



Introdução ao MIPS

- Representação de Instruções -

Arquitetura de Computadores 2018/2019

Revisão

- As funções são chamadas com `jal`, e regressam com `jr $ra`.
- Para passar parâmetros/argumentos utilizam-se os registos `$a0`, `$a1`, `$a2` e `$a3`
- Para devolver resultados utilizam-se os registos `$v0` e `$v1`
- A pilha é utilizada para guardar tudo aquilo de que precisamos ... Mas é preciso ter cuidado porque quando um procedimento regressa tem de deixar a pilha exactamente como a encontrou.
- Os procedimentos têm de respeitar a “Convenção de Registos”, ou seja:
 - A função chamante tem de fazer “backup” na pilha de todos os registos voláteis que esteja a utilizar (e depois repô-los)
 - A função chamada tem de repor todos os registos “*saved*” que tenha utilizado
- Os registos que já conhecemos
 - Todos !

Utilização da Pilha (Revisão)

- Compile “à mão a”

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y; }
```

x e y estão em \$a0 e \$a1

sumSquare:

“push”

```
addi $sp,$sp,-8    # espaço na stack 2 words
sw $ra, 4($sp)     # guardar ret addr
sw $a1, 0($sp)     # guardar y
```

```
add $a1,$a0,$zero  # mult(x,x)
jal mult           # chamar mult

lw $a1, 0($sp)     # restaurar y
add $v0,$v0,$a1    # mult()+y
lw $ra, 4($sp)     # obter ret addr
addi $sp,$sp,8     # libertar a stack
jr $ra
```

“pop”

mult: ...

Registos Gerais do MIPS

Constante 0	\$0	\$zero
Reservado para o Assembler	\$1	\$at
Retorno de Valores	\$2-\$3	\$v0-\$v1
Parâmetros	\$4-\$7	\$a0-\$a3
Variáveis Temporárias	\$8-\$15	\$t0-\$t7
Variáveis (<i>saved</i>)	\$16-\$23	\$s0-\$s7
Mais variáveis temporárias	\$24-\$25	\$t8-\$t9
Reservado para o <i>Kernel</i>	\$26-27	\$k0-\$k1
Ponteiro Global	\$28	\$gp
Ponteiro da Pilha	\$29	\$sp
Ponteiro de "Frame"	\$30	\$fp
Endereço de Retorno	\$31	\$ra

Existem ainda: Registos reservados (e.g. PC), e registos de vírgula flutuante

Convenção dos Registos (Revisão) - SAVED

- \$0: **Não Altera**. Sempre 0.
- \$s0-\$s7: **Repor se modificado**. É por isso que são chamados “*saved registers*”. Se a função chamada alterar estes registos deverá restaurá-los antes de regressar à função chamante.
- \$sp: **Repor se modificado**. O *stack pointer* deverá apontar para o mesmo endereço de memória antes e depois da instrução `jal` que passa a execução para a função chamada.
- DICA -- Todos os registos “*saved*” começam por **S**!

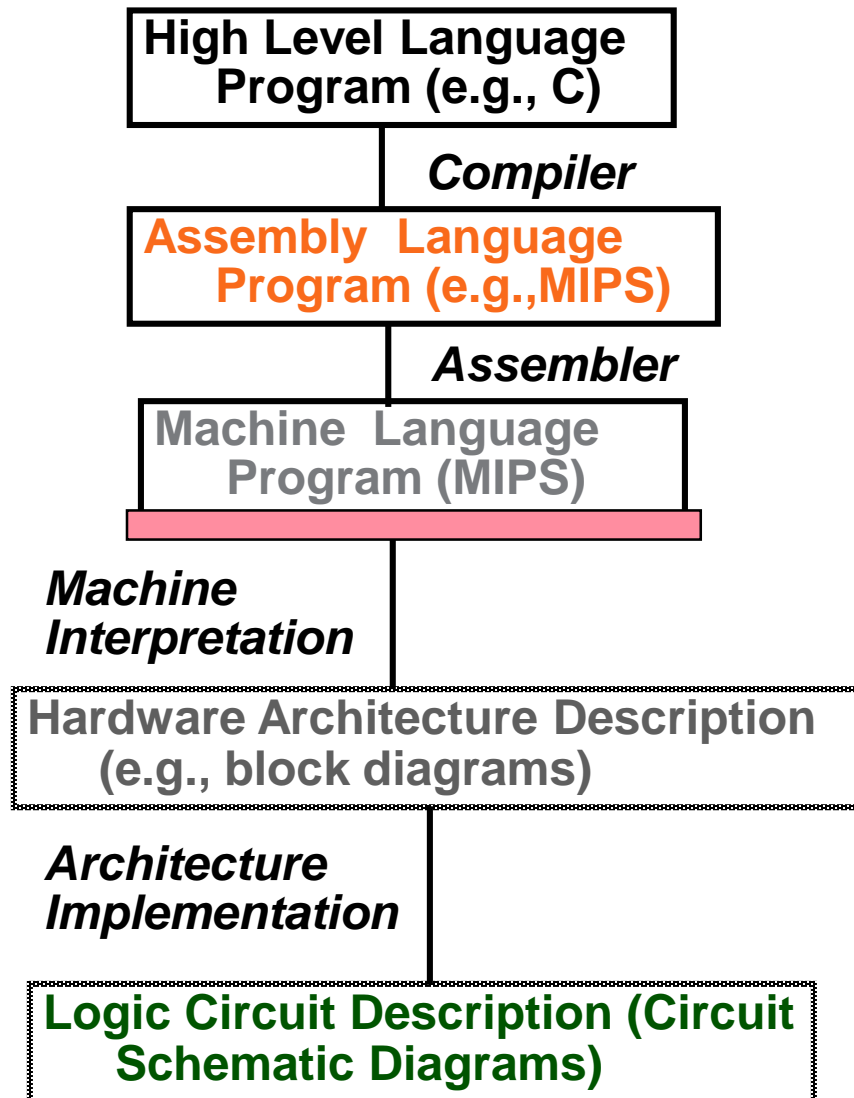
Convenção dos Registos (Revisão) - VOLÁTEIS

- `$ra`: **Podem ser alterados**. A própria instrução `jal` modifica este registo. A função Chamante tem a obrigação de o salvar na pilha antes de passar a execução a outra função.
- `$v0-$v1`: **Podem ser alterados**. Este registo contém os valores de retorno
- `$a0-$a3`: **Podem ser alterados**. Servem para passar parâmetros à função chamada. A função chamada tem que o salvar se precisar de manter estes valores depois da função chamada regressar.
- `$t0-$t9`: **Podem ser alterados**. Por alguma coisa são chamados temporários ...

Convenção de Registos (Revisão)

- Se R é a função chamante, e E é a função chamada, temos em resumo que ...
