shift left de 8 bits

- Um bom compilador de C detecta quando existem multiplicações por potências de 2 e usa a instrução `sll`

Por exemplo: `a *= 8;` (em C), compila como: `sll $s0,$s0,3` (em MIPS)

Instruções de deslocamento (3/3)

- Deslocamento aritmético
  - Exemplo: shift right arith de 8 bits

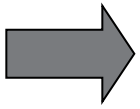


0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

```
sra $r1,$r2,8
```

  - Exemplo: shift right arith de 8 bits



1001 0010 0011 0100 0101 0110 0111 1000

1111 1111 1001 0010 0011 0100 0101 0110

```
sra $r1,$r2,8
```

- A instrução `sra` é utilizada para fazer divisões com sinal por potências de 2

Introdução ao MIPS
- Funções -

Arquitetura de Computadores 2024/2025

Funções em C

```
main() {
   int i,j,k;
   ...
   k = soma(i,j);
   ...
}
```

```
int soma (int op1, int op2){
   int res;
   res = op1 + op2;

   return res;
}
```

Numa chamada a função, que informação é que o compilador/programador precisa de registar ?

Que instruções permitem fazer isto?

Chamada de funções

- No MIPS os registos são fundamentais para guardar a informação necessária à chamada de funções.

- Convenção de utilização de registos:
  - Endereço de retorno. `$ra`
  - Argumentos / Parâmetros: `$a0, $a1, $a2, $a3`
  - Retorno de valores: `$v0, $v1`
  - Variáveis locais: `$s0, $s1, … , $s7`

Instruções de suporte a funções (1/6)

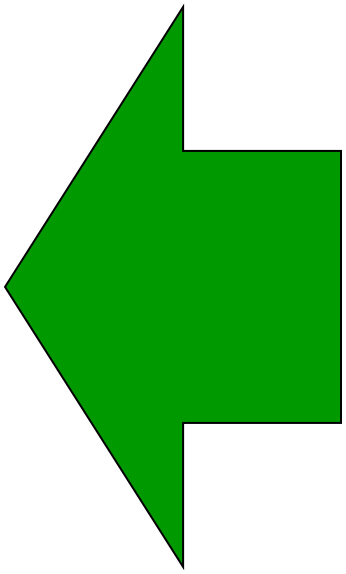
C

```
... soma(i,j);... /* i,j:$s0,$s1 */

int soma(int x, int y) { // x,y: $a0,$a1
   int res=op1+op2; return res; // $v0: res
}
```

MIPS

address
1000
1004
1008
1012
1016
2000
2004



No MIPS todas as instruções têm 4 bytes e são armazenadas em memória de forma semelhante aos dados. Estes são os endereços onde o programa está armazenado.

Instruções de suporte a funções (2/6)

C

```
... soma(i,j);... /* i,j:$s0,$s1 */

int soma(int x, int y) { // x,y: $a0,$a1
   int res=op1+op2; return res; // $v0: res
}
```

MIPS

```
address
  1000 add  $a0,$s0,$zero  # x = i
  1004 add  $a1,$s1,$zero  # y = j
  1008 addi $ra,$zero,1016 #$ra=1016
  1012 j  sum              #jump to soma
  1016 ...
  2000 sum: add $v0,$a0,$a1
  2004 jr   $ra # nova instrução - salta
# para o endereço apontado pelo registo
```

Instruções de suporte a funções (3/6)

C

```
... soma(i,j);... /* i,j:$s0,$s1 */

int soma(int x, int y) { // x,y: $a0,$a1
 int res=op1+op2; return res; // $v0: res
}
```

MIPS

- Pergunta: Porquê utilizar `jr`? Porque não `j`?
- Resposta: A função `soma` pode ser chamada de muitos sítios diferentes. Assim, não podemos regressar para um endereço fixo pré-definido. É preciso disponibilizar um mecanismo para dizer "regressa aqui" !

```
2000 sum: add $v0,$a0,$a1
2004 jr   $ra  # nova instrução
```

Instruções de suporte a funções (4/6)

- Instrução para simultaneamente saltar e fazer a salvaguarda do endereço de retorno: jump and link (`jal`)

- Sem jal:

```
1008 addi $ra,$zero,1016  #$ra=1016
1012 j sum                #goto sum
```

- Com jal:

```
1008 jal sum              # $ra=1012,goto sum
```

- Será que `jal` é imprescindível?
  - A chamada a funções é uma operação muito frequente.
  - Para além disso, com `jal` o programador não precisa de saber onde é que o código vai ser carregado.

Instruções de suporte a funções (5/6)

- A sintaxe do `jal` (jump and link) é semelhante à do `j` (jump):

```
jal label
```

- Na verdade o `jal` deveria ser chamado `laj` (link and jump):
  - Passo 1 (link) - Guarda o endereço da próxima instrução em `$ra`
  - Passo 2(jump) - Salta para a instrução assinalada por label

- Porque é que é guardado o endereço da instrução seguinte em vez da instrução actual?

Instrução de Suporte a Funções (6/6)

- Sintaxe do `jr` (*jump register*):

```
jr register
```

- Em vez de darmos um "*label*" ao *jump*, passamos um registo que contém o endereço para onde queremos saltar.

- Estas duas instruções são muito úteis para chamada de funções:
  - `jal` guarda o endereço de retorno no registo (`$ra`)
  - `jr $ra` salta de volta para o sítio onde a função foi chamada (se entretanto não alterarmos o conteúdo do registo)

Concluindo

- As funções são chamadas com `jal`, e regressam com `jr $ra`.

- As instruções que já aprendemos

  Aritmética: `add, addi, sub, addu, addiu, subu`

  Logicas: `and, andi, or, ori, sll, srl, sla, sra`

  Memória: `lw, sw, lb, sb, lbu, sbu`

  Decisão: `beq, bne, slt, slti, sltu, sltiu`

  Saltos incondicionais: `j, jal, jr`

- Os registos que já conhecemos
  - Todos !

