Tópicos:

- Introdução Sistemas de Computação de Uso Geral
- A arquitetura MIPS

Questões:

1. Quais são os 3 blocos fundamentais de um sistema computacional? I/O, Memory, CPU
2. Quais são os 3 principais blocos funcionais que integram um CPU? ALU, FileRegister, ControlUnit
3. Qual a função do registo *Program Counter*? Conter o endereço da instrução seguinte
4. Quais os passos mais importantes em que se decompõe a execução de uma instrução no CPU? Instruction Fetch, Instruction Decode, OperandFetch,Execute, Store Result
5. Descreva de forma sucinta a função de um compilador. Traduzir o código de alto nível para Assembly
6. Descreva de forma sucinta a função de um assembler. Traduzir o código de Assembly para linguagem máquina
7. Quantos registos internos de uso geral tem o MIPS? 32
8. Qual a dimensão, em bits, que cada um dos registos internos do MIPS pode armazenar? 32
9. Qual a sintaxe, em *Assembly*, de uma instrução aritmética no MIPS? inst $rdest,%rsrc1,$rsrc2
10. O que distingue a instrução SRL da instrução SRA do MIPS? O srl interpreta os valores como unsigned, sra interpreta como signed. (srl adiciona 0) (sra adiciona valor igual ao MSB)
11. Se **$5=0x81354AB3** **(10000001001101010100101010110011)**, qual o resultado, expresso em hexadecimal, das instruções:
    a. **`srl $3,$5,1`** (0100 0000 1001 1010 1010 0101 0101 1001) (0x409AA559)
    a. **`sra $4,$5,1`** (1100 0000 1001 1010 1010 0101 0101 1001) (0xC09AA559)
12. *System calls*:
    a. O que é uma *system call*? Um pedido de um serviço ao sistema operativo para realizar uma determinada funçao
    b. No MIPS, qual o registo usado para identificar a *system call* a executar? $v0
    c. Qual o registo ou registos usados para passar argumentos para as *systems calls*? $a0 a $a3 e $f12 a $f14
    d. Qual o registo usado para obter o resultado devolvido por uma *system call* nos casos em que isso se aplica? $v0 ou $f0

13. Em Arquitetura de Computadores, como definiria o conceito de endereço? Número único que identifica cada registo na memória

14. O que é o espaço de endereçamento de um processador? Gama total de endereços que o CPU referencia (2^n - 1)

15. Como se organiza internamente um processador?Datapath e Control Unit Quais são os blocos fundamentais da secção de dados? Muxs, PCcounter,ALU, File Register Para que serve a unidade de controlo? Para coordenar os elementos

16. Qual é o conceito fundamental por detrás do modelo de arquitetura "*stored-program*"? Programa guardado na memória externa

17. Como se codifica uma instrução? Que informação fundamental deverá ter o código de uma instrução? A informação fundamental é o opcode, registo de destino e registos fontes ,registos para cálculo de endereço, valores imediatos ou offsets, dependendo da instrução

18. Descreva pelas suas próprias palavras o conceito de **ISA**. É uma arquitetura do conjunto de instruções que representam a base do funcionamento de um processador

19. Quantas e quais são as classes de instruções que agrupam as diferentes instruções de uma dada arquitetura? 3, instruções tipo I, J,R

20. O que carateriza e distingue as arquiteturas do tipo "*register-memory*" e "*load-store*"? De que tipo é a arquitetura MIPS? Na arquitetura load store não é possível atuar diretamente sobre a memória, enquanto na register-memory, os operandos podem estar entre os registos internos e a memória. A arquitetura MIPS é load store

21. O ciclo de execução de uma instrução é composto por uma sequência ordenada de operações. Quantas e quais são essas operações (passos de execução)? OperandFetch,Execute, Store Result

22. Como se designa o barramento que permite identificar, na memória, a origem/destino da informação transferida? Address Bus

23. Qual a finalidade do barramento normalmente designado por *Data Bus*? Barramento que transfere informação entre a memória e o CPU ou entre a memória e o I/O

24. Os processadores da arquitetura hipotética ZWYZ possuem 4 registos internos e todas as instruções são codificadas em 24 bits. Num dos formatos de codificação existem 5 campos: um *OpCode* com 5 bits, três campos para identificar registos internos em operações aritméticas e lógicas e um campo para codificar valores constantes imediatos em complemento para dois. Qual a gama de representação destas constantes? A gama de representação é -2^12 a (2^12)-1

25. A arquitetura hipotética ZPTZ tem um barramento de endereços de 32 bits e um barramento de dados de 16 bits. Se a memória desta arquitetura for ***bit_addressable->cada bit tem um endereço***:

    a. Qual a dimensão do espaço de endereçamento desta arquitetura? 2^32 endereços

    b. Qual a dimensão máxima da memória suportada por esta arquitetura expressa em *bytes*? 536MB

26. Considere agora uma arquitetura em que o respetivo ISA especifica uma organização de memória do tipo ***word-addressable-> cada address tem 32 bits***, em que a dimensão da *word* é 32 bits. Tendo o espaço de endereçamento do processador 24 bits, qual a dimensão máxima de memória que este sistema pode acomodar expresso em *bytes*? 2^24 * 4 bytes

27. Relativamente à arquitetura MIPS:

    c. Com quantos bits são codificadas as instruções no MIPS? 32

    d. O que diferencia o registo **$0** dos restantes registos de uso geral? O $0 tem o valor 0x00000000 e não pode ser escrito

    e. Qual o endereço do registo interno do MIPS a que corresponde a designação lógica **$ra**? 31

28. No MIPS, um dos formatos de codificação de instruções é designado por R:

    a. Quais os campos em que se divide este formato de codificação? Opcode rd,rs,rt,shamt,funct

    b. Qual o significado de cada um desses campos? Operation code, registo destino, registo source1, registo source2, Shift amount, function(alu)

    c. Qual o valor do campo *opCode* nesse formato? 6 bits

    d. O que faz a instrução cujo código máquina é: **0x00000000**? NOP (sll $0,$0,0)

29. O símbolo " >> " da linguagem C significa deslocamento à direita e é traduzido por SRL ou SRA (no caso do MIPS). Em que casos é que o compilador gera um SRL e quando é que gera um SRA? Dependendo se as variáveis são unsigned ou signed

30. Qual a instrução nativa do MIPS em que é traduzida a instrução virtual "**move $4,$15**"? or $4,$0,$15

31. Determine o código máquina das seguintes instruções (verifique a tabela na última página):

    a. **xor $5,$13,$24** 000000 01101 11000 00101 00000 100110 (0x01B82826)

    b. **sub $25,$14,$8** 0000 0001 1100 1000 1100 1000 0010 0010 (0x01C8C822)

    c. **sll $3,$9,7** 0000 0001 0010 0000 0001 1001 1100 0000 (0x012019C0)

    d. **sra $18,$9,8** 0000 0001 0010 0000 1001 0010 0000 0011 (0x01209203)

32. Traduza para instruções *Assembly* do MIPS a seguinte expressão aritmética, supondo **x** e **y** são inteiros e residentes em **$t2** e **$t5**, respetivamente (apenas pode usar instruções nativas e não deverá usar a instrução de multiplicação): **y = -3 * x + 5;**

```
Add $t5, $t2, $t2

Add $t5, $t5, $t2

Nor $t5, $t5, $0

Addi $t5, $t5, 6
```

33. Traduza para instruções *assembly* do MIPS o seguinte trecho de código:

```
int a, b, c;                //a:$t0, b:$t1, c:$t2
unsigned int x, y, z;       //x:$a0, y:$a1,z:$a2
z = x >> 2 + y;
c = a >> 5 - 2 * b;

TEM AQUI BUG CARA
```

34. Considere que as variáveis **g**, **h**, **i** e **j** são conhecidas e podem ser representadas por uma variável de 32 bits num programa em C. Qual a correspondência em linguagem C às seguintes instruções:

```
a. add  h, i, j    h = i + j

b. addi j, j, 1    j = j + 1

c. add  h, g, j    h = g + j
```

35. Assumindo que **g=1**, **h=2**, **i=3** e **j= 4** qual o valor destas variáveis no final das sequências das alíneas da questão anterior?

