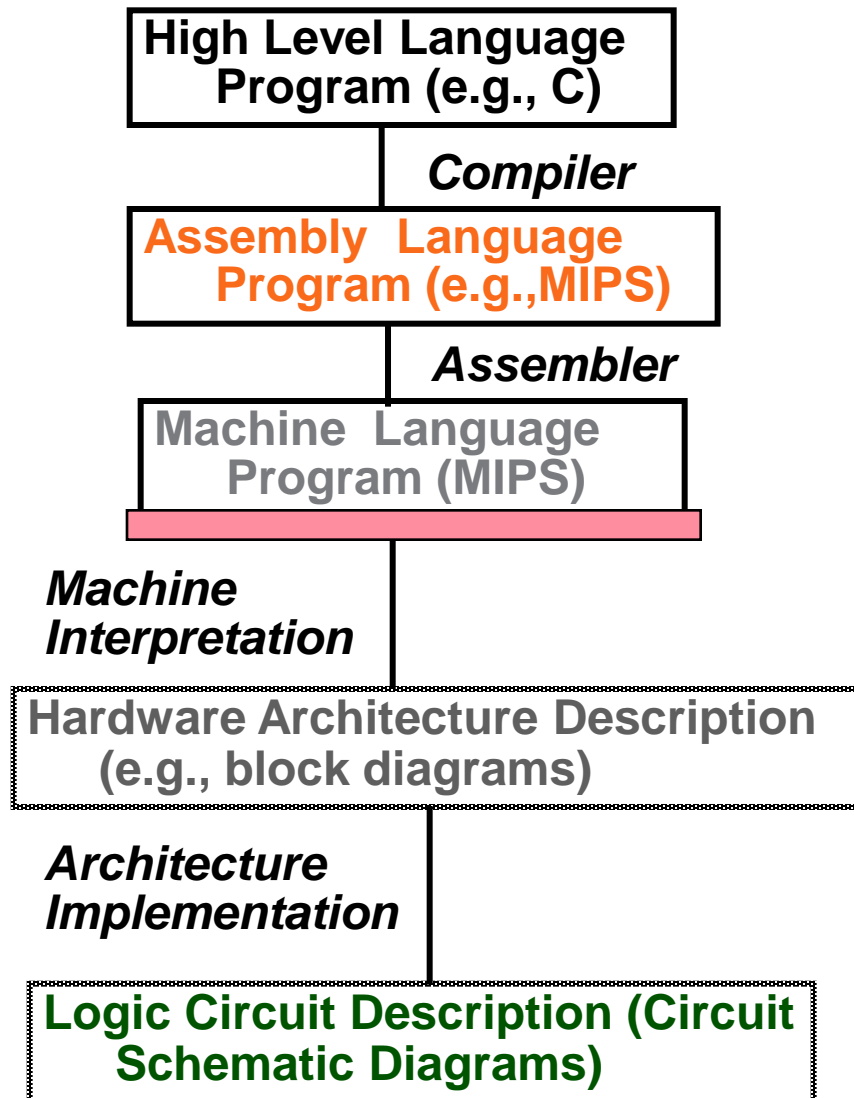




Representação de Instruções no MIPS

Arquitetura de Computadores 2024/2025

Níveis de representação num computador



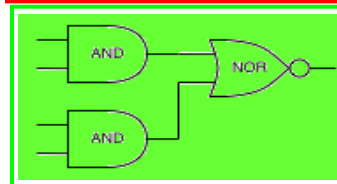
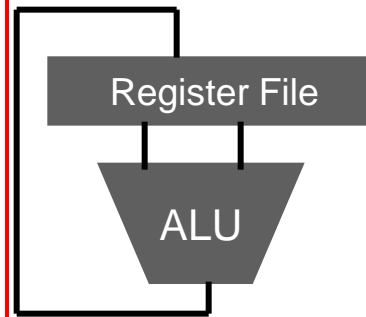
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