 - A função R , antes de fazer o jal para E , tem de guardar na pilha todos os registos temporários que tencione usar mais tarde (isto para além de $\$ra$)
 - A função E tem de guardar na pilha todos os registos S (saved) que pretende utilizar, de forma a poder repor os seus valores antes de regressar com jr
 - **Atenção:** *calle R /calle E só precisam de guardar os registos temporários/saved **que utilizem**, e não todos os registos.*

Níveis de representação num computador



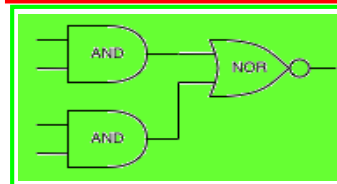
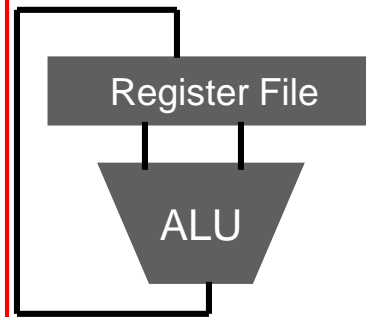
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

PPP

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

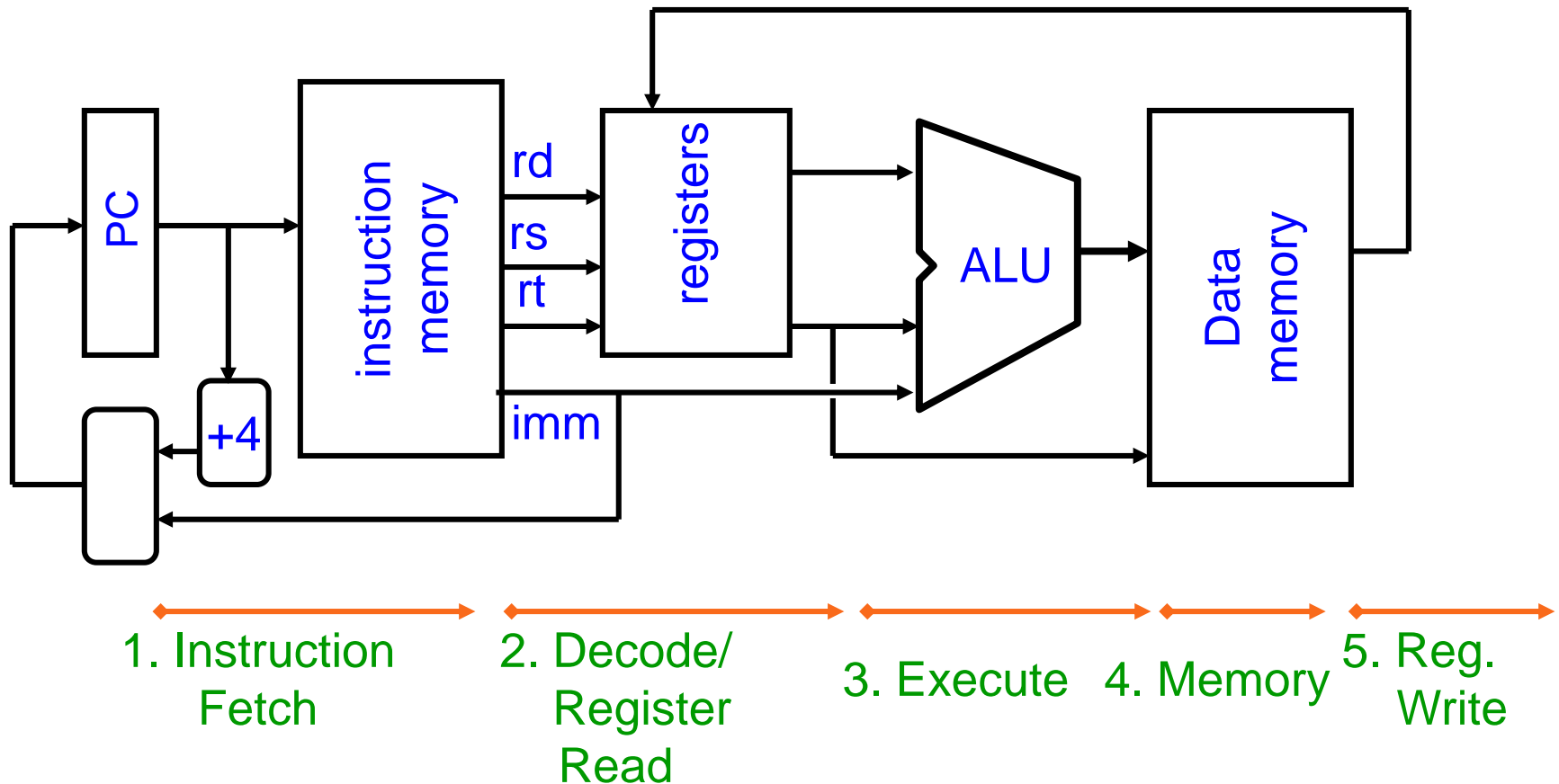
AC



TC



Etapas do Datapath (1/6)



Etapas do Datapath (2/6)

- 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 (3/6)

- Etapa 2: Instruction Decode
 - Depois do fetch, é necessário fazer a decodificação 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 (4/6)

- 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 conteúdo do registo `$t1`
 - A adição para o cálculo do endereço é feita nesta etapa

Etapas do Datapath (5/6)

- 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 (6/6)

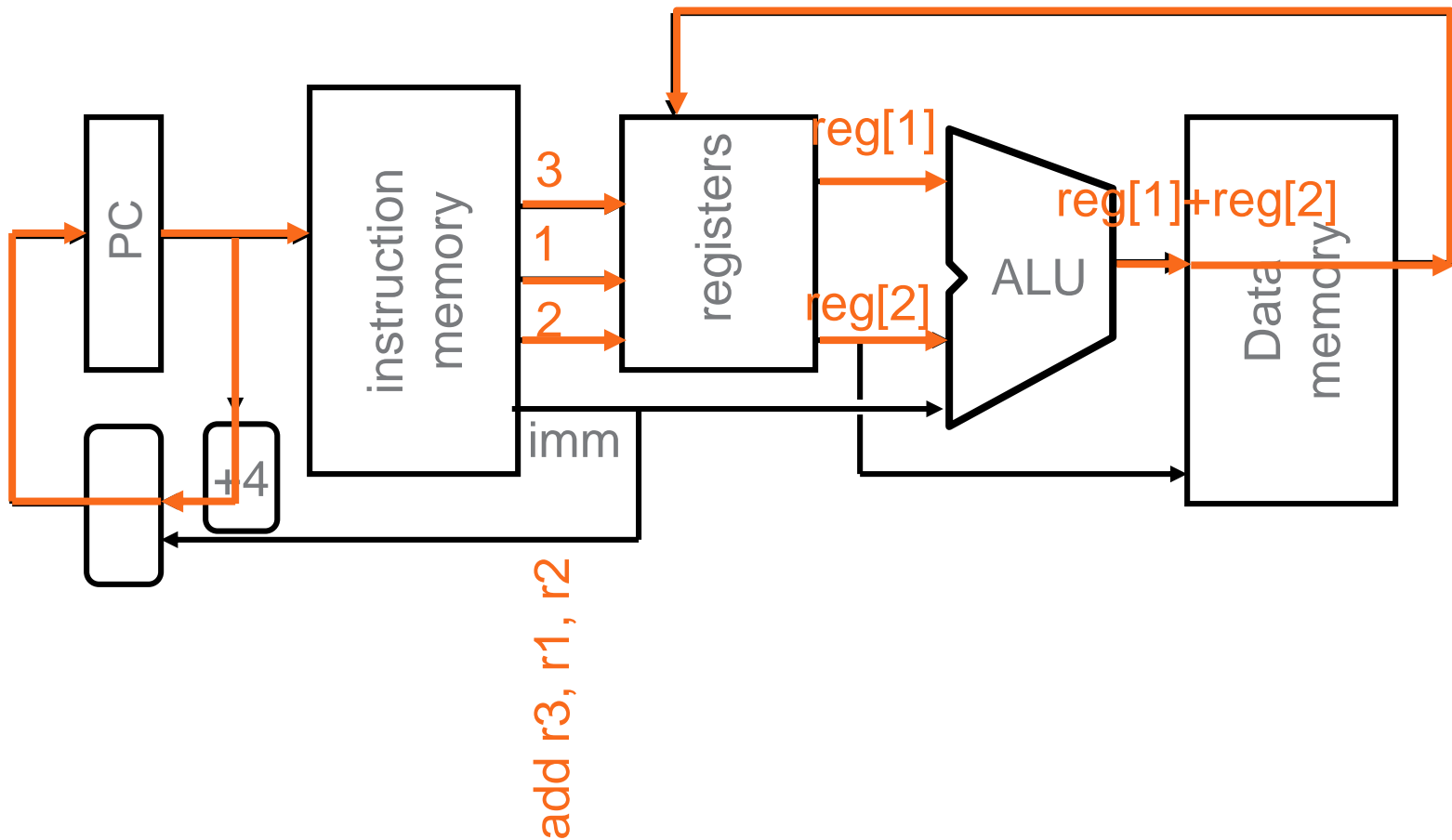
- Etapa 5: Register Write
 - A maioria das instruções escreve o resultado de uma determinada operação num registo destino.