A: h = 7, i = 3, j =4 B: j = 5 C: h = 6

36. Qual a operação realizada pela instrução "**slt**" e quais os resultados possíveis?

A instrução dá o valor '1' ou '0' ao rd, dependendo se o rs é menor que rt (1 no caso afirmativo)

37. Qual o valor armazenado no registo **$1** na execução da instrução "**slt $1, $3, $7**", admitindo que:

a. **$3=5 e $7=23** 1

b. **$3=0xFE e $7=0x913D45FC** 0

38. Com que registo implícito comparam as instruções "**bltz**", "**blez**", "**bgtz**" e "**bgez**"? $0

39. Decomponha em instruções nativas do MIPS as seguintes instruções virtuais:

a. **blt $15,$3,exit** `slt $1, $15, $3  bne $1, $0, exit`

b. **ble $6,$9,exit** `slt $1, $9, $6  beq $1, $0, exit`

c. **bgt $5,0xA3,exit** `slt $1, $5, 0xA3  beq $1, $0, exit`

d. **bge $10,0x57,exit** `slti $1, 0x57, $10 bne $1, $0, exit`

e. **blt $19,0x39,exit** `slti $1, $19, 0x39 bne $1,$0,exit`

f. **ble $23,0x16,exit** `slti $1, 0x16, $23 beq $1, $0, exit`

40. Na tradução e C para *assembly*, quais as principais diferenças entre um ciclo "**`while(…){…}`**" e um ciclo "**`do{…}while(…);`**" ? A condição no dowhile é no fim do ciclo e verifica a condição correta e o código do ciclo corre pelo menos uma vez

41. Traduza para *assembly* do MIPS os seguintes trechos de código de linguagem C (admita que **a**, **b** e **c** residem nos registos **$4**, **$7** e **$13**, respetivamente):

    a.
```
if(a > b && b != 0)  c = b << 2;
   else
     c = (a & b) ^ (a | b);
```

```
If: bge $4, $7, else

    Bne $7, $0, else

    Sll $13, $7, 1

    J endif

Else:

     And $13, $4, $7

     Or $4, $4, $7

     Xor $13, $13, $4

Endif:
```

    b.
```
if(a > 3 || b <= c)  c = c - (a + b);
   else
     c = c + (a - 5);
```

```
If: bgt $4, 3, inst

    ble $7, $13, inst

    j else

inst: add $4, $4, $7

      sub $13, $13, $4

    j endif

else: addi $4, $4, -5

      add $13, $13, $4

endif:~
```

42. Qual o modo de endereçamento usado pelo MIPS para ter acesso a palavras residentes na memória externa? Endereçamento indireto por registo com deslocamento

43. Na instrução "**`lw $3,0x24($5)`**" qual a função dos registos **`$3`** e **`$5`** e da constante **`0x24`**? $3 é o registo destino que vai conter o valor da memoria, $5 é o registo que contem o endereço da memoria, 0x24 é o valor em bytes do offset desse endereço de memoria

44. Qual é o formato de codificação das instruções de acesso à memória no MIPS e qual o significado de cada um dos seus campos? Opcode, rs, rt, offset (instrução tipo I) 6bits, 5bits, 5bits, 16bits

45. Qual a diferença entre as instruções "**`sw`**" e "**`sb`**"? O sw guarda uma word 4 bytes e o sb guarda 1 byte (o sw ocupa 4 enderecos na memoria enquanto que sb ocupa 1 endereco)

46. O que distingue as instruções "**`lb`**" e "**`lbu`**"? O lb vai buscar passar o byte para um valor em decimal signed, o lbu vai buscar o código byte sem sinal

47. O que acontece quando uma instrução **`lw/sw`** acede a um endereço que não é múltiplo de 4? Cria uma excecao

48. Traduza para *assembly* do MIPS os seguintes trechos de código de linguagem C (atribua registos internos para o armazenamento das variáveis **`i`** e **`k`**) :

    SE NÃO SABES DEVIAS CHUMBAR

    a.
    ```
    int i, k;  for(i=5, k=0; i < 20; i++, k+=5);
    ```

    b.
    ```
    int i=100, k=0;  for( ; i >= 0; )
    {  i--;
       k -=
       2;
     }
    ```

    c.
    ```
    unsigned int k=0;  for( ; ; )
    {  k +=
       10;
    }
    ```

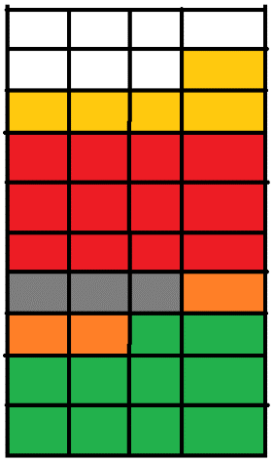
    d.
    ```
    int k=0, i=100;  do
    {  k +=
       5;
    } while(--i >= 0);
    ```

49. Sabendo que o *OpCode* da instrução "**`lw`**" é **`0x23`**, determine o código máquina, expresso em hexadecimal, da instrução "**`lw $3,0x24($5`**)". 1000 1100 1010 0011 0000 0000 0001 1000 (0X8CA30018)

50. Suponha que a memória externa foi inicializada, a partir do endereço **0x10010000**, com os valores **0x01,0x02,0x03,0x04,0x05** e assim sucessivamente. Suponha ainda que **$3=0x1001** e **$5=0x10010000**. Qual o valor armazenado no registo destino após a execução da instrução "**lw $3,0x24($5)**" admitindo uma organização de memória *little endian*?

    0x28272625

    a. **lbu $3,0xA3($5)** **0x000000A4**
    b. **lb $4,0xA3($5)** **0Xffffffa4**



51. Quantos *bytes* são reservados em memória por cada uma das diretivas:

    a. **L1: .asciiz "Aulas5&6T"** **10bytes**
    b. **L2: .byte 5,8,23** **3 bytes**
    c. **L3: .word 5,8,23** **12 bytes**
    d. **L4: .space 5** **5 bytes**

52. Desenhe esquematicamente a memória e preencha-a com o resultado das diretivas anteriores admitindo que são interpretadas sequencialmente pelo *Assembler*.

53. Supondo que "**L1:**" corresponde ao endereço inicial do segmento de dados, e que esse endereço é **0x10010000**, determine os endereços a que correspondem os *labels* "**L2:**", "**L3:**" e "**L4:**".

    L1: **0x10010000, L2:0x1001000A, L3:0x10010010, L4:0x1001001C**

54. Suponha que "**b**" é um *array* declarado como "**int b[25];**":

    a. Como é obtido o endereço inicial do *array*, i.e., o endereço a partir do qual está armazenado o seu primeiro elemento? la $t0, b

    b. Supondo uma memória "*byte-addressable*", como é obtido o endereço do elemento "**b[6]**"? lw $t0, 24($t0)

55. O que é codificado no campo offset do código máquina das instruções "**beq/bne**" ? Os 16 bits menos significativos do Branch Target Address

56. A partir do código máquina de uma instrução "**beq/bne**", como é formado o endereço-alvo (*Branch Target Address*)? PC + 4 + os 16 bits do offset extendidos para 32bits multiplicados por 4 (sll 2)

57. Qual o formato de codificação de cada uma das seguintes instruções: "**beq/bne**", "**j**", "**jr**"? Formato I, Formato J, Formato R

58. A partir do código máquina de uma instrução "**j**", como é formado o endereço-alvo (*Jump Target Address*)? Os 4 MSB do PC+4 + 26 bits do campo imediato de J extendidos para 28

59. Dada a seguinte sequência de declarações: **int b[25]; int a; int *p = b;**

Identifique qual ou quais das seguintes atribuições permitem aceder ao elemento de índice 5 do *array* "**b**":

`a = b[5];`	`a = *p + 5;`	`a = *(p + 5);`	`a = *(p + 20);`

60. Assuma que as variáveis **`f, g, h, i`** e **`j`** correspondem aos registos **`$t0`**, **`$t1`**, **`$t2`**, **`$t3`** e **`$t4`** respetivamente. Considere que o endereço base dos *arrays* **A** e **B** está contido nos registos **`$s0`** e **`$s1`**. Considere ainda as seguintes expressões:

```
f = g + h + B[2]

j = g - A[B[2]]
```

a. Qual a tradução para *assembly* de cada uma das instruções C indicadas?

```
Lw $t5, 8($s1)          lw $t5, 8($s1)

Add $t1, $t1, $t2       sll $t5, $t5, 2

Add $t0,  $t5, $t1      addu $t6, $s0, $t5

                        Lw $t5, 0($t6)

                        Sub $t4,  $t3, $t5
```

b. Quantas instruções *assembly* são necessárias para cada uma das instruções C indicadas? E quantos registos auxiliares são necessários? Para a primeira 1 auxiliar e 3 instrucoes, para a segunda 2 auxiliares e 5 instrucoes

c. Considerando a tabela seguinte que representa o conteúdo byte-a-byte da memória, nos endereços correspondentes aos *arrays* A e B, indique o valor de cada elemento dos *arrays* assumindo uma organização *little endian*.