Introdução ao MIPS
- *Datapath* -

Arquitetura de Computadores 2024/2025

Etapas do *Datapath* : Overview

- Problema: A utilização de um único bloco de hardware que "execute a instrução" do início ao fim conduziria a um design complexo e a um desempenho ineficiente.

- Ideia Chave: dividir o processo de "executar uma instrução" num conjunto de etapas, e depois ligar todas estas etapas para criar o datapath completo
  - Etapas menores especializadas são mais simples de desenhar em hardware (dividir o problema em sub-problemas)
  - Podemos optimizar uma determinada etapa sem interferir com as outras (modularidade)

Etapas do *Datapath* (1/6)

- O "*Instruction Set*" do MIPS é composto por instruções muito variadas: Quais serão as etapas que elas têm em comum?

- Etapa 1: *Instruction Fetch*
  - A word de 32-bits na qual a instrução é codificada tem de ser sempre lida a partir da memória (instruction fetch)
  - Para além disso o PC (program counter) tem de ser sempre incrementado para apontar para a instrução seguinte (PC = PC + 4)

Etapas do *Datapath* (2/6)

- Etapa 2: *Instruction Decode*
  - Depois do *fetch*, é necessário fazer a descodificação da instrução e obter os dados associados a cada campo
  - Primeiro, ler o *opcode* para determinar o tipo de instrução e o tamanho dos campos
  - Segundo, ler os dados de todos os registos indicados de forma a definir os operandos
    - Para o `add`, lê-se dois registos
    - Para o `addi`, lê-se um único registo
    - Para o `jal`, não é necessário ler-se registos

Etapas do *Datapath* (3/6)

- Etapa 3: ALU (Unidade de Lógica e Aritmética)
  - Na maior parte das instruções o trabalho efetivo é feito neste nível: aritmética (+, -, *, /), deslocamento, lógica (&, |), comparações (`slt`)
  - E quanto aos loads e stores?
    - `lw  $t0, 40($t1)`
    - Repare que é necessário calcular o endereço final através da adição de 40 (imediato) ao contéudo do registo `$t1`
    - A adição para o cálculo do endereço é feita nesta etapa

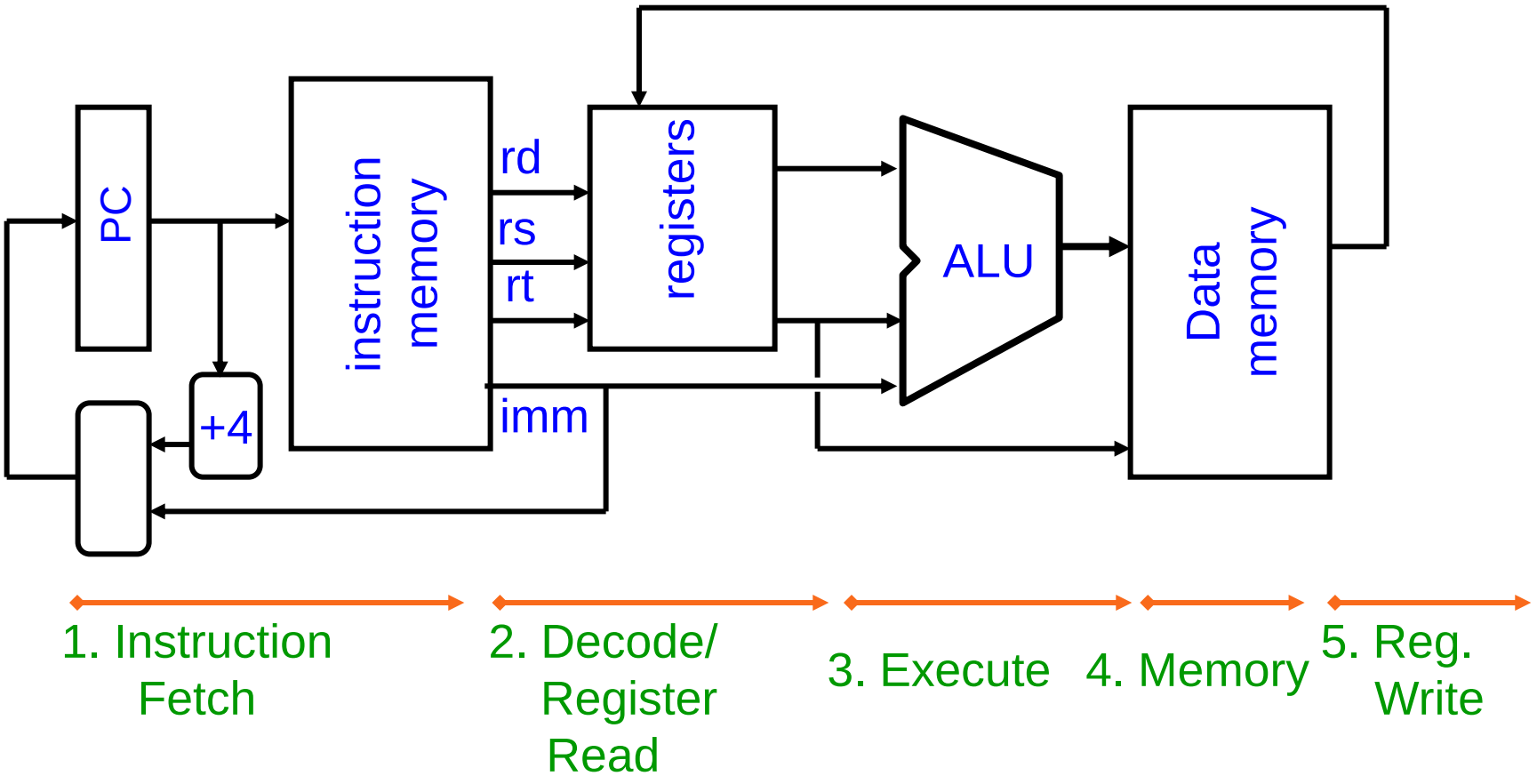
Etapas do Datapath (4/6)

- Etapa 4: *Memory Access*
  - Somente as instruções *load* e *store* é que fazem trocas de informação com a memória (leitura e escrita); todas as outras instruções ficam inativas (idle) durante esta etapa.
  - Este é uma etapa incontornável para a implementação dos *loads* e *stores*. Assim, e apesar das outras instruções não terem este passo, o datapath tem de conter esta etapa.