 - exemplos: operações aritméticas e lógicas, deslocamentos, loads, `slt`
 - E quanto aos stores, jumps e branches?
 - Estas instruções não escrevem nenhum resultado num registo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

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

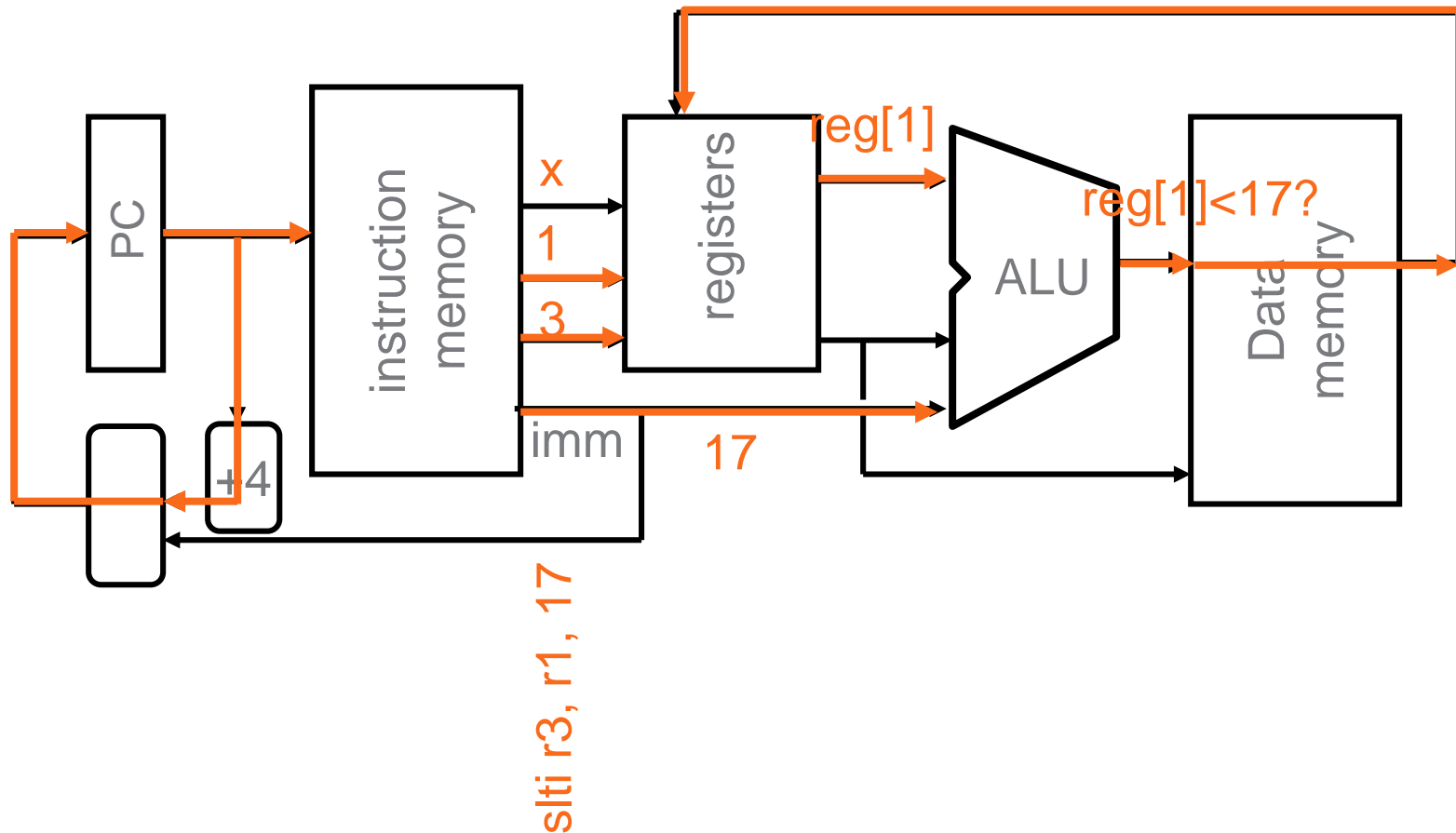


Datapath Walkthroughs (2/3)

- `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`

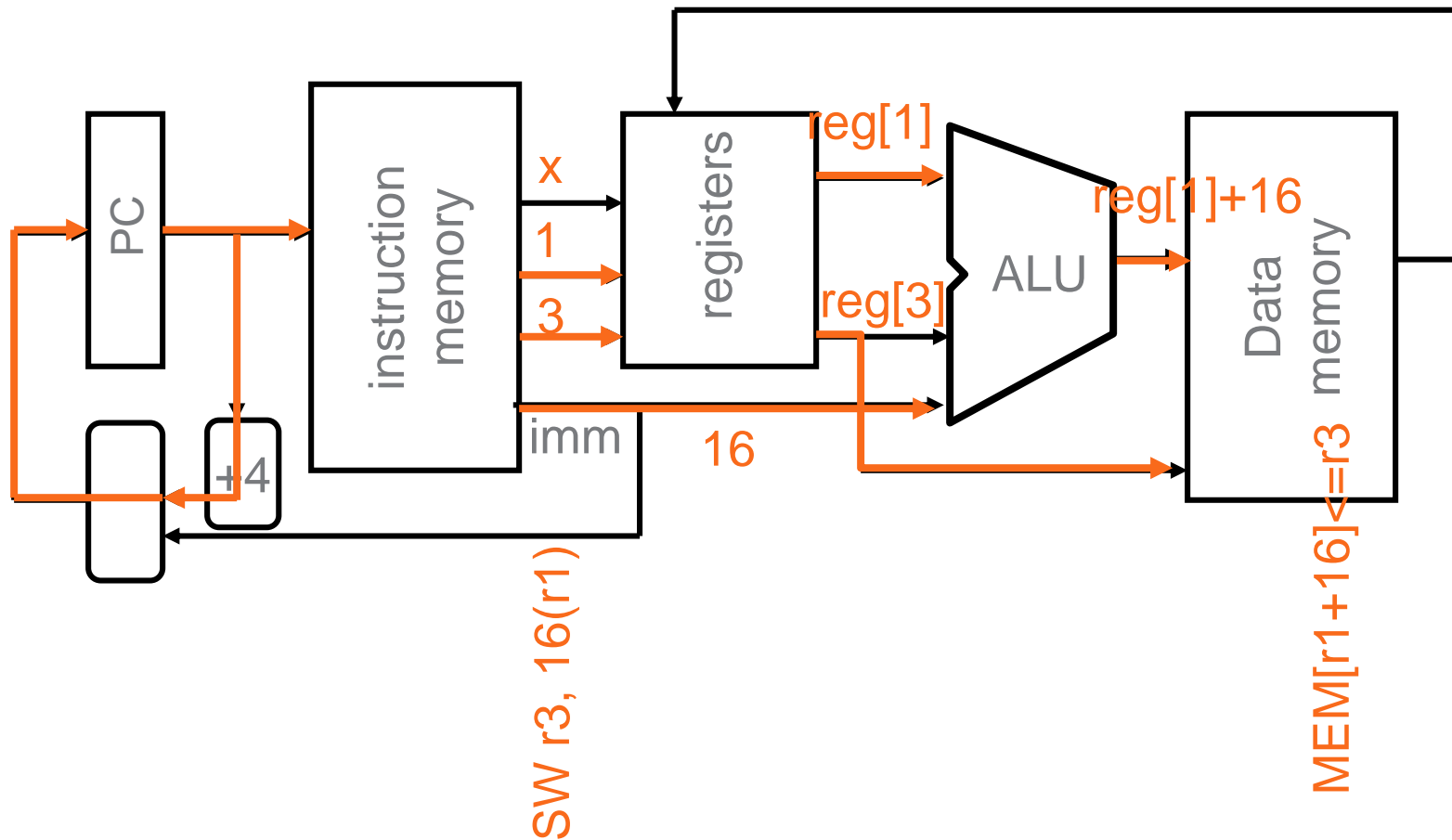