Endereço	Valor
A+12	…
A+11	0x00
A+10	0x00
A+9	0x00
A+8	0x01
A+7	0x22
A+6	0xED
A+5	0x34
A+4	0x00
A+3	0x00
A+2	0x00
A+1	0x00
A+0	0x12

Endereço	Valor
B+12	…
B+11	0x00
B+10	0x00
B+9	0x00
B+8	0x02
B+7	0x00
B+6	0x00
B+5	0x50
B+4	0x02
B+3	0xFF
B+2	0xFF
B+1	0xFF
B+0	0xFE

A[0]= 0x00000012
A[1]= 0x22ED3400
A[2]= 0x00000001

B[0]= FFFFFFFE
B[1]= 0x00005002
B[2]= 0x00000002

d. Assumindo que **g = -3** e **h = 2**, qual o valor final das variáveis **f** e **j**?

F = -1 +2 = 1, J = -3 + A[2] = -3 +1 = 2

61. Pretende-se escrever uma função para a troca do conteúdo de duas variáveis (troca(a, b);).
Isto é, se, antes da chamada à função, **a=2** e **b=5**, então, após a chamada à função, os valores de **a** e **b** devem ser: **a=5** e **b=2**

Uma solução incorreta para o problema é a seguinte:

```
void troca(int *x, int *y)
{  int aux;
     aux=*x;
     *x =*y;
     *y=aux;
}
```

Identifique o erro presente no trecho de código e faça as necessárias correções para que a função tenha o comportamento pretendido

63. Na instrução "**jr $ra**", como é obtido o endereço-alvo? Pelo campo rs do formato R

64. Qual é o menor e o maior endereço para onde uma instrução "**j**", residente no endereço de memória **0x5A18F34C**, pode saltar? Vai de 0x50000000 até 0x5FFFFFFC

65. Qual é o menor e o maior endereço para onde uma instrução "**beq**", residente no endereço de memória **0x5A18F34C**, pode saltar? Beq faz 2^{18} enderecos ou 2^{16} instrucoes logo -2^{17} até $2^{17}-1$, neste caso o 0 não conta porque o beq não salta para ele mesmo logo vai -2^{17} até 2^{17}

0x5A18F34C = 01011010000110001111001101001100
+11111111111111110000000000000000 (-2^17)
1 01011010000101111111001100001100 (0x5A17F30C)

01011010000110001111001101001100
+00000000000000010000000000000000(2^{17})
0101 1010 0001 1001 1111 0011 0100 1100 (0x5A19F34C) Vai de 0x5A17F30C até 0x5A19F34C

66. Qual é o menor e o maior endereço para onde uma instrução "**jr**", residente no endereço de memória **0x5A18F34C** pode saltar? Qualquer endereco

67. Qual a gama de representação da constante nas instruções aritméticas imediatas? -2^{15} até $2^{15}-1$

68. Qual a gama de representação da constante nas instruções lógicas imediatas? 0 até $2^{16}-1$

69. Por que razão não existe, no ISA do MIPS, uma instrução que permita manipular diretamente uma constante de 32 bits? Porque o barramento de dados das instruções tem tamanho de 32 bits, era necessário mais bits para poder especificar uma instrução

70. Como é que, no *assembly* do MIPS, se podem manipular constantes de 32 bits? Usando as instruções lui, e ori

71. Apresente a decomposição em instruções nativas das seguintes instruções virtuais:

```
a. li      $6,0x8B47BE0F     lui $6, 0x8B47
                             ori $6, $BE0F
b. xori $3,$4,0x12345678     lui $t0, 0x1234
                             ori $t0, 0x5678
                             xor $3, $4, $t0
c. addi $5,$2,0xF345AB17     lui $t0, 0xF345
                             ori $t0, 0xAV17
                             add $5, $2, $t0
d. beq     $7,100,L1         addi $t0, 100
                             beq $7, $t0, L1
e. blt     $3,0x123456,L2    lui $t0, 0x12
                             ori $t0, 0x3456
                             slt $1, $3, $t0
                             bne $1, $0, L2
```

72. O que é uma sub-rotina? É um excerto de código que é uma função e pode ser chamada e/ou chamadora. Quando esta é chamada retorna à função que a chamou e vice-versa

73. Qual a instrução do MIPS usada para saltar para uma sub-rotina? jal

74. Por que razão não pode ser usada a instrução "j" para saltar para uma sub-rotina? Porque assim quando era feito jr $ra para terminar a subrotina esta iria ser sempre terminal e não iria retornar à função chamadora.

75. Quais as operações que são sequencialmente realizadas na execução de uma instrução "**`jal`**"? É guardado o valor do PC+4 no $ra

76. Qual o nome virtual e o número do registo associado à execução dessa instrução? $ra e registo 31 (Duvida na aula OT pelo Martinho)

77. No caso de uma sub-rotina ser simultaneamente chamada e chamadora (sub-rotina intermédia) que operações é obrigatório realizar nessa sub-rotina? Guardar o $ra na stack e todos os registos que irão ser utilizados antes e depois de uma instrução jal e todos os $s.

78. Qual a instrução usada para retornar de uma sub-rotina? jr $ra

79. Que operação fundamental é realizada na execução dessa instrução? Um salto incondicional para o endereço que esta guarado no registo $ra

80. O que é uma *stack* e qual a finalidade do *stack pointer*? Serve para guardar os valores dos registos internos do CPU que não se quer que sejam alterados por outras subrotinas. A stack pointer aponta sempre para o último valor ocupado na stack

81. Como funcionam as operações de ***push*** e ***pop***? Push: addiu $sp, $sp, -ESPACO, sw $sN, X($sp)

    Pop: lw $sN, X($sp), addiu $sp, $sp, ESPACO

82. Por que razão as *stacks* crescem normalmente no sentido dos endereços mais baixos? Para possibilitar que um programa possa usar mais stack ou mais memoria. Caso contrário haveria uma barreira para a qual não se poderia crescer mais a stack, mesmo que a memória não tivesse a ocupar espaço.

83. Quais as regras para a implementação em software de uma *stack* no MIPS? Reservar espaço necessários para os registos que vao ser salvaguardados e no fim da utilização dar load aos mesmos registos e voltar a repor o espaço reservado

84. Qual o registo usado, no MIPS, como *stack pointer*? $sp

85. De acordo com a convenção de utilização de registos no MIPS:

    a. Que registos são usados para passar parâmetros e para devolver resultados de uma sub-rotina?

    Passar: $a0-$a3 ou $f12 e $f14, Retornar: $v0, $f0

    b. Quais os registos que uma sub-rotina pode livremente usar e alterar sem necessidade de prévia salvaguarda? $t, $a, $v e $f abaixo de 20

    c. Quais os registos que uma sub-rotina não pode alterar? $s, $f acima de 20, $ra e $sp

    d. Quais os registos que uma sub-rotina chamadora tem a garantia que a sub-rotina chamada não altera? $s, e $f acima de 20

    e. Em que situação devem ser usados registos "**$sn**"? Quando queemos usar um registo que é usado antes e depois de uma instrução jal

    f. Em que situação devem ser usados os restantes registos: **$tn**, **$an** e **$vn**? $t é à vontade do freguês, $a para passar argumentos e $v para retornar valores de subrotinas chamadas

86. De acordo com a convenção de utilização de registos do MIPS:

    a. Que registos podem ter que ser copiados para a stack numa sub-rotina intermédia? $s e $f acima de 20

    b. Que registos podem ter que ser copiados para a stack numa sub-rotina terminal? Nenhum

87. Para a função com o protótipo seguinte indique, para cada um dos parâmetros de entrada e para o valor devolvido, qual o registo do MIPS usado para a passagem dos respetivos valores: `char fun(int a, unsigned char b, char *c, int *d);` $a0, $a1, $a2, $a3 e $v0

88. Para uma codificação em complemento para 2, apresente a gama de representação que é possível obter com **3**, **4**, **5**, **8** e **16** bits (indique os valores-limite da representação em binário, hexadecimal e em decimal com sinal e módulo).