Etapas do Datapath (5/6)

- Etapa 5: *Register Write*
  - A maioria das instruções escreve o resultado de uma determinada operação num registo destino.
  - exemplos: operações aritméticas e lógicas, deslocamentos, loads, `slt`
  - E quanto aos stores, jumps e branches?
    - Estas instruções não escrevem nenhum resultado num registo destino
    - São instruções que permanecem inativas durante esta etapa.

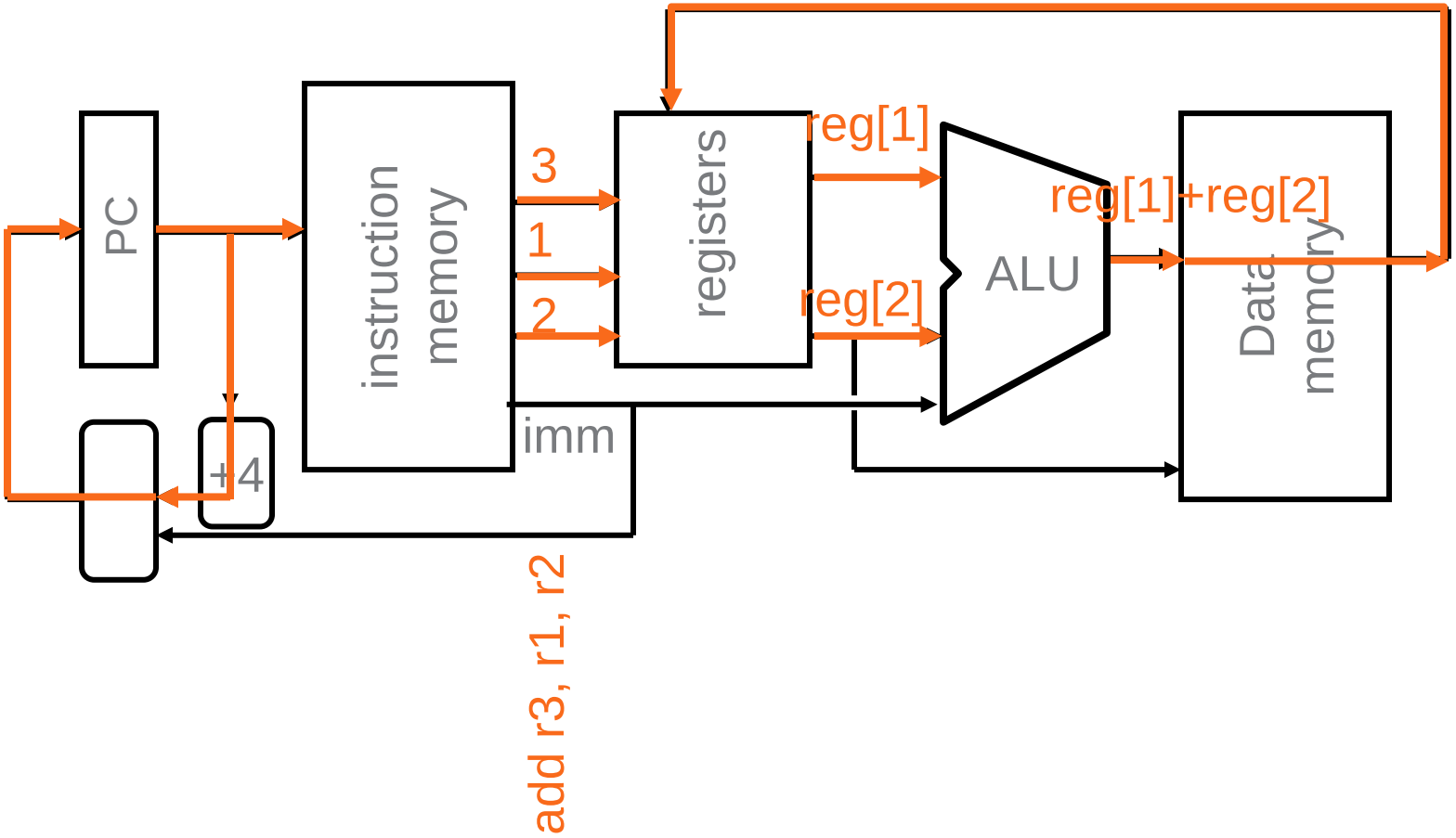
Etapas do Datapath (6/6)



Datapath Walkthroughs (1/3)

- `add $r3,$r1,$r2 # r3 = r1+r2`
  - Etapa 1: instruction fetch, inc. PC
  - Etapa 2: descodificação para determinar que é um `add`. Leitura dos registos `$r1` e `$r2`
  - Etapa 3: soma dos dois valores provenientes da etapa 2
  - Etapa 4: idle (não há qualquer leitura/escrita de memória)
  - Etapa 5: escrita do resultado da etapa 3 no registo `$r3`