Datapath Walkthroughs (3/3)

- `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)

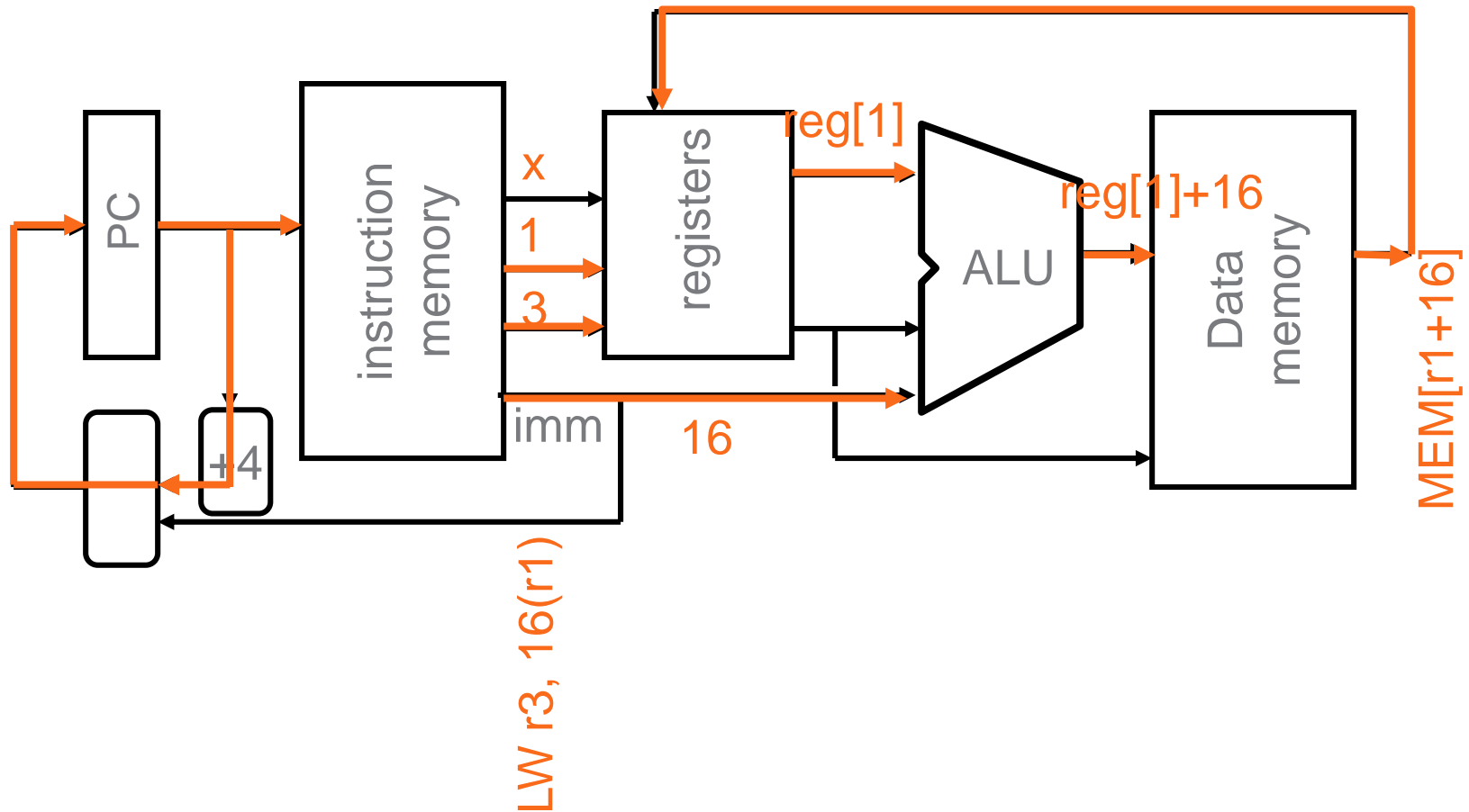
- Poderíamos ter um número diferente de etapas?
 - Sim, há outras arquiteturas que têm um número diferente
- Então porque é que o MIPS tem 5 etapas quando a maior parte das instruções estão inativas em pelo menos um estágio? Quatro não seria suficiente?
 - As cinco etapas são a união de todas as operações necessárias à implementação do Instruction Set.