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
```

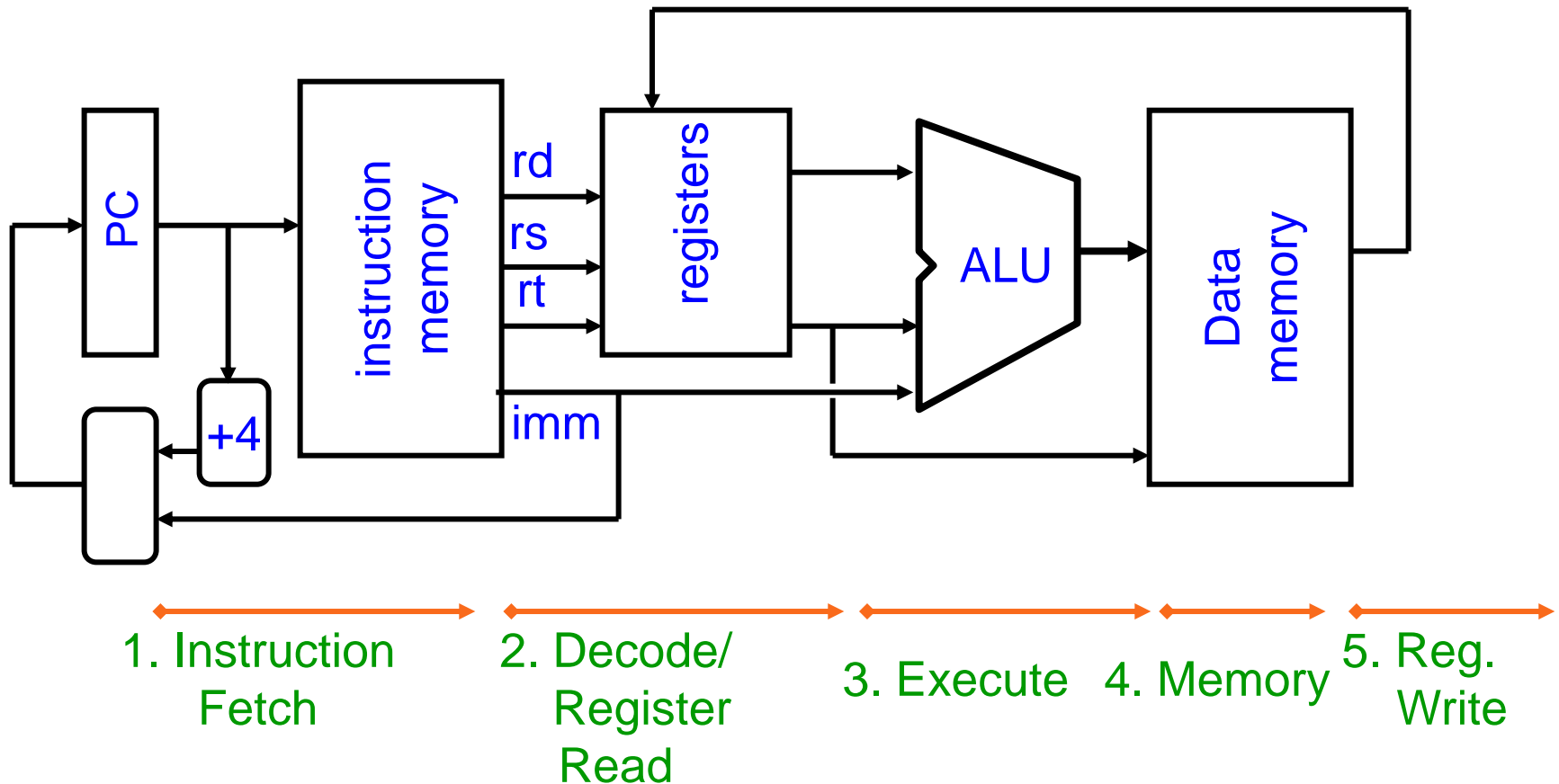
AC



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)

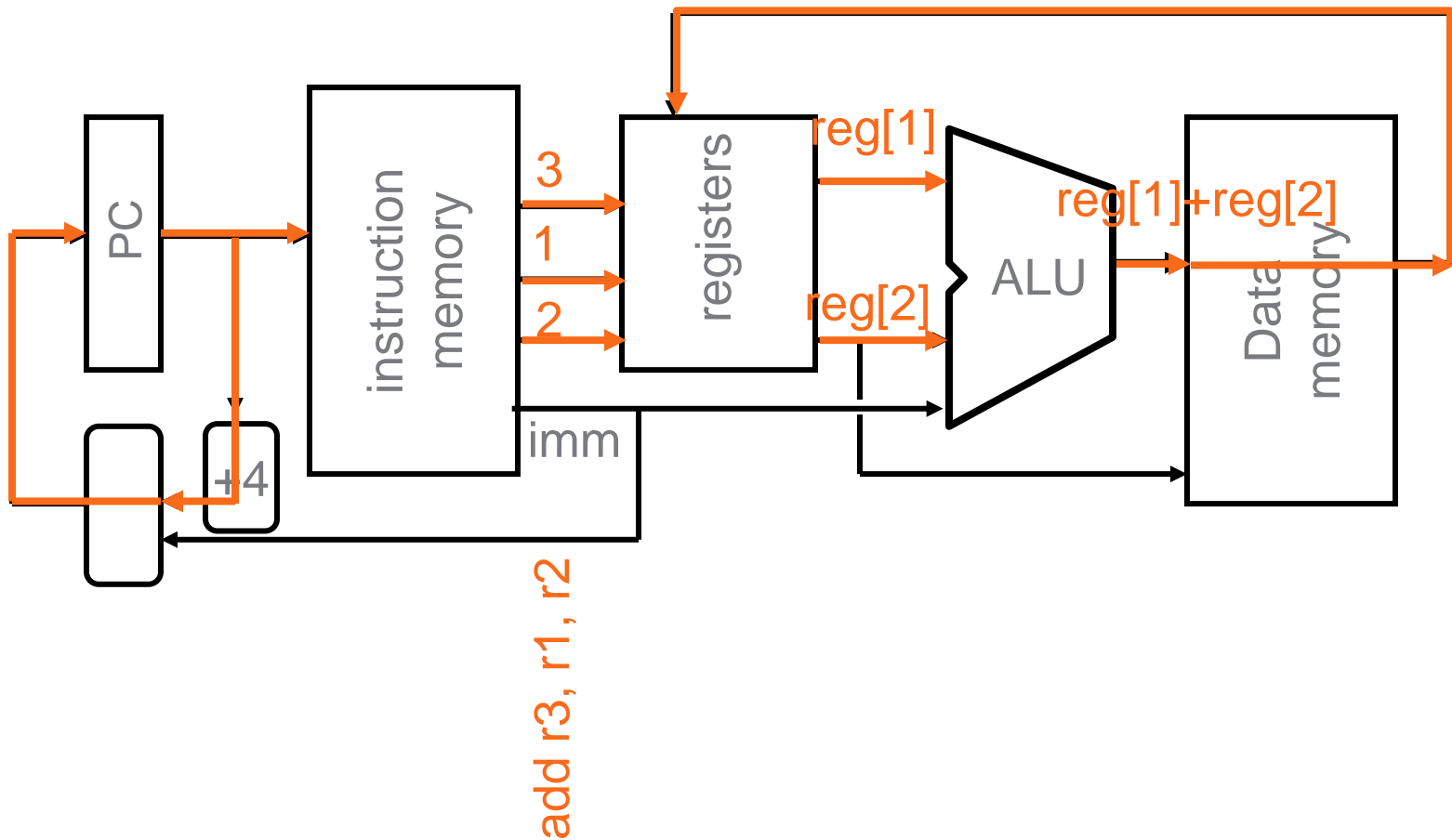
- Etapa 5: Register Write
 - A maioria das instruções escreve o resultado de uma determinada operação num registo destino.
 - exemplos: operações aritméticas e lógicas, deslocamentos, loads, `slt`
 - E quanto aos stores, jumps e branches?
 - Estas instruções não escrevem nenhum resultado num registo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

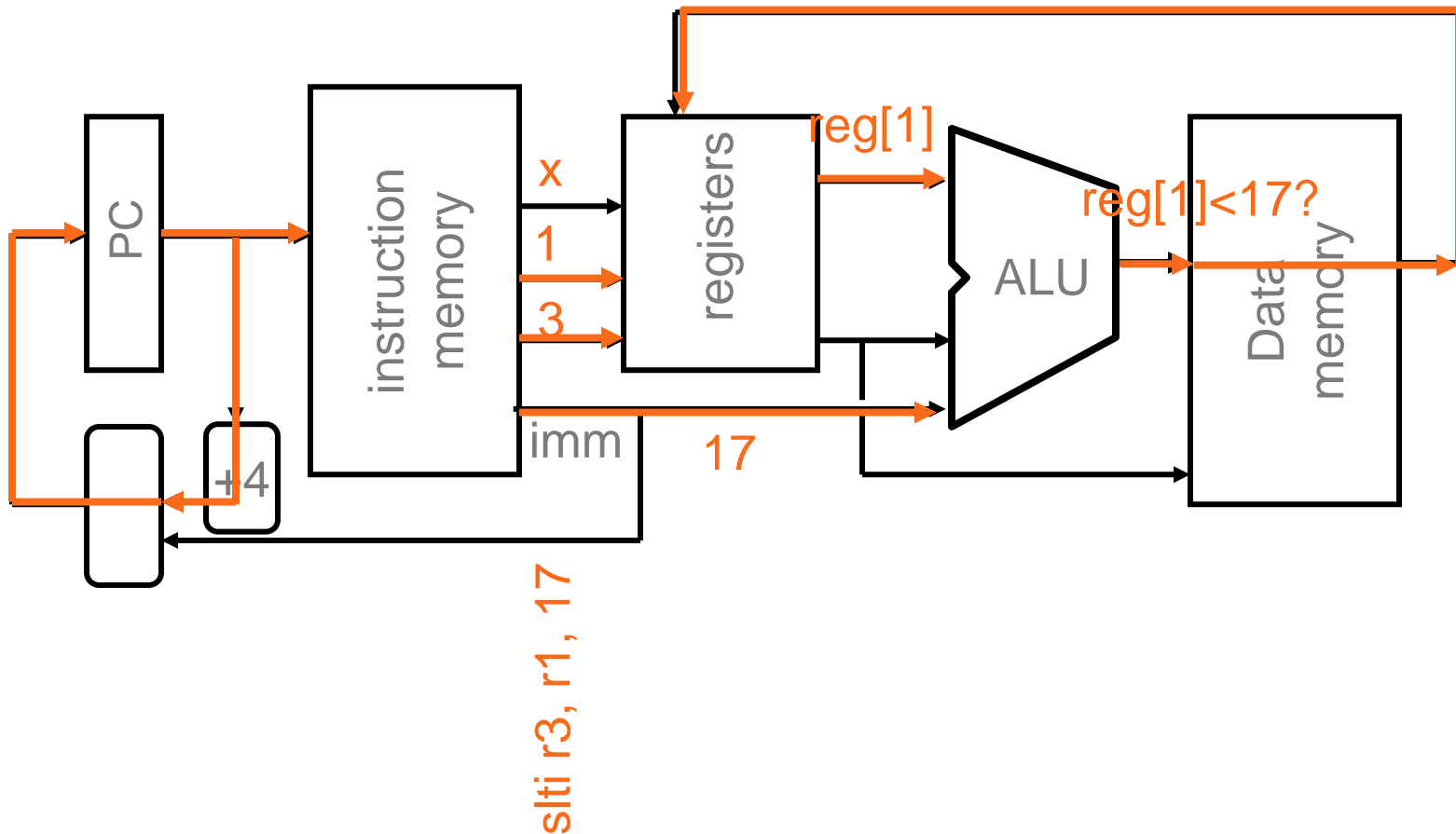


Datapath Walkthroughs (2/3)

- `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`

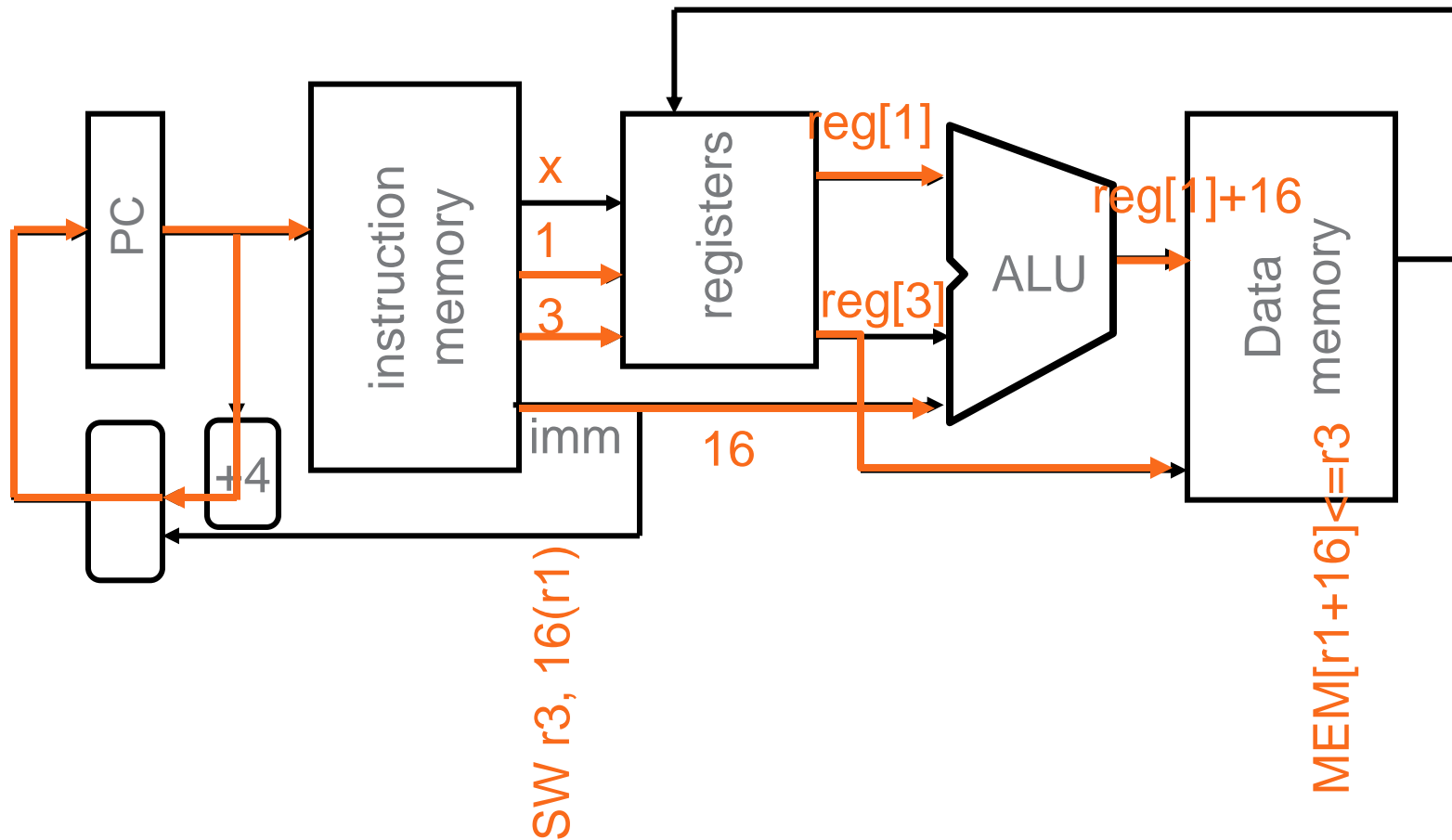