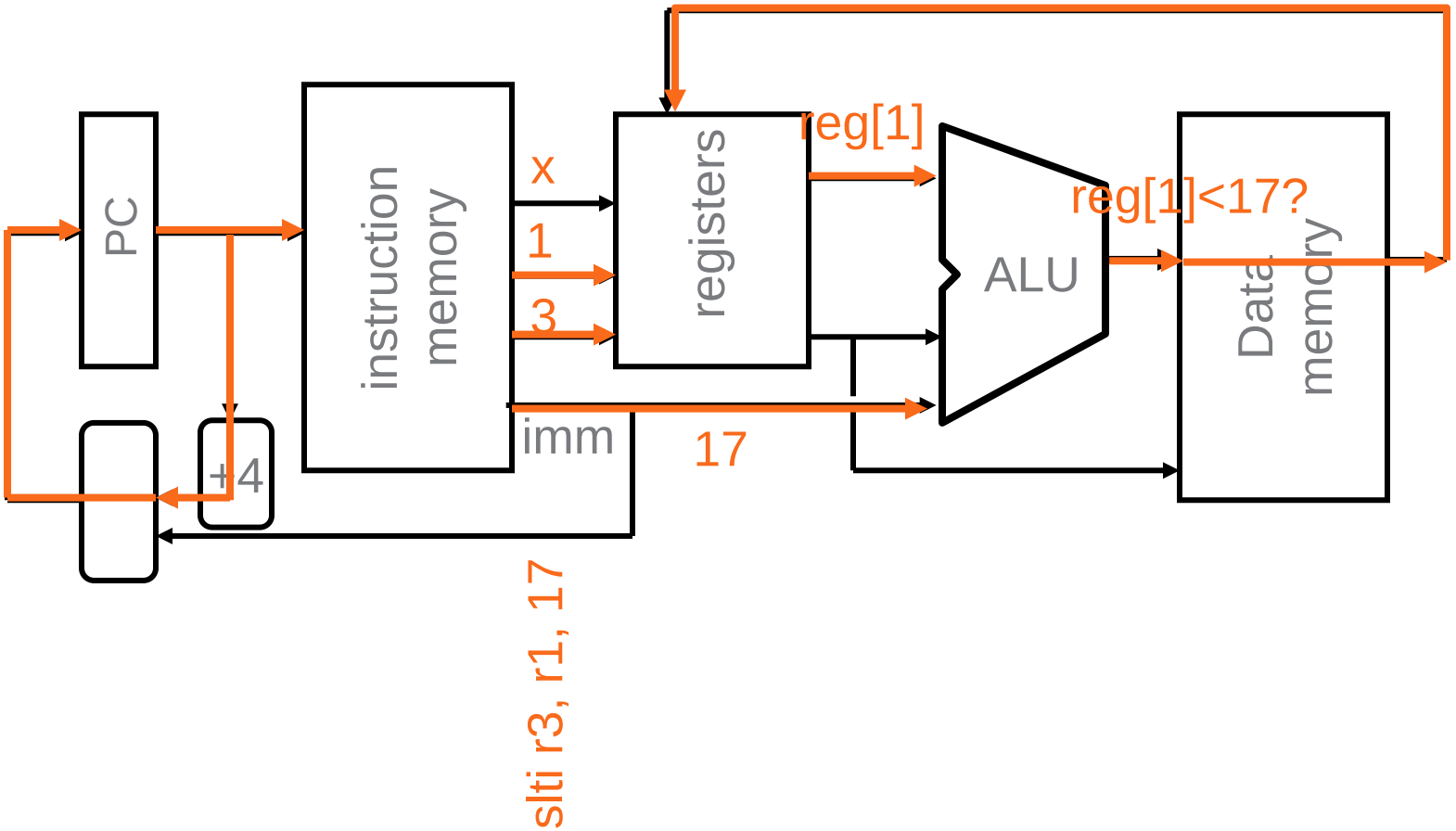
Exemplo: instrução add



Datapath Walkthroughs (2/3)

- `slti  $r3,$r1,17`
  - Etapa 1: fetch da instrução, inc. PC
  - Etapa 2: descodificação para descobrir que é um `slti`. Leitura do registo `$r1`
  - Etapa 3: comparação do valor proveniente da Etapa 2 com o inteiro 17
  - Etapa 4: idle
  - Etapa 5: escrita do resultado da etapa 3 no registo `$r3`