 - Há uma instrução que está activa nas cinco etapas: o load

Porquê 5 etapas? (2/2)

- $lw \quad \$r3, \ 16(\$r1)$
 - Etapa 1: fetch da instrução, inc. PC
 - Etapa 2: descodificação para determinar que é um lw . Leitura do registo $\$r1$
 - Etapa 3: soma 16 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 $\$r3$

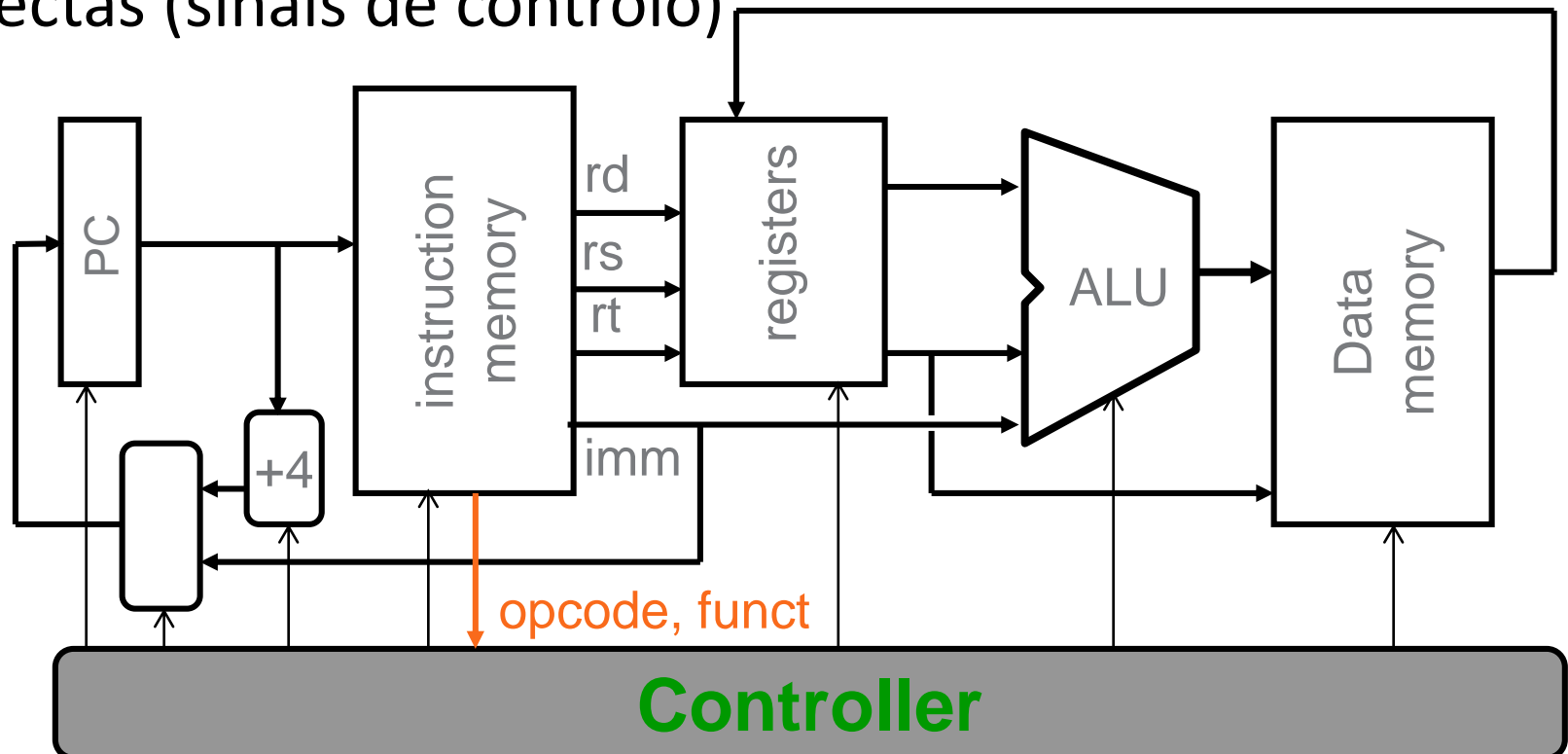


Exemplo: instrução lw



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)

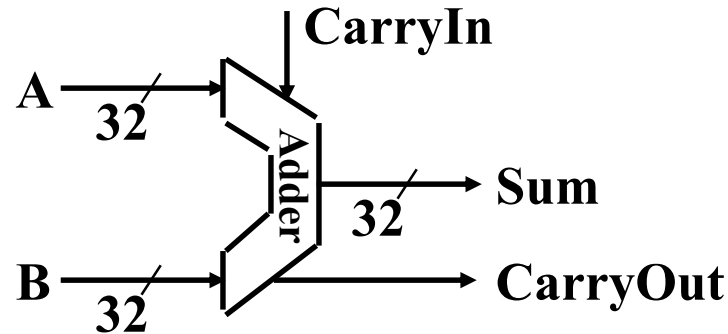
- PC: um registo que guarda o endereço de memória onde se encontra a próxima instrução
- Registos de Utilização Geral
 - Usados nas Etapas 2 (Leitura) e 5 (Escrita)
 - MIPS tem 32 registos destes
- Memória
 - Usada nas Etapas 1 (Fetch) e 4 (R/W)
 - Veremos à frente que o sistema de cache tenta tornar estas duas Etapas (quase) tão rápidas como as restantes.

Qual é o hardware necessário? (2/2)

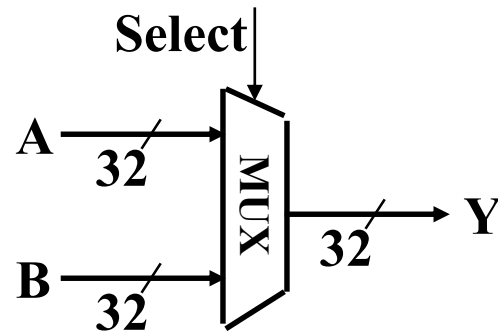
- ALU
 - Usada na Etapa 3
 - Algo que implementa todas as funções necessárias: aritméticas, lógicas, etc.
- Registos Auxiliares
 - Nas implementações em que cada etapa é executada num ciclo de relógio, é muitas vezes necessário utilizar registos auxiliares para guardar resultados intermédios entre etapas, bem como sinais de controlo que viajam de uma etapa para a outra.