    Para 3 bits: Binario 100 até 011, Hexadecimal – 0xC até 0x3, Decimal -8 até 7

    Para 4 bits: Binario 1000 até 0111, Hexadecimal – 0x8 até 0x7, Decimal -16 até 15

    Para 5 bits: Binario 10000 até 01111, Hexadecimal – 0xF0 até 0x0F, Decimal -32 até 31

    Para 8 bits: Binario 10000000 até 01111111, Hexadecimal – 0x80 até 0x7F, Decimal -256 até 255

89. Traduza para *assembly* do MIPS a seguinte função "**`fun1()`**", aplicando a convenção de passagem de parâmetros e salvaguarda de registos:

```
char *fun2(char *, char);

char *fun1(int n, char *a1, char *a2)
{   int j = 0;
     char *p =a1;
     do
     {  if((j % 2) == 0)
           fun2(a1++,*a2++);
     } while(++j < n);
     *a1='\0';
    return p;
}
```

```
fun1:                               #char *fun1(int n, char *a1, char *a2) {
    addiu   $sp, $sp, -24           #   por espaco na pilha
    sw      $ra, 0($sp)             #   salvaguardar $ra na pilha
    sw      $s0, 4($sp)             #   salvaguardar o $s0
    sw      $s1, 8($sp)             #   salvaguardar o $s1
    sw      $s2, 12($sp)            #   salvaguardar o $s2
    sw      $s3, 16($sp)            #   salvaguardar o $s3
    sw      $s4, 20($sp)            #   salvaguardar o $s4
    move    $s4, $a0                #   $s4 = n;
    move    $s2, $a1                #   $s2 = *a1;
    move    $s3, $a2                #   $s3 = *a2;
    li      $s0, 0                  #   j = 0;
    move    $s1, $a0                #   *p = a1;
do:                                 #   do {
    rem     $t0, $s0, 2             #       $t0 = j%2;
if: bne     $t0, 0, endif           #       if($t0 == 0) {
    move    $a0, $s2                #           arg1 = *a1;
    lb      $a1, 0($s3)             #           arg2 = a2;
    jal     fun2                    #           fun(a1, a2);
    addiu   $s2, $s2, 1             #           a1++;
    addiu   $s3, $s3, 1             #           a2++;
endif:                              #       }
    addi    $s0, $s0, 1             #       j++;
while:ble   $s0, $s4, do            #   while(j < n);
    sb      '\0', 0($s2)            #   *a1 = '\0';
    move    $v0, $s1                #   return p;
    lw      $ra, 0($sp)             #   salvaguardar $ra na pilha
    lw      $s0, 4($sp)             #   salvaguardar o $s0
    lw      $s1, 8($sp)             #   salvaguardar o $s1
    lw      $s2, 12($sp)            #   salvaguardar o $s2
    lw      $s3, 16($sp)            #   salvaguardar o $s3
    lw      $s4, 20($sp)            #   salvaguardar o $s4
    addiu   $sp, $sp, 24            #   repor o espaco na pilha
    jr      $ra                     #}
```

90. Determine a representação em complemento para 2 com 16 bits das seguintes quantidades:

5, -3, -128, -32768, 31, -8, 256, -32

0000000000000101 (5), 1111111111111101 (-3), 1111111110000000 (-128), 1000000000000000 (-32768), 0000000000011111 (31), 1111111111111000 (-8), 0000000100000000 (256), 1111111111100000 (-32)

91. Determine o valor em decimal representado por cada uma das quantidades seguintes, supondo que estão codificadas em complemento para 2 com 8 bits:

0b00101011, 0xA5, 0b10101101, 0x6B, 0xFA, 0x80

0b00101011 = 43, 0XA5 = 1010 0101 = -(1+2+8+16+64) = -91, 0b10101101 = -128 + 32 + 8+4+1 = -83, 0x6B = 0110 1011 = 6 x 16 + 11 = 107, 0xFA = 1111 1010 = -8 + 2 = -6, 0x80 = 1000 0000 = -128

92. Determine a representação das quantidades do exercício anterior em hexadecimal com 16 bits (também codificadas em complemento para 2).

0b00101011 = 0x002B, 0xA5 = 0xFFA5, 0b10101101 = 0xFFAD, 0x6B = 0x006B, 0x80 = 0xFF80

93. Como é realizada a deteção de *overflow* em operações de adição com quantidades sem sinal? O overflow ocorre se houver um bit de carry-out no fim da adição.

94. Como é realizada a deteção de *overflow* em operações de adição com quantidades com sinal (codificadas em complemento para 2)? O overflow é detectado se o but mais significativo é diferente do carry-out.

95. Considere os seguintes pares de valores em **`$s0`** e **`$s1`**:

```
i.  $s0 = 0x70000000 $s1 = 0x0FFFFFFF
ii. $s0 = 0x40000000 $s1 = 0x40000000
```

a. Qual o resultado produzido pela instrução **`add $t0, $s0, $s1`**? 0x7FFFFFFF, 0x80000000

b. Para a alínea anterior os resultados são os esperados ou ocorreu *overflow*? Deu overflow na segunda alínea porque a soma de dois positivos deu negativo

c. Qual o resultado produzido pela instrução **`sub $t0, $s0, $s1`**?

`0x70000000 - 0x0FFFFFFF = 0x60000001 e 0x00000000`

d. Para a alínea anterior os resultados são os esperados ou ocorreu *overflow*? São os esperados os dois

e. Qual o resultado produzido pelas instruções:

add $t0, $s0,$s1 0x7FFFFFFF

add $t0, $t0,$s1 0x7FFFFFFF + 0x0FFFFFFF = 0x8FFFFFFE

f. Para a alínea anterior os resultados são os esperados ou ocorreu *overflow*? Ocorreu overflow na segunda situacao

96. Para a multiplicação de dois operandos de "**m**" e "**n**" bits, respetivamente, qual o número de bits necessário para o armazenamento do resultado? O dobro de bits do maior entre m e n

97. Apresente a decomposição em instruções nativas das seguintes instruções virtuais:

a. **mul $5,$6,$7**
```
mul $6, $7
mflo $5
```

b. **la $t0,label c/ label = 0x00400058**
```
lui  $t0, 0x0040
ori  $t0, 0x0058
```

c. **div $2,$1,$2**
```
div $1, $2
mflo $2
```

d. **rem $5,$6,$7**
```
div $6, $7
 mfhi $5
```

e. **ble $8,0x16,target**
```
slti $1, 0x16, $8
beq $1, $0, target
```

f. **bgt $4,0x3F,target**
```
slti $1, $4, 0x3F
bne  $1, $0, target
```

98. Determine o resultado da instrução **mul $5,$6,$7**, quando **$6=0xFFFFFFFE e $7=0x00000005.**

```
 11111111111111111111111111111110
x00000000000000000000000000000101
|11111111111111111111111111111110
+11|11111111111111111111111111110__
100|11111111111111111111111110110    Dá overflow $5 = 0xFFFFFF6
```

99. Determine o resultado da execução das instruções virtuais **div $5,$6,$7 e rem $5,$6,$7** quando **$6=0xFFFFFFFE e $7=0x00000003** Da 0x00000000 e 0xFFFFFFE porque 0x03 é maior em modulo que 0xFE

100. Admita que pretendemos executar, em *Assembly* do MIPS, as operações:

     `$t0 = $t2/$t3` e **`$t1 = $t2 % $t3`**.

     Escreva a sequência de instruções em *Assembly* que permitem realizar estas duas operações. Use apenas instruções nativas

     div $t2, $t3

     mflo $t0

     mfhi $t1

101. Descreva as regras que são usadas, na ALU do MIPS, para realizar uma divisão inteira entre duas quantidades com sinal. Para multiplicar 2 signed integer divide-se o dividendo pelo divisor em módulo. O quociente tem sinal negativo se os sinais do dividendo e divisor forem diferentes. O resto tem o sinal do dividendo. Dividendo = Divisor * Quociente + resto.

102. Considerando que **`$t0=-7`** e **`$t1=2`**, determine o resultado da instrução **`div $t0,$t1`** e o valor armazenado respetivamente nos registos **`HI`** e **`LO`**.