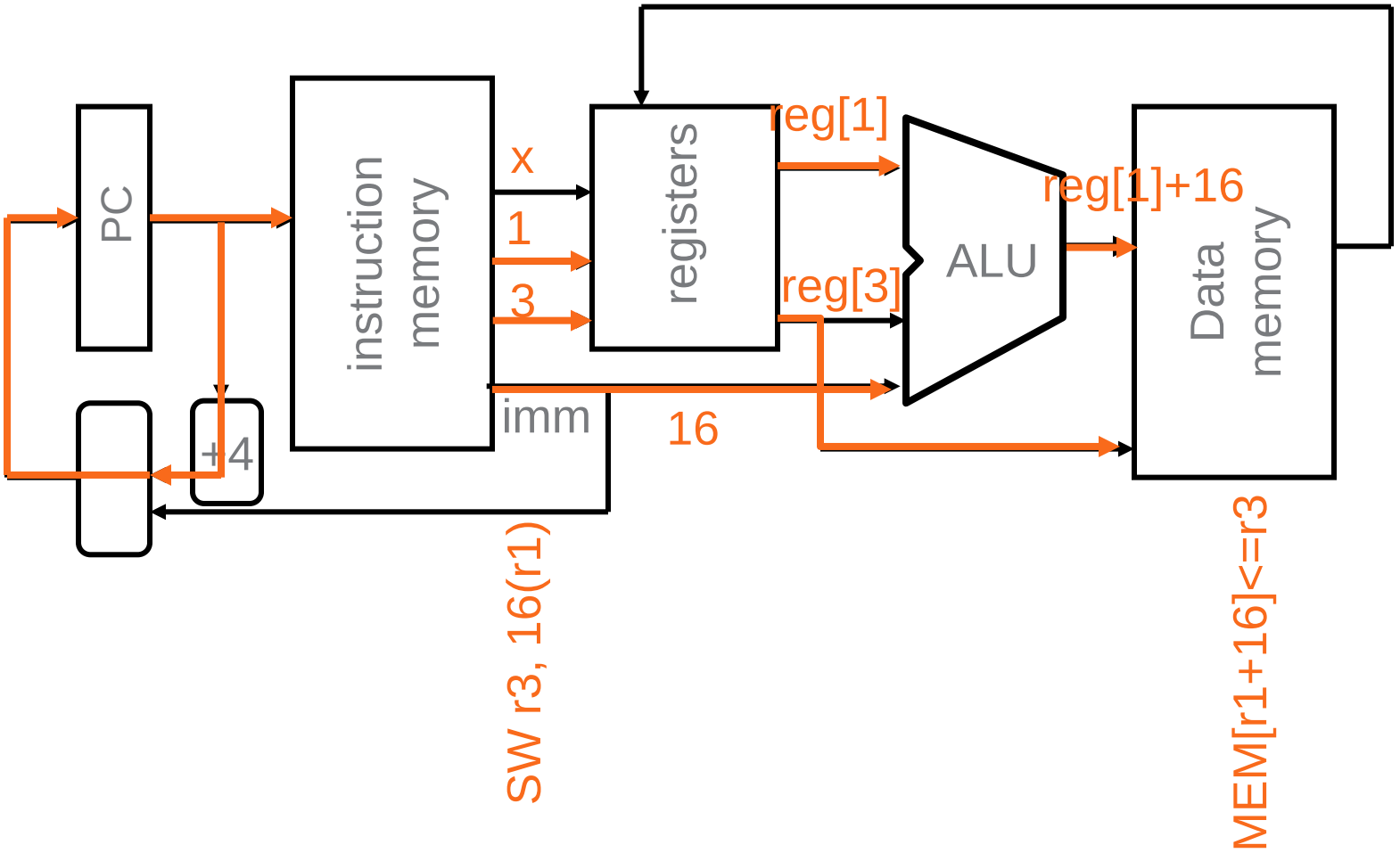
Exemplo: Instrução `slti`



Datapath Walkthroughs (3/3)

- `sw  $r3, 16($r1)`
  - Etapa 1: fetch da instrução, inc. PC
  - Etapa 2: descodificação para saber que é um sw. Leitura dos registos $r1 e $r3
  - Etapa 3: soma de 16 ao valor do registo $r1
  - Etapa 4: escrita do valor que reside no registo $r3 (proveniente da Etapa 2) na posição de memória com o endereço calculado na Etapa 3
  - Etapa 5: idle (não há nada a escrever nos registos)

Exemplo: Instrução sw



Porquê 5 etapas? (1/2)

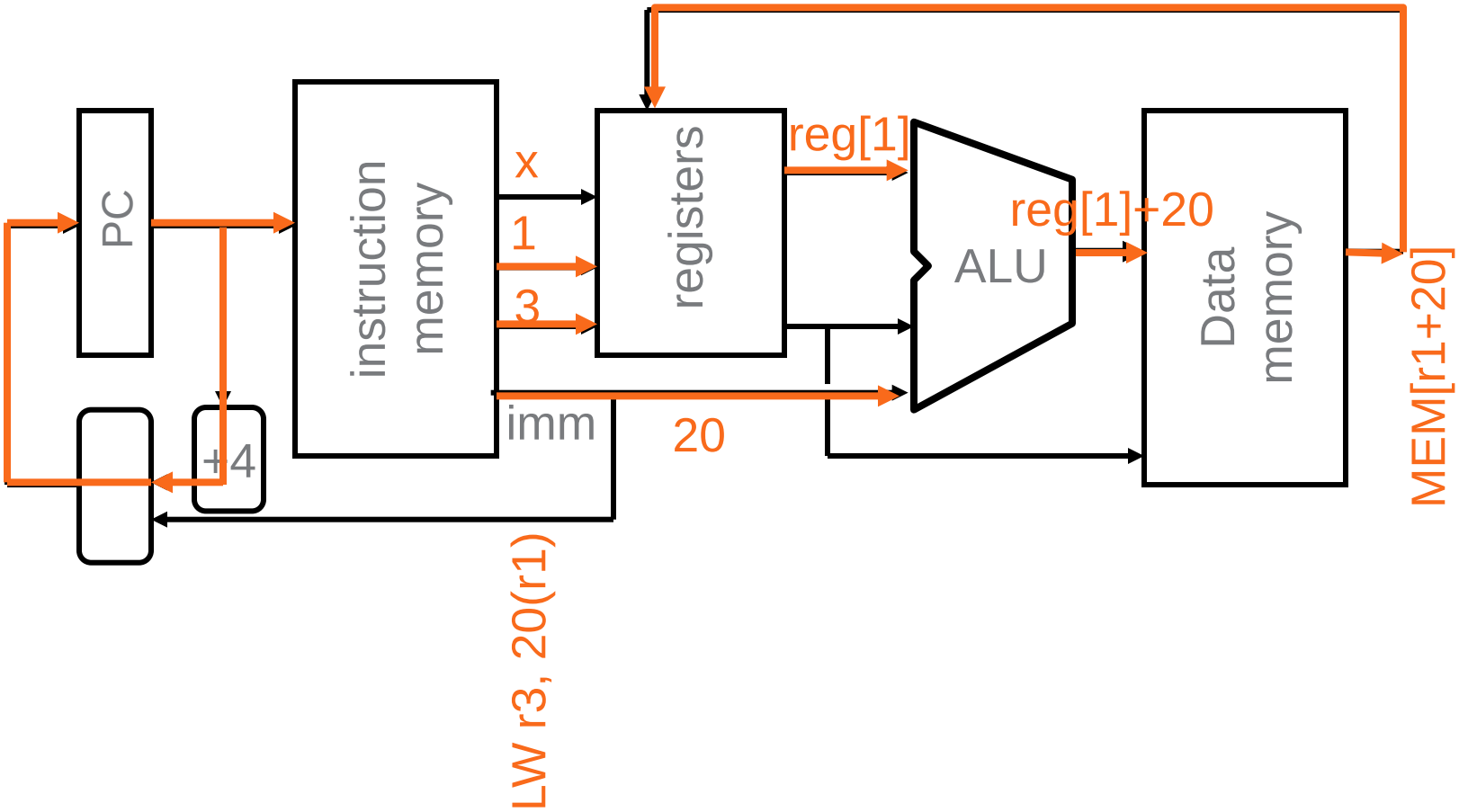
- Poderíamos ter um número diferente de etapas?
  - Sim, há outras arquiteturas que têm um número diferente