Building Blocks - Lógica Combinatória

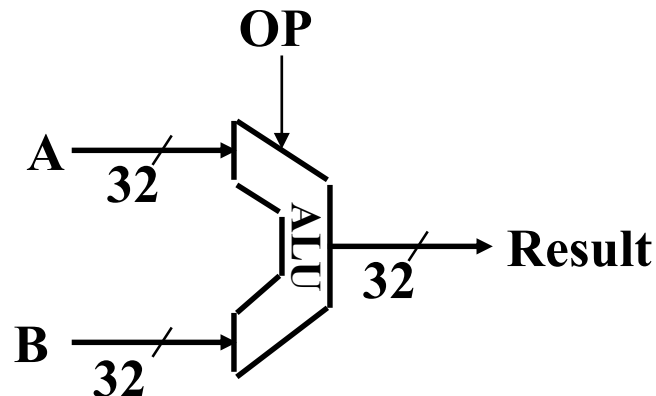
- Somador



- MUX



- ALU



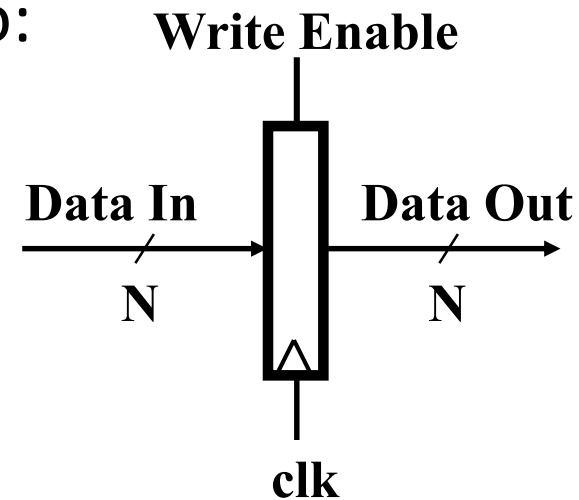
Building Blocks – Armazenamento em registos

– Semelhante a um Flip-Flop D, excepto:

- Entrada e saída de N-bits
- Write Enable

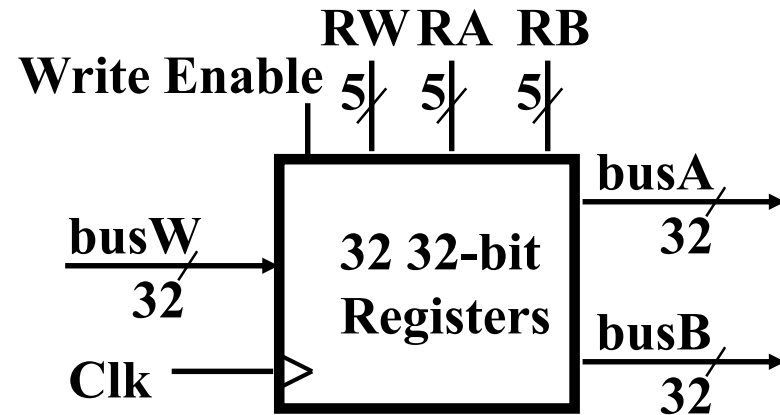
– Write Enable:

- 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

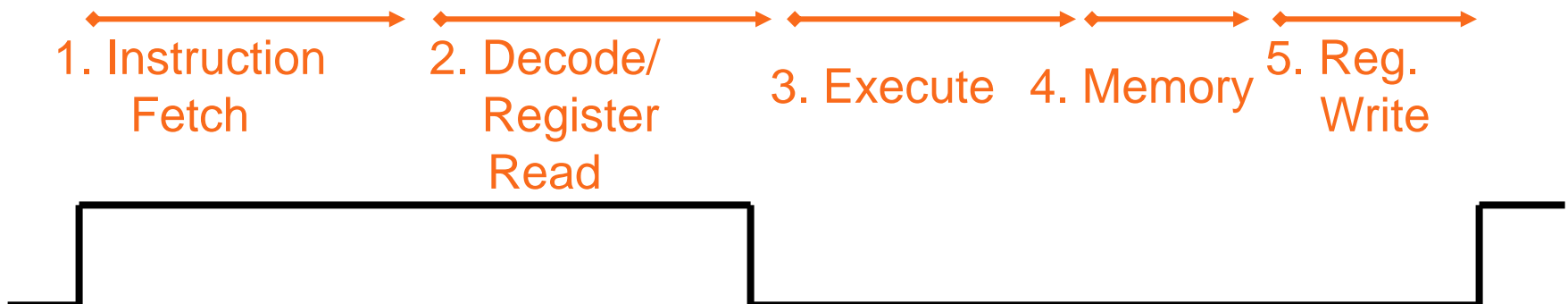
- Consiste em 32 registos:
 - 2 buses de saída de 32-bit (busA and busB)
 - 1 bus de entrada de 32-bit: busW
- O Registo é selecionado por:
 - RA (número) seleciona o registo para busA
 - RB (número) seleciona o registo para busB
 - RW (número) seleciona o registo a ser escrito via busW quando Write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



CPU clocking (1/2)

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

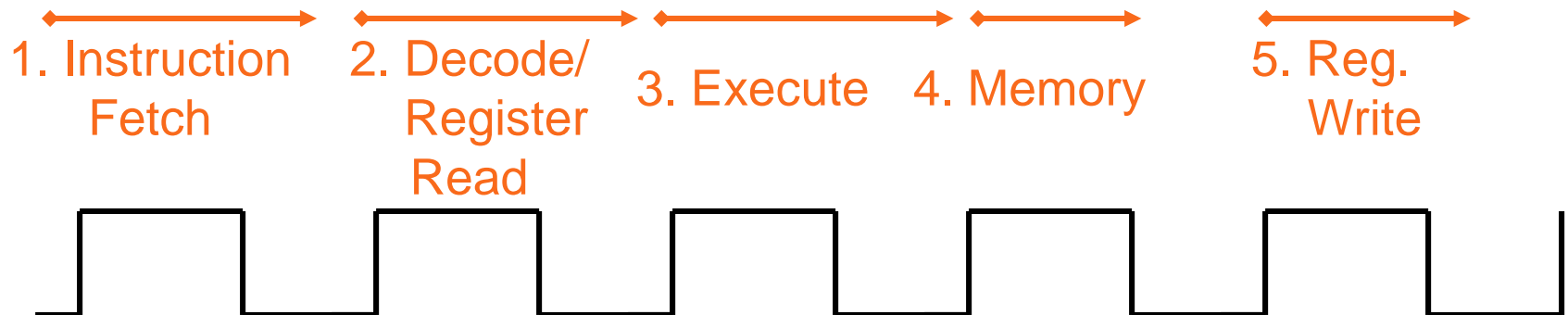
- Single Cycle CPU: Todas as etapas de uma instrução são completadas em um único longo ciclo de relógio.



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

Ideia Brilhante: O conceito de Stored-Program

- Os computadores baseiam-se em 2 princípios chave:
 - 1) As instruções são representadas através de “*bitstrings*”/ padrões de bits - podemos pensar nas instruções como números.
 - 2) Assim, programas inteiros podem ser armazenados em memória para serem lidos ou escritos de forma semelhante ao que acontece com os dados.
- VANTAGEM: Simplifica o SW/HW dos computadores:
 - A tecnologia de memória para dados é usada também para programas

Consequência 1: Tudo funciona por endereços

- Como tanto as instruções como os dados são armazenados em memória, tudo é referenciado por endereços: instruções, dados, *words*, etc.
- Os ponteiros do C são simplesmente endereços de memória — isto permite-nos apontar para qualquer coisa o que pode conduzir a bugs difíceis de apanhar
- O MIPS tem um registo, o “[Program Counter](#)” ([PC](#)), que indica a próxima instrução a ser executada.
- Os “*branches*” e os “*jumps*” modificam a sequência de execução através de escritas no PC

Consequência 2: Binary Compatibility

- Os programas são normalmente distribuídos em binário por questões de simplicidade de instalação e protecção da propriedade intelectual:
 - O programa fica vinculado a um determinado *instruction set*
 - Diferentes versões para diferentes arquitecturas (Macintosh, PC)
 - A comunidade “*open source*” muitas vezes disponibiliza as fontes (rpm vs build)
- As novas máquinas querem simultaneamente correr velhos programas (“*binaries*”) bem como novos programas compilados com novas instruções
- Isto obriga a haver “*backward compatibility*” entre os vários *instruction sets* (e.g. Intel)

As instruções como números binários (1/2)

- No MIPS a manipulação de dados é feita com base em *words* (blocos de 32-bits):
 - Cada registo é uma *word*
 - Tanto lw e sw transacionam com a memória uma *word* de cada vez.
- Então como será que devemos representar instruções em binário?
 - A filosofia do MIPS (RISC) é baseada na simplicidade: assim, se os dados estão em *words*, é conveniente colocar as instruções também em *words*.
- 1 instrução => 1 *word* em memória

As instruções como números binários (2/2)

- Como uma word tem 32 bits, dividimos a word que representa uma instrução em partes designadas por “campos”.