     -3 no quociente e -1 no resto, logo HI = 0xFFFFFFFF LO = 0xFFFFFFFD (11111111111111111111111111111101)

103. Repita o exercício anterior admitindo agora que **`$t0=0xFFFFFFF9`** e **`$t1=0x00000002.`**

     3 no quociente e -1 no resto, logo HI = 0xFFFFFFFF LO = 0xFFFFFFFD (11111111111111111111111111111101)

104. Considerando que **`$5=-9`** e **`$10=2`**, determine o valor que ficará armazenado no registo destino pela instrução virtual **`rem $6, $5, $10`**. $6 = 0xFFFFFFFF

105. Para a implementação de uma arquitetura de multiplicação de 32 bits são necessários, entre outros, registos para o multiplicador e multiplicando, e ainda uma ALU. Determine a dimensão exata, em bits, de cada um destes três elementos funcionais. 64 de multiplicador 32 de multiplicando e 32 na ALU

106. As duas sub-rotinas seguintes permitem detetar *overflow* nas operações de adição com e sem sinal, no MIPS. Analise o código apresentado e determine o resultado produzido, pelas duas sub-rotinas, nas seguintes situações:

     a. **`$a0=0x7FFFFFF1, $a1=0x0000000E; $v0 = 0; $v0 = 0`**
     b. **`$a0=0x7FFFFFF1, $a1=0x0000000F; $v0 = 0; $v0 = 0`**
     c. **`$a0=0xFFFFFFF1, $a1=0xFFFFFFFF; $v0 = 0; $v0 = 1`**
     d. **`$a0=0x80000000, $a1=0x80000000; $v0 = 1; $v0 = 1`**

```
# Overflow detection, signed
# int isovf_signed(int a, intb);
isovf_signed: ori $v0,$0,0
```

```
                xor $1,$a0,$a1
                slt $1,$1,$0
                bne$1,$0,notovf_s
                addu $1,$a0,$a1
                xor $1,$1,$a0
                slt $1,$1,$0
                beq$1,$0,notovf_s
                ori $v0,$0,1
notovf_s:  jr $ra


# Overflow detection, unsigned
# int isovf_unsigned(unsigned int a, unsigned int b);
isovf_unsig: ori $v0,$0,0
             nor $1,$a1,$0
             sltu $1,$1,$a0
             beq $1,$0,notovf_u
             ori $v0,$0,1
notovf_u:  jr $ra
```

107. As duas sub-rotinas anteriores podem ser também escritas alternativamente com o código abaixo. A abordagem á ligeiramente diferente. No caso de operações sem sinal, o *overflow* pode ser detetado para as operações de soma e subtração. Analise o código apresentado e determine o resultado produzido, pelas duas sub-rotinas, nas condições indicadas nas alíneas da questão anterior:

```
# Overflow detection in addition, unsigned
# int isovf_unsigned_plus(unsigned int a, unsigned int b);
isovf_unsig_plus:
             ori  $v0, $0, 0
             addu $t2, $a0, $a1  # temp = A + B;
            bge  $t2, $a0, notovf_uadd
             bge  $t2, $a1, notovf_uadd
             ori  $v0, $0, 1
notovf_uadd:  jr $ra

# Overflow detection in subtraction, unsigned
# int isovf_unsigned_sub(unsigned int a, unsigned int b);
isovf_unsig_sub:
            ori $v0,$0,0
            slt $1, $a0, $a1
            beq $1, $0, notovf_usub
             ori  $v0, $0,1
notovf_usub:  jr $ra

# Overflow detection, signed  # int isovf_signed(int a, int b);  isovf_signed:  ori $v0,$0,0
```

```
        add  $1, $a0, $a1      # res = a + b;
        xor  $a1, $a0, $a1     # tmp = a ^ b;
        bltz $a1, notovf_s     # if (tmp < 0)no_ovf();
        xor  $a1, $1, $a0      # tmp = res ^ a;
        bgez $a1, notovf_s     # if (tmp >= 0) no_ovf();
        ori $v0,$0,1
notovf_s:  jr $ra
```

108. Ainda no código das sub-rotinas das questões anteriores, qual a razão para não haver salvaguarda de qualquer registo na stack? Porque as subrotinas são terminais, ou seja não são chamadoras.

109. Na conversão de uma quantidade codificada em formato IEEE 754, precisão simples, para decimal, qual o número máximo de casas decimais com que o resultado deve ser apresentado? 6 casas decimais

110. Responda à questão anterior admitindo que o valor original se encontra agora representado com precisão dupla no formato IEEE 754. 15 casa decimais

111. Determine a representação em formato IEEE 754, precisão simples, da quantidade real $19{,}1875_{10}$. Determine a representação da mesma quantidade em precisão dupla.

$19_{10} = 10011_2$ $0{,}1875_{10} = 00110_2$
$10011.00110 = 1.001100110 \times 2^4$

Single
Sinal = 0, Expoente = 4 => 127 + 4 = 131, Mantissa = 001100110
0 10000011 00110011000000000000000

Double
Sinal = 0, Expoente = 4 => 1023 + 4 = 1027, Mantissa = 001100110
0 10000000011 0011001100000000000000000000000000000000000000000000

112. Determine, em decimal (vírgula fixa), o valor das quantidades representadas em formato IEEE 754, precisão simples. Na alínea b) apresente apenas o valor em notação científica usando base 2.

a. 0xC19A8000.

1 10000011 00110101000000000000000

$-1.00110101000000000000000 \times 2^{131-127=4}$

10011.01010000000000000000 = -19,3125

0,25 + 0,0625 = 0,3125

b. 0x80580000.

1 00000000 10110000000000000000000

$-0.10110000000000000000000 \times 2^{-126}$

113. Considere que o conteúdo dos dois seguintes registos da FPU representam a codificação de duas quantidades reais no formato IEEE754 precisão simples:

```
$f0 = 0x416A0000
$f2 = 0xC0C00000
```

Calcule o resultado das instruções seguintes, apresentando o seu resultado em hexadecimal:

a. `abs.s $f4,$f2` # $f4 = abs($f2)

0x40C00000 (MSB fica a 0)

b. `neg.s $f6,$f0` # $f6 = neg($f0)

0xC16A0000 (Troca o sinal MSB)

c. `sub.s $f8, $f0,$f2` # $f8 = $f0 – $f2

$f0 = 0100 0001 0110 1010 0000 0000 0000

0 10000010 11010100000000000000000

$1.11010100000000000000000 \times 2^3$ Normalizada

$f2 = 1 10000001 10000000000000000000000

$-1.10000000000000000000000 \times 2^2$ Normalizada

$-0.11000000000000000000000 \times 2^3$ Desnormalizada

$1.11010100000000000000000 \times 2^3$

$+0.11000000000000000000000 \times 2^3$

$10.10010100000000000000000 \times 2^3 = 1.01001010000000000000000 \times 2^4$

0(Sinal) 10000011(127+4) 01001010000000000000000(Mantissa)

0x41A50000

d. `sub.s $f10,$f2,$f0` # $f10 = $f2 - $f0

0xC1A50000

e. `add.s $f12,$f0,$f2` # $f12 = $f0 + $f2

$1.11010100000000000000000 \times 2^3$

$-0.11000000000000000000000 \times 2^3$

$1.00010100000000000000000 \times 2^3$

0(Sinal) 10000010(127+3) 00010100000000000000000(Mantissa)

0x410A0000

f. mul.s $f14,$f0,$f2 # $f14 = $f0 * $f2

$1.11010100000000000000000 \times 2^{3}$

$x1.10000000000000000000000 \times 2^{2}$

11101010000000000000000

+11101010000000000000000X

$10.10111110000000000000000 \times 2^{5} = 1.01011111000000000000000 \times 2^{6}$

1(Sinal) 10000101(127+6) 01011111000000000000000(Mantissa)

0xC2AF8000

g. div.s $f16,$f0,$f2 # $f16 = $f0 / $f2

1.11010100000000000000000 0,5+0,25+0,0625+0,015625 = 0,828125

1.10000000000000000000000 0,5

1,828125 / 1,5 = 1,21875 mantissa 1.00111000000000000000000

Expoente 2^{3} / 2^{2} = 2^1

1 10000000 00111000000000000000000

0xC01C0000

h. div.s $f18,$f2,$f0 # $f18 = $f2 / $f0

?

i. cvt.d.s $f20,$f2 # Convert single to double

$f2 = 1 1 0000001 10000000000000000000000

Sinal:1; Expoente:2; Mantissa: 10000000000000000000000

Double expoente: 1023 + 2 = 1025

1 10000000001 1000000000000000000000000000000000000000000000000000

(0xC018000000000000)

j. cvt.w.s $f22,$f0 # Convert single to integer

$1.11010100000000000000000 \times 2^{3}$

1110.10100000000000000000 c/ truncatura 1110.00000000000000000000

0x0000000E

114. Considere a sequência de duas instruções Assembly: **`lui $t0,0xC0A8 mtc1 $t0,$f8`**

qual o valor que ficará armazenado no registo **`$f8,`** expresso em base dez e vírgula fixa, admitindo uma interpretação em IEEE 754 precisão simples?