- Então porque é que o MIPS tem 5 etapas quando a maior parte das instruções estão inativas em pelo menos um estágio? Quatro não seria sufciente?
  - As cinco etapas são a união de todas as operações necessárias à implementação de todo o Instruction Set.
  - Há uma instrução que está activa nas cinco etapas: o *load*

Porquê 5 etapas? (2/2)

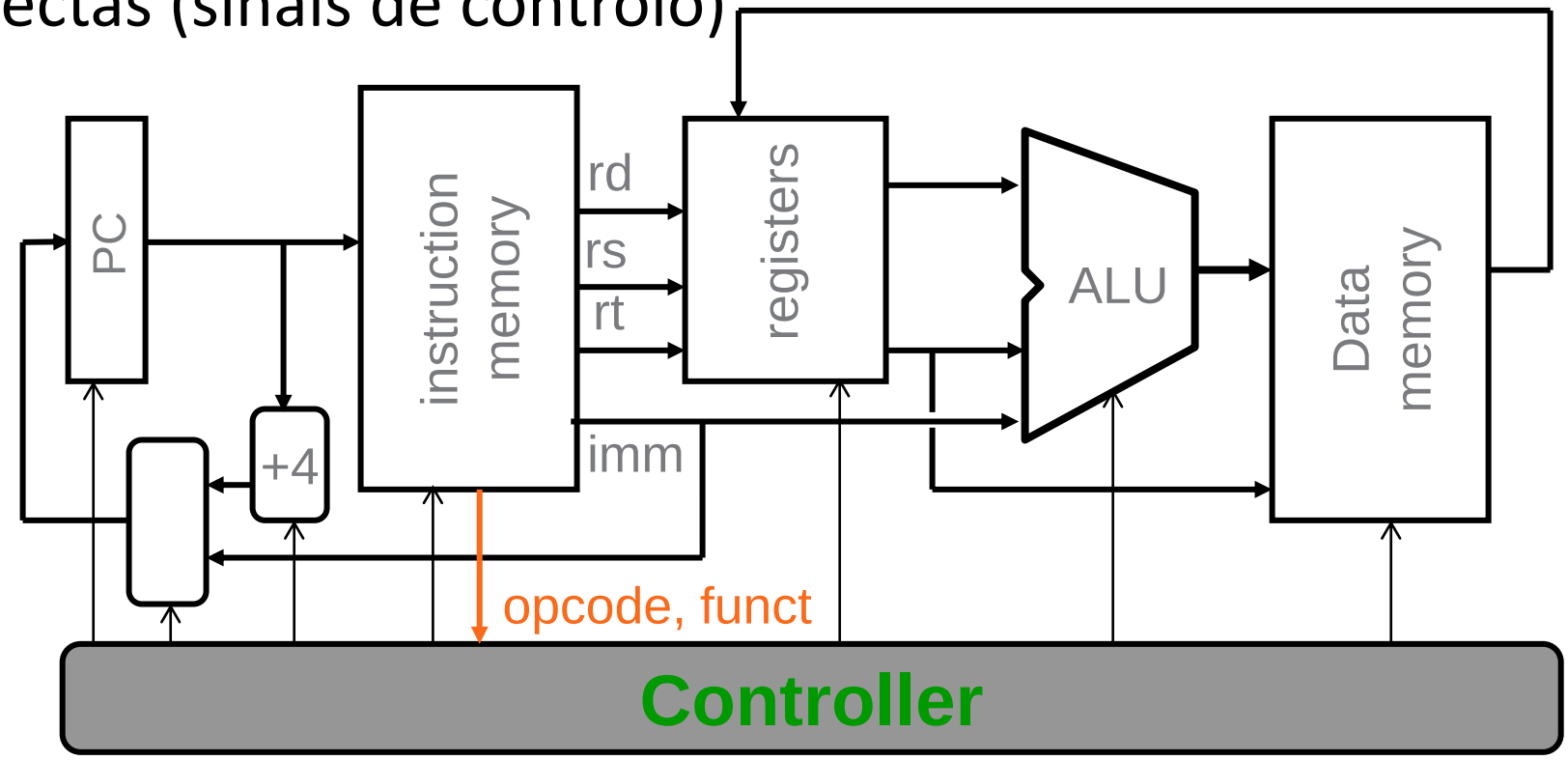
- `lw  $r3, 20($r1)`
  - Etapa 1: fetch da instrução, inc. PC
  - Etapa 2: descodificação para determinar que é um `lw`. Leitura do registo `$r1`
  - Etapa 3: soma 20 ao valor do registo `$r1`
  - Etapa 4: leitura da posição de memória com o endereço calculado no estágio 3
  - Etapa 5: escrita do valor lido no registo `$r3`

Exemplo: instrução `lw`



Sumário - Datapath

- O datapath é definido pelas transferências de dados necessárias à execução da instrução
- O controlador faz acontecer as transferências de dados correctas (sinais de controlo)



Qual é o hardware necessário? (1/2)

- PC: um registo que guarda o endereço de memória onde se encontra a próxima instrução

- Registos de Utilização Geral
  - Usados nas Etapas 2 (Leitura) e 5 (Escrita)
  - MIPS tem 32 registos destes

- Memória
  - Usada nas Etapas 1 (Fetch) e 4 (R/W)
  - Veremos à frente que o sistema de cache tenta tornar estas duas Etapas (quase) tão rápidas como as restantes.