- Cada “campo” diz ao processador algo sobre a instrução em causa.
- Poderíamos definir “campos” diferentes para instruções diferentes, no entanto, isso contraria a filosofia do MIPS de simplicidade e “standardização”.
- O MIPS tem somente três tipos de instruções, obedecendo cada tipo à mesma organização em termos de “campos”:
 - **formato I**: usado para codificar instruções com imediatos (excepto os `shifts`), os `lw` e `sw` (em que o offset conta como um imediato), e os “branches” (`beq` e `bne`),
 - **formato J**: usado para o `j` e `jal`
 - **formato R**: usado para todas as outras instruções

Instruções formato R (1/3)

- Tem seis “**campos**” distintos com o seguinte número de bits: $6 + 5 + 5 + 5 + 5 + 6 = 32$

6	5	5	5	5	6
---	---	---	---	---	---

◆ Cada campo tem um nome/sigla:

opcode	rs	rt	rd	shamt	funct
---------------	-----------	-----------	-----------	--------------	--------------

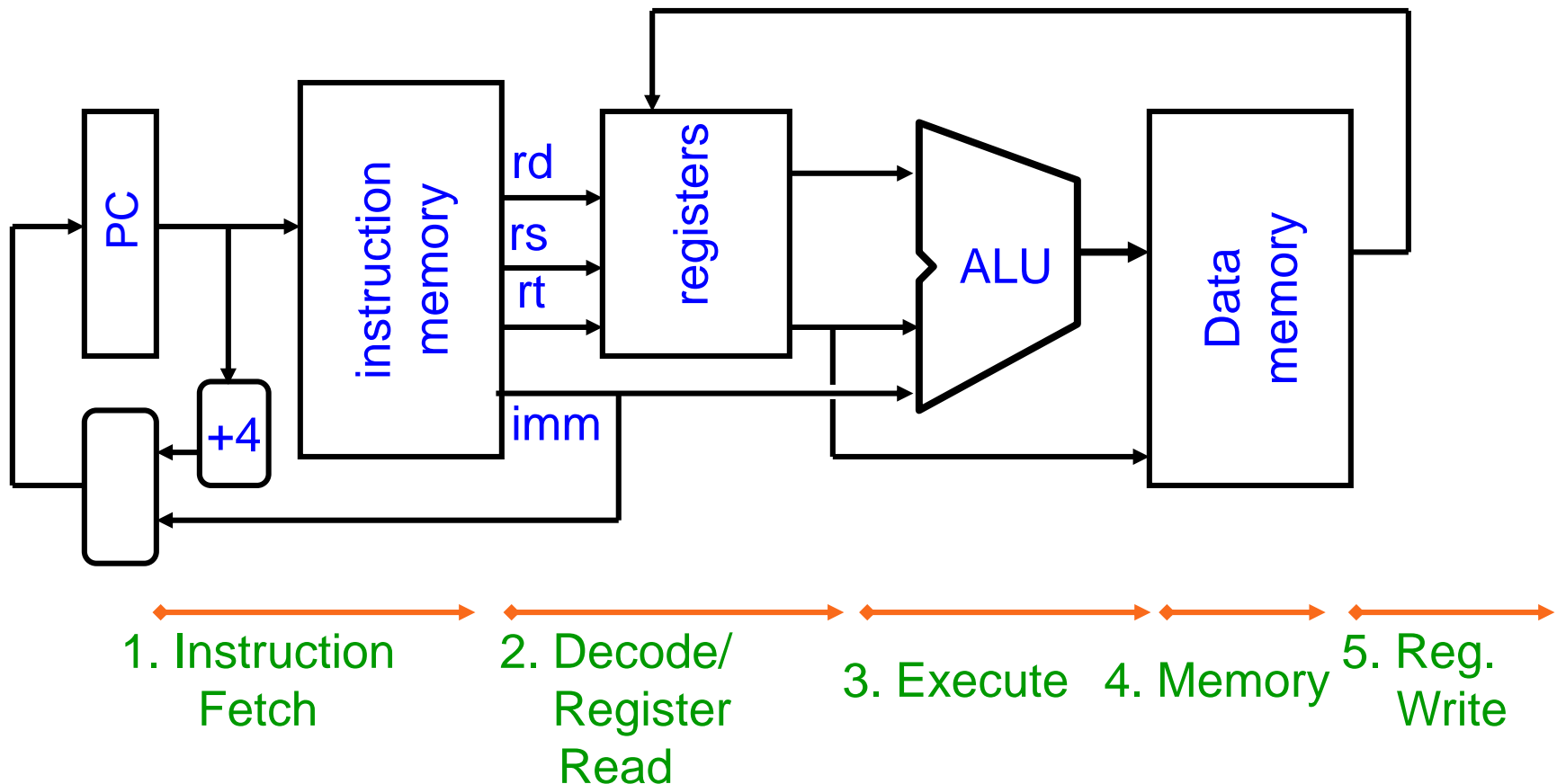
◆ Os campos “r” normalmente especificam registos

- rs (Source Register): especifica o primeiro operando
- rt (Target Register): especifica o segundo operando
- rd (Destination Register): especifica o registo que recebe o resultado

Nota: Cada campo tem 5 bits permitindo distinguir 32 entidades (bate certo?)



Motivação para o Formato das Instruções



Instruções formato R (2/3)

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

- O campo opcode especifica parcialmente a instrução.
- O campo funct é combinado com opcode para definir exactamente a instrução (um add, sub, etc)
- No caso das instruções do tipo R o campo opcode é sempre zero. Assim, a instrução é definida unicamente pelo conteúdo de `funct`.

Instruções formato R (2/3)

opcode	rs	rt	rd	shamt	funct
---------------	-----------	-----------	-----------	--------------	--------------

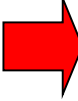
- 
 - Questões pertinentes:
 - Porque é que `opcode` e `funct` não são contíguos formando um único campo de 12 bits?
 - Porque é que as instruções de tipo R têm campo `opcode`?
 - Resposta: Vamos ver isto melhor mais à frente ... Mas a razão é mais uma vez simplicidade e uniformidade da arquitectura.
 - O campo `shamt` indica o deslocamento a ser feito pelas instruções `sllr`, `sll` e `sar`. Este campo está a 0 em todas as instruções R que não representem operações de shift.
 - Repare que os campos `rs`, `rt`, `rd` e `shamt` só têm 5 bits, o que significa que só podem representar números inteiros entre 0 e 31.
 - Será isto suficiente?