Datapath Walkthroughs (3/3)

- `sw $r3, 16($r1)`
 - Etapa 1: fetch da instrução, inc. PC
 - Etapa 2: decodificação 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)**



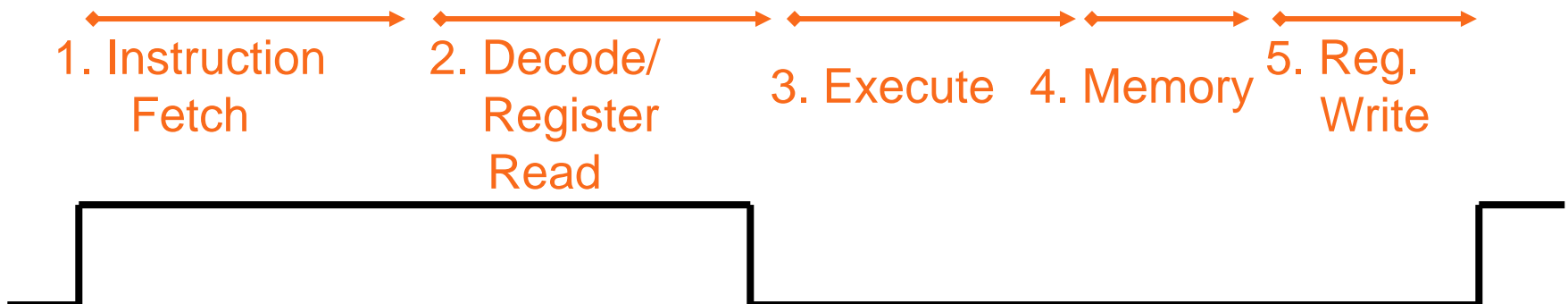
Exemplo: Instrução sw



CPU clocking (1/2)

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

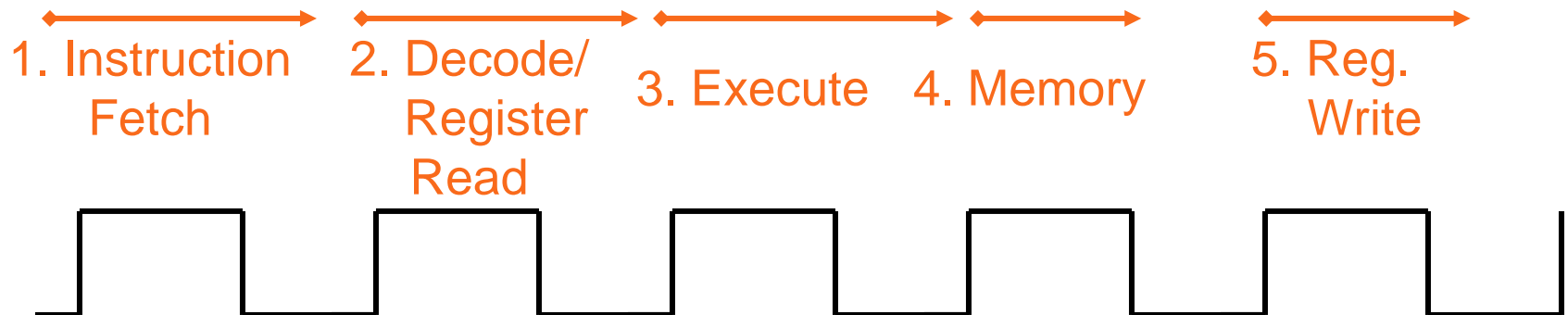
- Single Cycle CPU: Todas as etapas de uma instrução são completadas em um único longo ciclo de relógio.



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.
- 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: Compatibilidade Binária

- 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)

◆ Cada campo tem um nome/sigla:

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

◆ Os campos “r” normalmente especificam registos

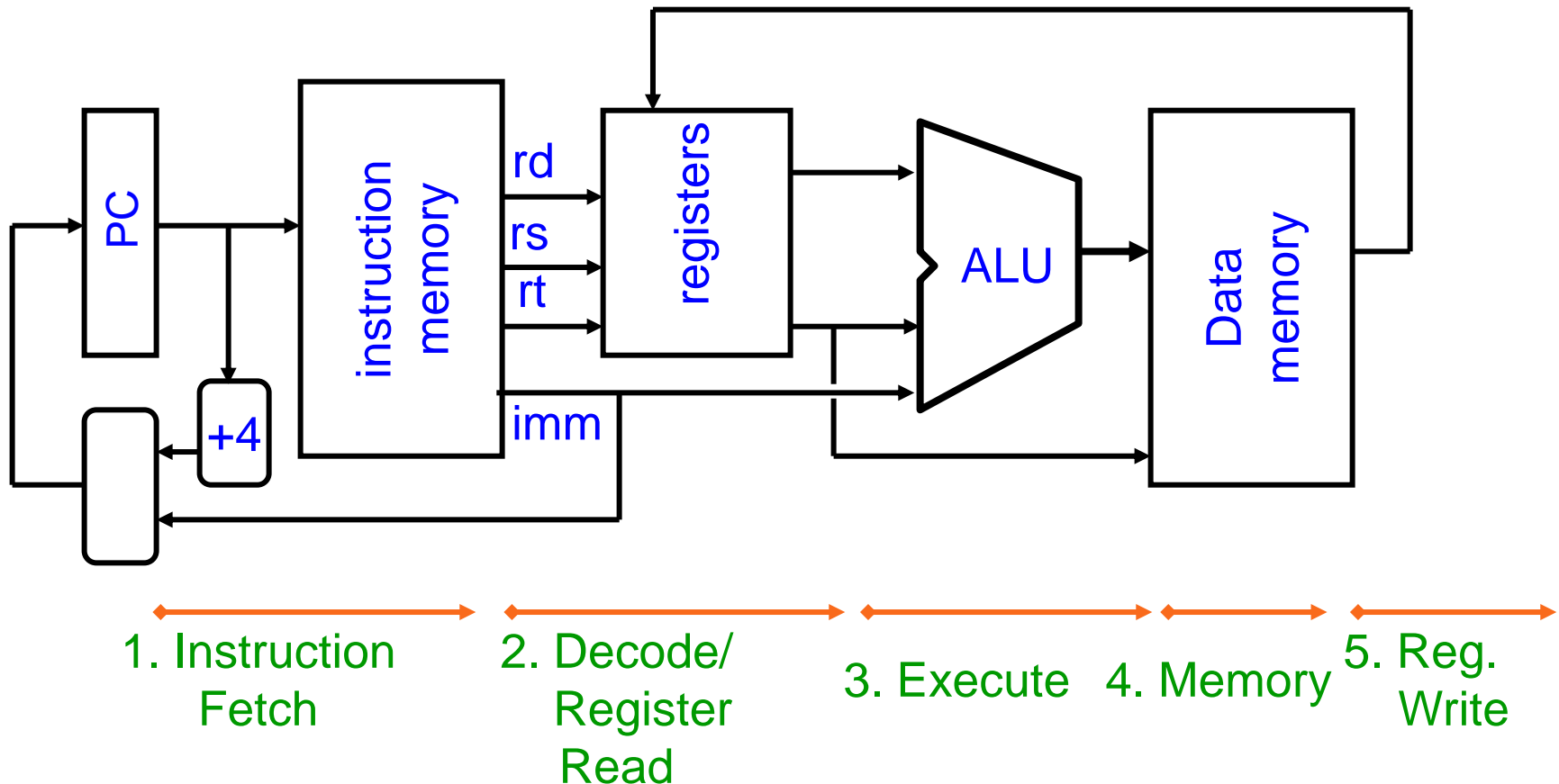
- rs (Source Register): especifica o primeiro operando
- rt (Target Register): especifica o segundo operando
- rd (Destination Register): especifica o registo que recebe o resultado

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

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

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 (Shift amount) indica o deslocamento a ser feito pelas instruções `slr`, `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	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

opcode = 0 (veja a tabela no slide anterior)