0xC0A80000 = 1 10000001 01010000000000000000000
- 1.01010000000000000000000x2^{2}
- 101.01000000000000000000$_{2}$ = -5,25$_{10}$

115. Considerando que **$f2=0x3A600000** e **$f4=0xBA600000**, determine o resultado armazenado em **$f0** pela instrução **`sub.s $f0,$f2,$f4`**.
$f2=0x3A600000 = 0011 1010 0110 0000 0000 0000 0000 0000
$f4=0xBA600000 = 1011 1010 0110 0000 0000 0000 0000 0000

0 01110100 11000000000000000000000
1.11000000000000000000000x2^{-11}
1 01110100 110 0000 0000 0000 0000 0000
-1.11000000000000000000000x2^{-11}

Mesmo número com sinais diferentes resultado da 0x00000000

116. Repita o exercício anterior admitindo agora as seguintes condições:

$f4=0x3F100000 e **$f6=0x408C0000** e a instrução **`add.s $f8,$f4,$f6.`**
0x3F100000 = 0011 1111 0001 0000 0000 0000 0000 0000
0 01111110 001 0000 0000 0000 0000 0000
1.00100000000000000000000 x2^{-1}
0.00100100000000000000000 x 2^{2}

0x408C0000 = 0100 0000 1000 1100 0000 0000 0000 0000
0 10000001 00011000000000000000000
1.00011000000000000000000 x2^{2}

0.00100100000000000000000
+1.00011000000000000000000
1.00111100000000000000000

0 10000001 00111100000000000000000 (0x409E0000)

$f2=0x3F900000 e **$f4=0xBEA00000** e a instrução **`mul.s $f0,$f2,$f4`**

0x3F900000 = 0 01111111 00100000000000000000000
1.00100000000000000000000x2^{0}

0xBEA00000 = 1 01111101 01000000000000000000000
-1.01000000000000000000000x2^{-2}

1.00100000000000000000000x2^{0}
x1.01000000000000000000000x2^{-2}
100100000000000000000000
000000000000000000000000X
100100000000000000000000XX
1.01101000000000000000000x2^{-2}

1 01111101 01101000000000000000000 (0xBEB40000)
$f2=0x258c0000 e **$f4=0x41600000** e a instrução **div.s $f0,$f2,$f4**

117. Numa norma hipotética KPT de codificação em vírgula flutuante, a mantissa normalizada após a realização de uma operação aritmética tem o valor **1.1111 1111 1111 1110 1000 0000**. Qual será o valor final da mantissa (com 16 bits na parte fracionária) após arredondamento para o ímpar mais próximo?

118. Assuma que **x** é uma variável do tipo **float** residente em **$f8** e que o *label* **endWhile** corresponde ao endereço da primeira instrução imediatamente após um ciclo *while()*. Se a avaliação da condição para executar o *loop* for *while (x > 1.5){..}* escreva, em Assembly do MIPS, a sequencia de instruções necessárias para determinar esta condição.

```
variable: .float 1.5
l.s $f6, variable
c.le.s $f8, $f6
bc1t endWhile
```

119. Determine, de acordo com o formato IEEE 754 precisão simples, a representação normalizada, e arredondada para o par mais próximo, do número $100,1101100000000000010110_2$.

100.11011000000000000010110 = 1.00110110000000000000101 100 x2²

Arredondado 1.00110110000000000000110

120. Numa implementação *single cycle* da arquitetura MIPS, a frequência máxima de operação é de 2GHz (para os atrasos de propagação a seguir indicados). Determine o atraso máximo que pode ocorrer nas operações da ALU. Considere que, para o *File Register* e para as memórias, os tempos de escrita indicados são os tempos de preparação da operação antes de uma transição ativa do sinal de relógio.

Como é freq maxima temos que contar com a instrução mais demorada LW e T = 500ps

$T_{EXEC} = T_{RM} + \max(T_{RFF}, T_{CNTL}, T_{SE}) + T_{ALU} + T_{RM} + T_{WRF}$

$T_{EXEC} = 175 + 175 + 45 + 15 + T_{ALU} = 410 + T_{ALU}$ ps

$T_{ALU} = 500 - 410 = 90$ ps

Memórias externas: leitura – 175ps, escrita – 120ps; *File register*: leitura – 45ps, Escrita – 15ps;
Unidade de Controlo: 10ps; Somadores: 40ps; Outros: 0ns; Setup time do *Program Counter*: 5ps

121. Determine, numa implementação *single-cycle* da arquitetura MIPS, a frequência máxima de operação imposta pela instrução "sw", assumindo os atrasos a seguir indicados: 40MHz

$T_{EXEC} = T_{RM} + \max(T_{RFF}, T_{CU}, T_{SE}) + T_{ALU} + T_{WM}$

$T_{EXEC} = 12 + \max(4, 1, 0) + 5 + 4 = 12 + 4 + 5 + 4 = 25$ns

Memórias externas: leitura – 12ns, escrita – 4ns; File register: leitura – 4ns, Escrita – 1ns;
Unidade de Controlo: 1ns; ALU (qualquer operação): 5ns; Somadores: 2ns; Outros: 0ns.
Setup time do *Program Counter*: 1ns

122. Determine, numa implementação *single-cycle* da arquitetura MIPS, a frequência máxima de operação imposta pela instrução "beq", assumindo os atrasos a seguir indicados, é: 50 MHz

T_{beq}= TRM + max(max(Rrf,ControlUnit)+Alu(cálculo do 0), TSe + TShiftLeft2+Tadd(tempo de cálculo do BTA) + tPC

T_{beq} = 11 + max(max(3,1) + 5, 0 + 0+2) + 1 = 11 + 8 +1 = 20ns

Memórias externas: leitura – 11ns, escrita – 3ns; File register: leitura – 3ns, Escrita – 1ns;
Unidade de Controlo: 1ns; ALU (qualquer operação): 5ns; Somadores: 2ns; Outros: 0ns.
Setup time do *Program Counter*: 1ns

123. Determine, numa implementação *single cycle* da arquitetura MIPS, o período mínimo do sinal de relógio imposto pelas instruções tipo R, assumindo os atrasos a seguir indicados, é: 45 MHz

$T_{rtypeinstruction}$= TRM + max(RFr,CU) + TALU + WFR

$T_{rtypeinstruction}$ = 12 + max(3,1) + 6 + 1 = 12 + 6 + 3 + 1 = 22ns

Memórias externas: leitura – 12ns, escrita – 4ns; File register: leitura – 3ns, Escrita – 1ns;
Unidade de Controlo: 1ns; ALU (qualquer operação): 6ns; Somadores: 2ns; Outros: 0ns.
Setup time do *Program Counter*: 1ns

124. Identifique os principais aspetos que caracterizem uma arquitetura *single cycle*, quer do ponto de vista do modelo da arquitetura, como das características da sua unidade de controlo. Utilização de 2 memórias, cada instrução demora um ciclo de relógio a correr. A unidade de controlo é uma unidade combinatória. Várias instruções usam os mesmos componentes

125. Numa implementação *single cycle* da arquitetura MIPS, no decurso da execução de uma qualquer instrução, a que corresponde o valor presente na saída do registo PC? O endereço da instrução a ser executada a seguir

126. Preencha a tabela seguinte, para as instruções indicadas, com os valores presentes à saída da unidade de controlo principal da arquitetura *single cycle* dada nas aulas.