Tabela de *Opcodes* e Funções

Mnemonic	Meaning	Type	Opcode	Funct
add	Add	R	0x00	0x20
addi	Add Immediate	I	0x08	NA
addiu	Add Unsigned Immediate	I	0x09	NA
addu	Add Unsigned	R	0x00	0x21
and	Bitwise AND	R	0x00	0x24
andi	Bitwise AND Immediate	I	0x0C	NA
beq	Branch if Equal	I	0x04	NA
bne	Branch if Not Equal	I	0x05	NA
div	Divide	R	0x00	0x1A
divu	Unsigned Divide	R	0x00	0x1B
j	Jump to Address	J	0x02	NA
jal	Jump and Link	J	0x03	NA
jr	Jump to Address in Register	R	0x00	0x08
lbu	Load Byte Unsigned	I	0x24	NA
lhu	Load Halfword Unsigned	I	0x25	NA
lui	Load Upper Immediate	I	0x0F	NA
lw	Load Word	I	0x23	NA
mfhi	Move from HI Register	R	0x00	0x10
mflo	Move from LO Register	R	0x00	0x12

Mnemonic	Meaning	Type	Opcode	Funct
mult	Multiply	R	0x00	0x18
multu	Unsigned Multiply	R	0x00	0x19
nor	Bitwise NOR (NOT-OR)	R	0x00	0x27
xor	Bitwise XOR (Exclusive-OR)	R	0x00	0x26
or	Bitwise OR	R	0x00	0x25
ori	Bitwise OR Immediate	I	0x0D	NA
sb	Store Byte	I	0x28	NA
sh	Store Halfword	I	0x29	NA
slt	Set to 1 if Less Than	R	0x00	0x2A
slti	Set to 1 if Less Than Immediate	I	0x0A	NA
sltiu	Set to 1 if Less Than Unsigned Immediate	I	0x0B	NA
sltu	Set to 1 if Less Than Unsigned	R	0x00	0x2B
sll	Logical Shift Left	R	0x00	0x00
srl	Logical Shift Right	R	0x00	0x02
sra	Arithmetic Shift Right	R	0x00	0x03
sub	Subtract	R	0x00	0x22
subu	Unsigned Subtract	R	0x00	0x23
sw	Store Word	I	0x2B	NA

Exemplo formato R (1/2)

- Instrução MIPS:

`add $8, $9, $10`

`opcode = 0` (veja a tabela no livro)

`funct = 32` (veja a tabela no livro)

`rd = 8` (destino)

`rs = 9` (primeiro *operando*)

`rt = 10` (segundo *operando*)

`shamt = 0` (não é um shift)

Exemplo formato R (2/2)

- Instrução MIPS:

add \$8, \$9, \$10

Representação em decimal do valor de cada campo:

0	9	10	8	0	32
---	---	----	---	---	----

Representação em binário:

000000	01001	01010	01000	00000	100000
--------	-------	-------	-------	-------	--------

hex

Representação em hexa: 012A 4020_{hex}

Representação em decimal: 19,546,144_{ten}

- Isto é uma Instrução em Linguagem Máquina ([Machine Language Instruction](#))

Instruções formato I (1/4)

- E quanto às instruções com valores imediatos (constantes)?
 - Um campo de 5-bits só pode representar valores entre 0 e 31: normalmente os valores imediatos são bastante maiores do que 31
 - Idealmente o MIPS só teria um formato de instrução, mas infelizmente isso não é possível. Assim temos de aceitar compromissos (é por isso que somos engenheiro(a)s ;^)
- Vamos tentar definir um novo formato que permita representar imediatos e simultaneamente seja o mais consistente possível com o formato R:
 - Repare que as instruções com imediatos envolvem no máximo 2 registos (e nunca 3).

Instruções formato I (2/4)

- Vamos definir uma divisão em “campos” com o seguinte número de bits: $6 + 5 + 5 + 16 = 32$ bits

6	5	5	16
---	---	---	----

◆ O nome dos campos são:

opcode	rs	rt	imediato
---------------	-----------	-----------	-----------------

◆ **Ideia Chave:** Repare que só o último campo é inconsistente com o formato R. E ainda mais importante: o `opcode`, que define a instrução, está ainda no mesmo sítio.

- Começa a perceber agora o porquê dos campos `opcode` e `funct` nas instruções R?

Instruções formato I (3/4)



- O que significam estes campos
 - opcode: o mesmo que vimos para as instruções R com a excepção de que agora não existe um campo `funct`. O campo `opcode` define sozinho de que instrução se trata.
 - Isto também esclarece o facto das instruções R terem dois campos de 6-bits para identificar a instrução, em vez de um único campo de 12-bits. É a forma de manter a coerência entre diferentes formatos, deixando 16 bits contíguos para acomodar imediatos no caso das instruções I.
 - rs: especifica um registo operando (no caso de existir)
 - rt: especifica o registo que vai receber o resultado (target register).

Instruções formato I (4/4)

- O campo imediato:
 - O campo imediato tem 16 bits e pode representar 2^{16} valores diferentes
 - Esta gama é suficientemente ampla para armazenar o deslocamento típico em instruções `lw` e `sw`, bem como a maioria dos valores usados com a instrução `slti`.
 - Nas instruções `addi`, `slti`, `sltiu`, o sinal do resultado é estendido para 32 bits e guardado no registo `rt`. Assim, o imediato é interpretado como um inteiro com sinal (complementos de 2).
 - Veremos à frente o que fazer quando o número imediato é demasiado grande para ser representado só com 16 bits...

Exemplo formato I (1/2)

- Instrução MIPS:

`addi $21, $22, -50`

`opcode` = 8 (ver tabela no livro)

`rs` = 22 (registro operando)

`rt` = 21 (registro alvo/destino)

`immediate` = -50 (valor passado)

Exemplo formato I (2/2)

- MIPS Instruction:

`addi $21, $22, -50`

Representação de campos decimal:

8	22	21	-50
---	----	----	-----

Representação de campos binária:

001000	10110	10101	1111111111001110
--------	-------	-------	------------------

Representação hexadecimal : 22D5 FFCE_{hex}

Representação decimal: 584,449,998_{ten}

Quiz

Que instrução é representado por $35_{(10)}$?

1. add \$0, \$0, \$0
2. subu \$s0,\$s0,\$s0
3. lw \$0, 0(\$0)
4. addi \$0, \$0, 35
5. subu \$0, \$0, \$0

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

opcode	rs	rt	offset		
--------	----	----	--------	--	--

opcode	rs	rt	immediate		
--------	----	----	-----------	--	--

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

Números e nomes dos registos:

0: \$0, .. 8: \$t0, 9:\$t1, ..15: \$t7, 16: \$s0, 17: \$s1, .. 23: \$s7

Opcodes e campos

add: opcode = 0, funct = 32

subu: opcode = 0, funct = 35

addi: opcode = 8

lw: opcode = 35

Quiz

Que instrução é representada por $35_{(10)}$?

1. add \$0, \$0, \$0

0	0	0	0	0	32
---	---	---	---	---	----

2. subu \$s0,\$s0,\$s0

0	16	16	16	0	35
---	----	----	----	---	----

3. lw \$0, 0(\$0)

35	0	0	0		
----	---	---	---	--	--

4. addi \$0, \$0, 35

8	0	0	35		
---	---	---	----	--	--

5. subu \$0, \$0, \$0

0	0	0	0	0	35
---	---	---	---	---	----

Números e nomes dos registos:

0: \$0, .. 8: \$t0, 9:\$t1, ..15: \$t7, 16: \$s0, 17: \$s1, .. 23: \$s7

Opcodes e campos

add: opcode = 0, funct = 32

subu: opcode = 0, funct = 35

addi: opcode = 8

lw: opcode = 35

Quiz

Indique qual dos seguintes códigos em hexadecimal correspondem à codificação da instrução em *Assembly* do MIPS `xor $4,$6,$2`.

- a) 0x00C22026
- b) 0x00C41026
- c) 0x00C22020
- d) 0x20C4FFFF

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