funct = 32 ou 0x20 (veja a tabela no slide anterior)

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 hexadecimal:

012A 4020_{hex}

- 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;
- 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
--------	----	----	----------

I

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

R

- ◆ **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

Limitação do formato I (1/3)

- Problema:
 - Na maior parte das situações instruções como `addi`, `lw`, `sw` e `slli` têm immediatos que são suficientemente pequenos para caberem num campo de 16 bits.
 - Isto valida a opção de usar instruções I que ocupam uma *word* (*make the common case faster*)
 - ...no entanto o que fazer quando o imediato não couber no campo de 16 bits?
 - Precisamos de ter uma estratégia para lidar com immediatos de 32 bits.

Limitação do formato I (2/3)

- Solução:
 - Resolver com software + nova instrução de suporte
 - Em vez de criarmos um conjunto de novas instruções, vamos manter aquelas que já vimos que serão coadjuvadas por nova instrução adicional.
- Nova instrução:
`lui register, immediate`
 - `lui` significa Load Upper Immediate
 - A instrução agarra nos 16-bits do imediato e coloca-os nos bits mais significativos do registo destino
 - A metade mais baixa do registo fica com 0s

Limitação do formato I (3/3)

- Solução do problema:
 - Como é `lui` nos pode ajudar?

- Exemplo:

```
addi    $t0, $t0, 0xABABCD
```

É codificado:

```
lui      $at, 0xABAB
ori      $at, $at, 0xCDCD
add      $t0, $t0, $at
```

Lembra-se do registo
\$at ?
É o registo “assembler
temporary”

- As instruções de formato I `ori` e `addi` têm um imediato de 16-bits.
- *Era bom que o assembler fizesse este desdobramento de forma automática ...*

Pseudo-Instruções (1/4)

- Pseudo-Instrução: É um comando para o MIPS que não é directamente mapeado numa instrução linguagem máquina.
 - Em vez de ser codificada em hardware, a pseudo-instrução é convertida pelo *assembler* numa sequência de instruções linguagem máquina.
- Exemplos:
 - Register move

```
move  reg2, reg1
```

É desdobrado em:

```
add   reg2, $zero, reg1
```

Pseudo-Instruções (2/4)

- Exemplos:

- Load Immediate

- ```
li reg,value
```

- Se o imediato couber em 16 bits:

- ```
addi reg,$zero,value
```

- Caso contrário:

- ```
lui reg,upper 16 bits of value
ori reg,reg,lower 16 bits
```

Nota: Repare que o *assembler* tem que fazer a avaliação na compilação

# Pseudo-Instruções (3/4)

- Exemplo:
  - Load Address: Coloca o endereço de uma instrução ou variável global num registo

```
la reg, label
```

Se o valor couber em 16 bits:

```
addi reg, $zero, label_value
```

Se não:

```
lui reg, upper 16 bits of value
ori reg, reg, lower 16 bits
```

# Pseudo-Instruções (4/4)

## • Exemplo

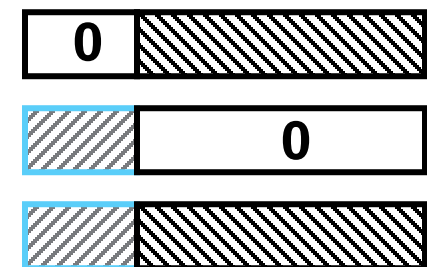
– Rotate Right Instruction

```
ror reg, reg_org, value
```



Fica como:

```
srl $at, reg, value
sll reg, reg, 32-value
or reg, reg_org, $at
```



- ◆ O registo `$at` é utilizado pelo *assembler* como registo auxiliar para implementar as pseudo-instruções. Por isso não dever ser utilizado directamente pelo programador

# True Assembly Language (1/2)

- **MAL** (MIPS Assembly Language): conjunto de instruções que o programador pode utilizar para fazer código para o MIPS; isto inclui as pseudo-instruções.
- **TAL** (True Assembly Language): conjunto de instruções que são traduzidas directamente para uma instrução linguagem máquina de 32 bits
- Um programa tem de ser convertido de MAL para TAL antes de ser traduzido em 1s e 0s.

# True Assembly Language (2/2)

- Como é que o assembler do MIPS reconhece uma pseudo-instrução?
  - Verifica se a instrução está na lista oficial de pseudo-instruções (caso do `ror` e `move`)
  - Também existem situações em que a instrução tem um sinónimo TAL mas os operandos estão incorrectos (tipicamente existe um imediato com mais de 16 bits). Neste caso faz o desdobramento ...

```
addi $t0, $s0, 0x0ABC3EF1
```

O imediato tem mais do que 16 bits. Assim de MAL para TAL temos ..

```
lui $at, 0x0ABC
ori $at, $at, 0x3EF1
add $t0, $s0, $at
```

# Branches e endereçamento relativo (1/5)

- Considere o formato I para codificar a instrução `beq` ou `bne`

|                     |                 |                 |                        |
|---------------------|-----------------|-----------------|------------------------|
| <code>opcode</code> | <code>rs</code> | <code>rt</code> | <code>immediate</code> |
|---------------------|-----------------|-----------------|------------------------|

- ◆ `opcode` especifica `beq` ou `bne`
- ◆ `rs` e `rt` especificam os registos a ser comparados
- ◆ O que é que o campo `immediate` especifica?
  - `Immediate` só tem 16 bits
  - **PC** (**Program Counter**) tem o endereço da instrução que está a ser executada. É um ponteiro para memória com 32-bits.
- ◆ Assim o `immediate` não pode especificar o endereço completo para onde queremos saltar com o branch.



# Branches e endereçamento relativo (2/5)

- Como é que tipicamente se usam branches (“*check the common case*”)?
  - Resposta: instruções `if-else`, ciclos `while`, `for`,...
  - Os Loops são normalmente pequenos: tipicamente até 50 instruções
  - As chamadas de funções e os saltos incondicionais são feitos com instruções `j` e `jal`), e não branches
- Conclusão: potencialmente um “*branch*” pode mover a execução para qualquer ponto da memória, mas, na maior parte dos casos, o *branch* só precisa de alterar o **PC** adicionando-lhe um valor relativamente pequeno (positivo ou negativo)

# Branches e endereçamento relativo (3/5)

- Solução para os “branches” serem codificados numa instrução de 32-bits: **PC-Relative Addressing**
- O campo `immediate` de 16 bits é interpretado como um inteiro com sinal em complementos de 2. Este valor é **adicionado** ao PC no caso de se verificar o salto (endereçamento relativo à posição actual)
- Com este mecanismo é possível fazer saltos de  $\pm 2^{15}$  bytes com relação ao valor corrente do registo PC. Isto é suficiente para a maior parte dos ciclos!
- Ideias para optimizar isto ainda mais?



# Branches e endereçamento relativo (4/5)

- Lembre-se que as instruções são *words*, e que as *words* são guardadas de forma alinhada na memória (o “*byte address*” de uma instrução é sempre um múltiplo de 4, o que significa que, em representação binária, termina sempre em 00).
  - Assim, o número de bytes a adicionar ao PC é sempre um múltiplo de 4 de forma a respeitar o alinhamento.
  - Então podemos especificar o *immediate* em termos de *words*.
- Com este ajuste passamos a poder dar saltos de  $\pm 2^{15}$  words a partir do PC (or  $\pm 2^{17}$  bytes), sendo possível lidar com *loops* 4 vezes maiores.

# Branches e endereçamento relativo (5/5)

- Cálculo de saltos em Branches :
  - Se não houver salto:
$$PC = PC + 4$$
$$PC+4 = \text{“byte address” da próxima instrução}$$
  - Se houver salto:
$$PC = (PC + 4) + (\text{immediate} * 4)$$
  - Observações:
    - `Immediate` especifica o número de *words* a saltar, o que é o mesmo que dizer o número de instruções.
    - `Immediate` pode ser um número positivo ou negativo.

# Exemplo de *Branch*

- Código MIPS:

```
Loop: beq $9, $0, End
 add $8, $8, $10
 addi $9, $9, -1
 j Loop
```

End:

- beq branch tem formato I:

opcode = 4

rs = 9

rt = 0

immediate = 3 (número de instruções a saltar)

**Cuidado:** o que aconteceria se a terceira instrução fosse

addi \$9, \$9, 0xAFFFF ?

# Questões PC-addressing

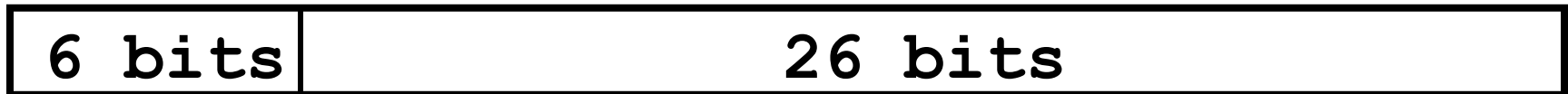
- O valor do deslocamento num *branch* altera-se se movermos o código na memória?
- O que fazer quando o destino do salto está a mais do que  $2^{15}$  instruções?
- Porque necessitamos de diferentes modos de endereçamento (modos diferentes de compor o endereço de memória)? Porque não ter apenas um modo?

# Instruções formato J (1/4)

- No caso dos branches, partimos do princípio de que o salto nunca seria muito distante. Isto permitiu a codificação em instruções formato I usando endereçamento relativo a partir do valor corrente de PC.
- No entanto, no caso de saltos incondicionais (`j` e `jal`), podemos querer saltar para qualquer lugar na memória.
- Nesta caso deveríamos ser capazes de especificar um endereço de 32 bits.
- Infelizmente, é impossível colocar numa instrução com o tamanho de uma word um `opcode` de 6 bits e um endereço de 32 bits.

# Instruções formato J (2/4)

- Este tipo de instruções tem dois “campos” com o seguinte tamanho:



◆ Os nomes dos campos são:



## ◆ Ideia chave

- Manter o campo de `opcode` idêntico ao formato R e formato I por uma questão de consistência.
- Colapsar todos os outros campos para arranjar o máximo de espaço possível para colocar o endereço.



## Instruções formato J (3/4)

- Para já, conseguimos acomodar 26 bits de um endereço de 32-bits.
- Optimização:
  - Como a memória está alinhada podemos usar o mesmo truque que usámos para as instruções I: o campo é interpretado em termos de número de *words* em vez de *bytes*.
  - Desta forma conseguimos “*cobrir*” uma região de  $2^{28}$  bytes de memória.

# Instruções formato J (4/4)

- Assim conseguimos especificar 28 bits do endereço de 32-bits
- O que fazer quanto aos 4 bits que faltam?
  - Na prática consideramos que os 4 bits mais significativos de PC se mantêm, e a instrução só especifica os 28 menos significativos.
  - Tecnicamente isto significa que não podemos saltar para qualquer sítio da memória. No entanto esta solução permite resolver 99.9999...% das situações reais
    - Repare que conseguimos lidar com blocos de memória até 256 MB
- Nos casos em que é necessário especificar um endereço de 32 bits temos que o colocar num registo e usar a instrução `jr` (este *jump* é uma instrução de tipo R)

# QUIZ

- Quais das seguintes instruções são **MAL** e quais são **TAL**?

- i. `addi $t0, $t1, 40000`
- ii. `beq $s0, $s1, Exit`
- iii. `sub $t0, $t1, 1`

|     | ABC        |
|-----|------------|
| 1 : | <b>MMM</b> |
| 2 : | <b>MMT</b> |
| 3 : | <b>MTM</b> |
| 4 : | <b>MTT</b> |
| 5 : | <b>TMM</b> |
| 6 : | <b>TMT</b> |
| 7 : | <b>TTM</b> |
| 8 : | <b>TTT</b> |

# Concluindo ...

- MIPS Machine Language Instruction:  
cada instrução é representada por uma word de 32 bits

|             |        |                |    |           |       |       |
|-------------|--------|----------------|----|-----------|-------|-------|
| R<br>I<br>J | opcode | rs             | rt | rd        | shamt | funct |
|             | opcode | rs             | rt | immediate |       |       |
|             | opcode | target address |    |           |       |       |

- Os branches usam endereçamento relativo a partir do valor corrente de PC, os jumps usam endereçamento absoluto.
- A Desassemblagem é possível se começarmos por fazer a descodificação do campo `opcode`. (a ver)

# Para saber mais ...

- P&H - Capítulos 2.4, 2.9 e 2.10
- Anexo A17