Instrução	Opcode	ALUOp[1..0]	Branch	RegDst	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem WRite
lw (imm)	100011	00(soma)	0	0(rt)	1	1	1	1	0
sw (imm)	101011	00	0	x	1	X	0	0	1
addi (imm)	001000	00	0	0(rt)	1	0	1	0	0
slti	001010	11(slti)	0	0	1	0	1	0	0
beq	000100	01(sub)	1	X	1	X	0	0	0
R - Format	000000	10(AluC decide)	0	1	0	0	1	0	0

127. Admita que na versão *single cycle* do CPU MIPS dado nas aulas, pretendíamos acrescentar o suporte das instruções **`jal address`** e **`jr $reg`**. Esquematize as alterações que teria de introduzir no *datapath* para permitir a execução destas instruções (use o esquema da próxima página). Ver aulas guardadas

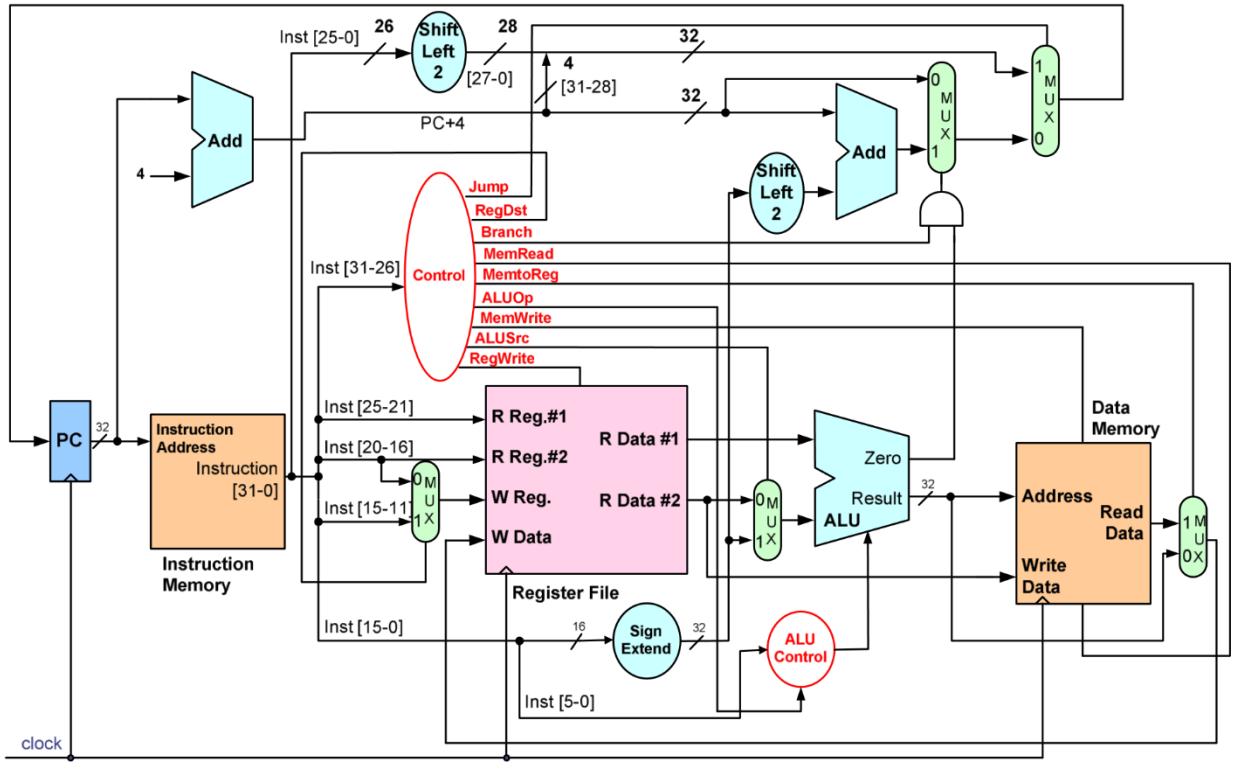


Fig. 1 - Datapath single-cycle

128. Admita que na versão *single cycle* do CPU MIPS, pretendíamos executar a instrução **`slt $3,$5,$9`**. Descreva por palavras suas como é esta instrução realizada ao nível da ALU, e qual o conteúdo final no registo **`$3`**, admitindo que **`$5=0xFF120008`** e **`$9=0x00C00FFF`**. $3 = 0x00000001

A ALU faz uma subtração entre os valores lidos dos registos fonte. Se o resultado der positivo, ou seja, se o MSB for 0 esse é o valor no registo destino. O mesmo acontece para o negativo

129. Suponha que os tempos de atraso introduzidos pelos vários elementos funcionais de um *datapath single-cycle* são os seguintes:

Acesso à memória para leitura (tRM):	12ns	Acesso à memória para preparar escrita (tWM):	4ns
Acesso ao register file para leitura (tRRF):	5ns	Acesso ao register file para preparar escrita (tWRF):	2ns
Operação da ALU (tALU):	7ns	Operação de um somador (tADD):	2ns
Multiplexers e restantes elementos funcionais:	0ns	Unidade de controlo (tCNTL):	2ns
Tempo de setup do PC (tstPC):	1ns		

a. Determine o tempo mínimo para execução das instruções tipo **`R`**, **`LW`**, **`SW`**, **`BEQ`** e **`J`**.

R: Texec = TReadMemory + max(TRRf,TCntl) + tAlu + TWRf = 12 + 5 + 2 + 7 = 26ns

LW: Texec = TReadMemory + max(TRRf,Tcntl) + TAlu + TRM + TWRF (MAIOR DE TODAS) = 12 + 5+ 7+ 12 + 2 = 38 ns

SW: Texec = TReadMemory + max(Trrf,TCntl) + Talu + TWM = 12 + 5 + 7 + 4 = 28ns

BEQ: Texec = TreadMemory +max(max(RRF,TCntl) + TALU, tADD) + TsPC = 12 + 12 +1 = 25ns

J: Texec = TreadMemory + max(CU,Sl2) + TsPc = 12 + 2 + 1 = 15 ns

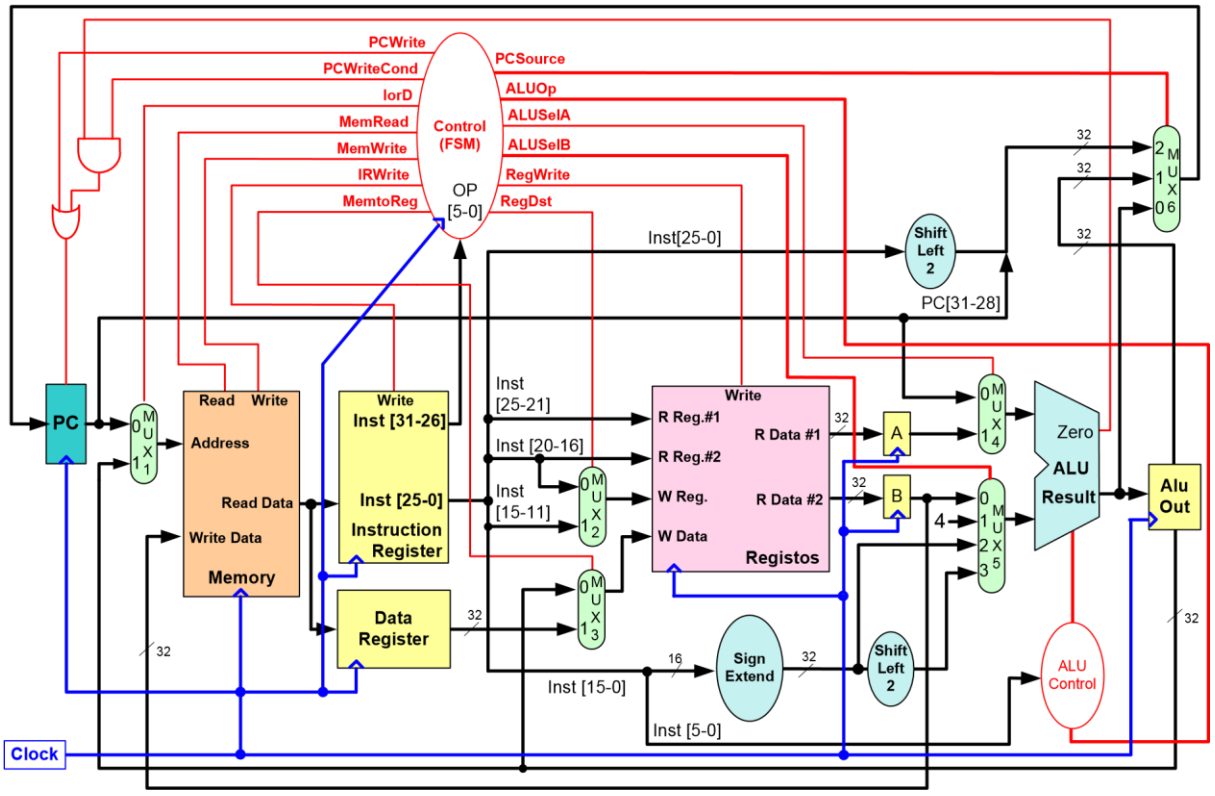
b. Calcule a máxima frequência do relógio que garanta uma correta execução de todas as instruções.