Qual é o hardware necessário? (2/2)

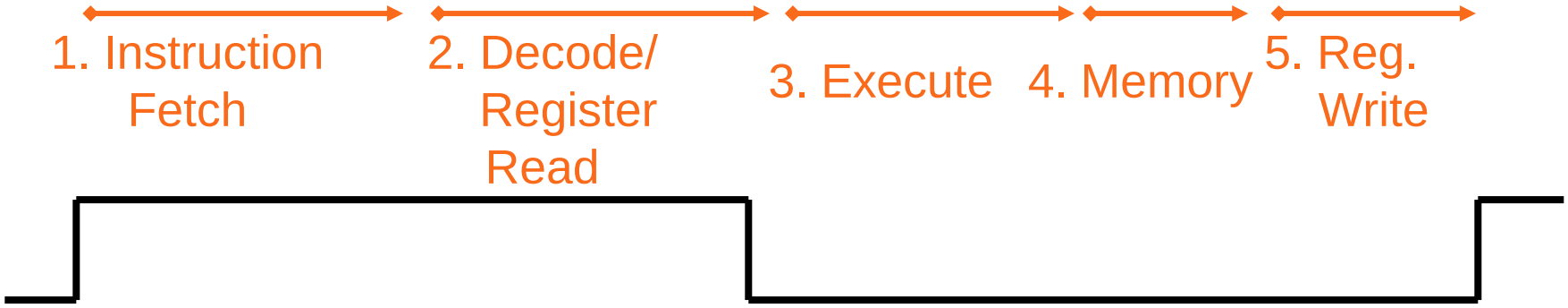
- ALU
  - Usada na Etapa 3
  - Algo que implementa todas as funções necessárias: aritméticas, lógicas, etc.

- Registos Auxiliares
  - Nas implementações em que cada etapa é executada num ciclo de relógio, é muitas vezes necessário utilizar registos auxiliares para guardar resultados intermédios entre etapas, bem como sinais de controlo que viajam de uma etapa para a outra.

CPU clocking (1/2)

Como é que controlamos o fluxo de informação que atravessa o datapath?

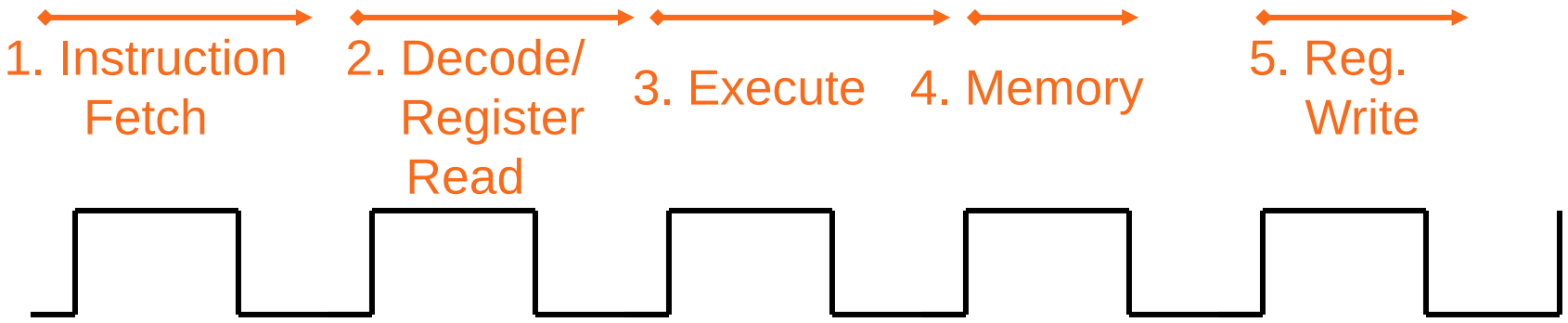
- Single Cycle CPU:Todas as etapas de uma instrução são completadas em um único *longo* ciclo de relógio.



CPU clocking (2/2)

Como é que controlamos o fluxo de informação que atravessa o datapath?

- Multiple-cycle CPU: Cada etapa corresponde a um ciclo de relógio.
  - O período do relógio é igual à duração da etapa mais longa



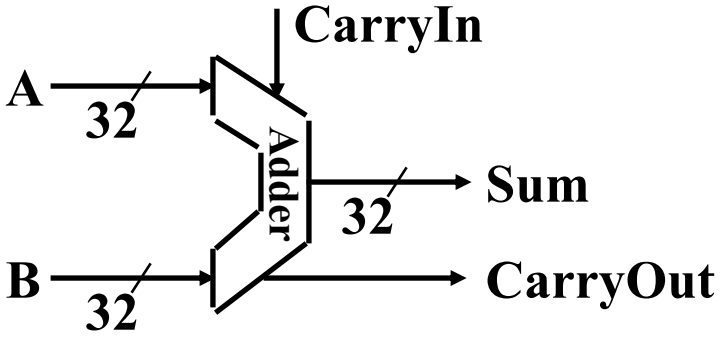
- O multi-cycle tem vantagens em relação ao single cycle:
  - Podemos saltar etapas em que uma determinada instrução está inactiva
  - Podemos implementar mecanismos de sobreposição/pipelining.

Como desenhar um processador: passo-a-passo

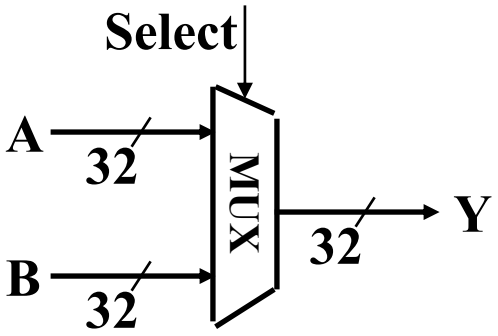
1. Analise o "Instruction Set" a ser implementado (ISA) para obter os requisitos do datapath
   - Cada instrução define um conjunto de transferências entre registos que deve ser suportada pelo datapath
2. Seleccione os componentes de hardware (somadores, mux, etc) que vai utilizar e defina um método de clocking:
   - Single Cycle CPU ou Multi-Cycle CPU
3. Faça a montagem do datapath de forma a ir ao encontro dos requisitos
4. Analise a implementação de cada instrução para determinar os pontos de controlo que afectam a transferência entre registos
5. Construa a lógica de controlo

Building Blocks - Lógica Combinatória

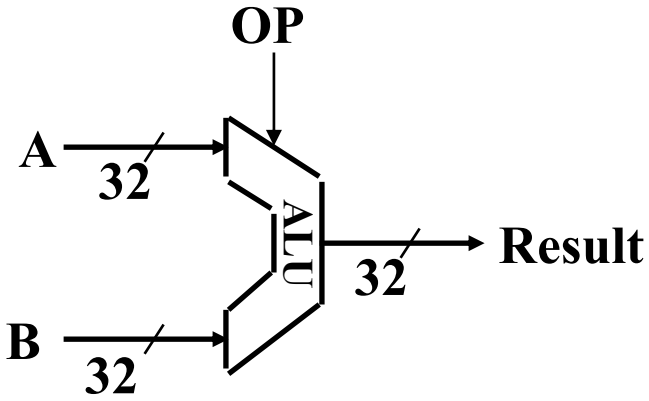
- Somador



- MUX

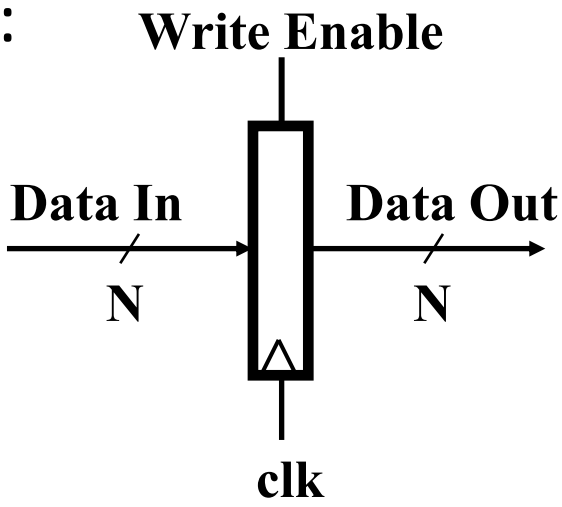


- ALU



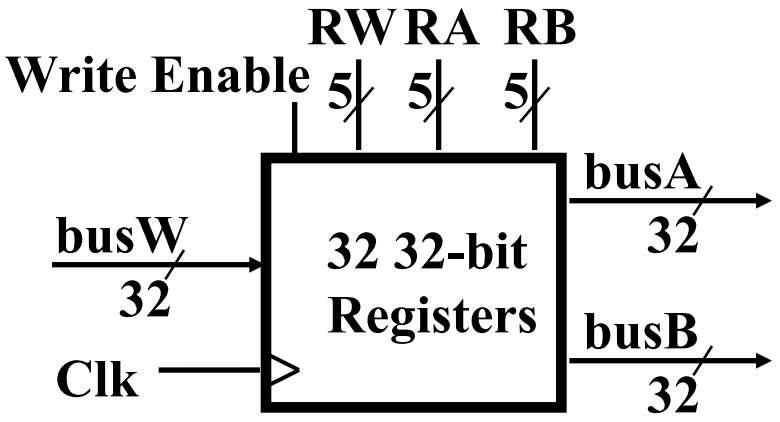
Building Blocks – Armazenamento em registos

- Semelhante a um Flip-Flop D, excepto:
  - Entrada e saída de N-bits
  - Write Enable
- Write Enable:
  - Não asserido (0):
    Data Out não se modifica
  - Asserido (1):
    Data Out fica igual a Data In na vertente positiva do relógio



Armazenamento: Register File

- Consiste em 32 registos:
  - 2 buses de saída de 32-bit (busA and busB)
  - 1 bus de entrada de 32-bit: busW
- O Registo é selecionado por:
  - RA (número) seleciona o registo para busA
  - RB (número) seleciona o registo para busB
  - RW (número) seleciona o registo a ser escrito via busW quando Write Enable é 1



- Repare que é possível fazer leitura e escrita simultaneamente
- Clock input (clk)
  - O clk input só é importante para operações de escrita
  - Ne leitura o "register file" comporta-se como lógica combinacional:
    - RA ou RB válido $\Rightarrow$ busA ou busB válido depois de "access time."

Notas Finais

- O desenho da lógica de controlo é sempre a parte mais complexa na implementação em hardware de uma arquitectura
- Repare que consegue antever como tudo isto pode ser feito usando os conhecimentos que adquiriu em Tecnologias dos Computadores
- O livro discute como fazer a implementação de um single-cycle CPU (Cap. 5.3) e de um multi-cycle CPU (Cap. 5.4)
- Disciplinas avançadas que discutem o desenho de CPUs
  - Arquitetura de Computadores (DEEC)
  - Projeto de Sistemas Digitais (DEEC)
- Mais adiante no semestre vamos assumir uma implementação do tipo multi-cycle e discutir como aumentar o desempenho tirando partido do paralelismo entre instruções

Para saber mais ...

- P&H - Capítulos 2.1, 2.2, 2.3 e 2.6
- P&H - Capítulo 2.9 páginas 95 e 96

Para saber mais ...

- P&H - Capítulos 5.1 e 5.2

- P&H - Capítulos 5.3, 5.4 (implementação de um single-cycle CPU) e 5.5 (implementação de um multi-cycle CPU). Esta matéria não foi ainda discutida com detalhe nas aulas, mas poderá interessar aos mais curiosos.