26MHz(instrução lw(mais longa))

130. Suponha agora que dispunha de uma tecnologia que que o período de relógio podia ser adaptado instrução a instrução, em função da instrução em curso. Determine qual o ganho de eficiência que poderia obter com esta tecnologia face a uma tecnologia em que a frequência do relógio é a que obteve na questão anterior (admita os mesmos atrasos de propagação). Para isso, assuma que o programa de *benchmarking* tem a seguinte distribuição de ocorrência de instruções: Tem um ganho de 1.4

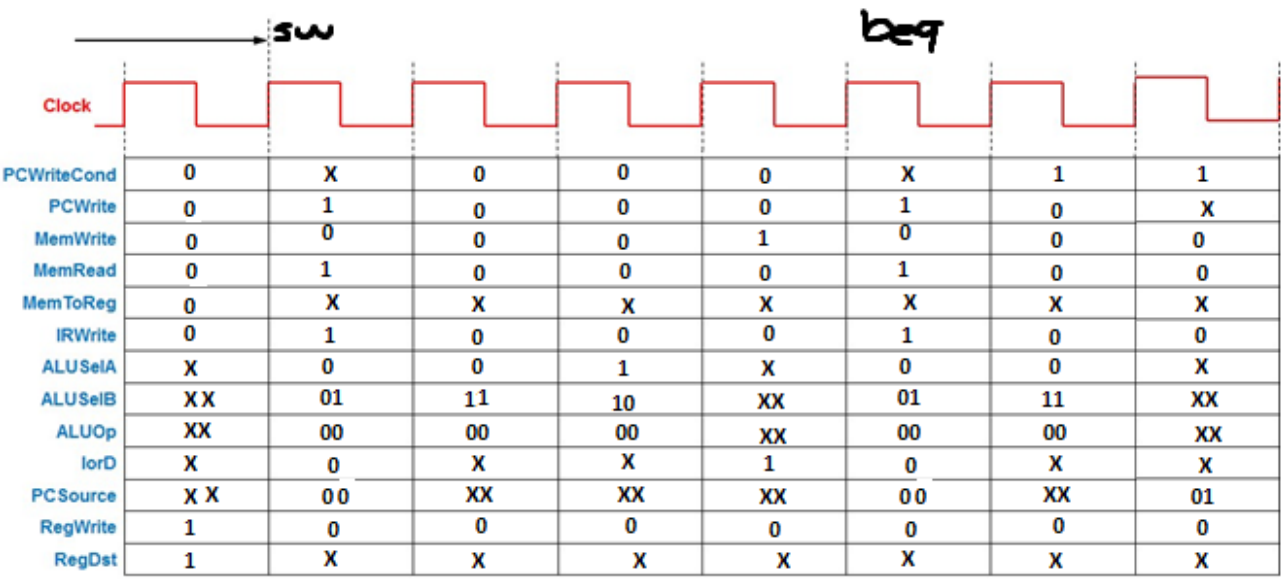
15% de **`lw`**, 15% de **`sw`**, 40% de tipo **`R`**, 20% de **`branches`** e 10% de **`jumps`**

131. Ainda para os tempos utilizados nas duas questões anteriores, determine qual a máxima frequência de trabalho no caso de o *datapath* ser do tipo *multi-cycle*. 83 MHz fmax = 1/tempo de elemento mais lento



132. Considere o *datapath multi-cycle* presente na figura anterior e a respetiva unidade de controlo. Preencha a tabela abaixo considerando que a coluna da esquerda corresponde ao último ciclo de execução de uma instrução, e que a sequencia em causa é a seguinte:

```
a. add  $t0, $t2, $t1
   sw   $t0, 0($t3)
   beq  $t0, $t1, next
```



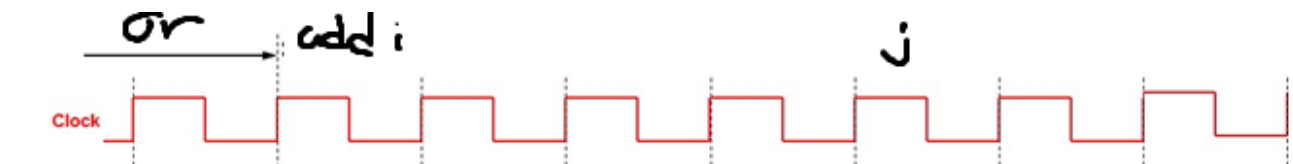
PCWriteCond	0	X	0	0	0	X	1	1
PCWrite	0	1	0	0	0	1	0	X
MemWrite	0	0	0	0	1	0	0	0
MemRead	0	1	0	0	0	1	0	0
MemToReg	0	X	X	X	X	X	X	X
IRWrite	0	1	0	0	0	1	0	0
ALUSelA	X	0	0	1	X	0	0	X
ALUSelB	XX	01	11	10	XX	01	11	XX
ALUOp	XX	00	00	00	XX	00	00	XX
IorD	X	0	X	X	1	0	X	X
PCSource	XX	00	XX	XX	XX	00	XX	01
RegWrite	1	0	0	0	0	0	0	0
RegDst	1	X	X	X	X	X	X	X

133. Repita o exercício anterior para as seguintes sequências de instrução:

a.	b.	c.

`or $t0, $0, $t1`
`addi $t0, $t1, 0x20`
`j label`
	`lw $s0, 0($t1)`
	`lw $s1, 4($t1)`
	`add $t2, $s1, $s2`
		`sw $t0, 0($t1)`
		`sub $t0, $t3, $t2`
		`slt $t1, $t0, $t2`

a)



PCWriteCond	0	X	0	0	0	X	0	0
PCWrite	0	1	0	0	0	1	0	1
MemWrite	0	0	0	0	0	0	0	0
MemRead	0	1	0	0	0	1	0	0
MemToReg	X	X	X	X	0	X	X	X
IRWrite	0	1	0	0	0	1	0	0
ALUSelA	X	0	0	1	X	0	0	X
ALUSelB	XX	01	11	10	XX	01	11	XX
ALUOp	XX	00	00	00	XX	00	00	XX
IorD	X	0	X	X	X	0	X	X
PCSource	XX	0	XX	XX	XX	0	XX	10
RegWrite	1	0	0	0	1	0	0	X
RegDst	1	X	X	X	0	X	X	X

b)

PCWriteCond	0	X	0	0	0	0	X	0
PCWrite	0	1	0	0	0	0	1	0
MemWrite	0	0	0	0	0	0	0	0
MemRead	0	1	0	0	1	0	1	0
MemToReg	1	X	X	X	X	1	X	X
IRWrite	0	1	0	0	0	0	1	0
ALUSelA	X	0	0	1	X	X	0	0
ALUSelB	XX	01	11	10	XX	XX	01	11
ALUOp	XX	00	00	00	XX	XX	00	00
IorD	X	0	X	X	1	X	0	X
PCSource	XX	00	XX	XX	XX	XX	00	XX
RegWrite	1	0	0	0	0	1	0	0
RegDst	0	X	X	X	X	0	X	X

c)

PCWriteCond	0	X	0	0	0	X	0	0
PCWrite	0	1	0	0	0	1	0	0
MemWrite	1	0	0	0	0	0	0	0
MemRead	0	1	0	0	0	1	0	0
MemToReg	X	X	X	X	0	X	X	X
IRWrite	0	1	0	0	1	1	0	0
ALUSelA	X	0	0	1	X	0	0	1
ALUSelB	XX	01	11	00	XX	01	11	0
ALUOp	XX	00	00	10	XX	00	00	10
IorD	1	0	X	X	X	0	X	X
PCSource	XX	00	XX	XX	XX	00	XX	XX
RegWrite	0	0	0	0	1	0	0	0
RegDst	X	X	X	X	1	X	X	X

134. Para as mesmas sequências de instruções apresentadas nos dois exercícios anteriores, preencha, na forma de um diagrama temporal, a tabela seguinte. X

135. Ainda para as mesmas sequências de instruções apresentadas nos três exercícios anteriores, preencha a tabela abaixo com os valores presentes à saída da ALU e dos elementos de estado indicados. Consulte a tabela da última página se necessário. Admita que, no início de cada sequência, o conteúdo dos registos relevantes é o seguinte:

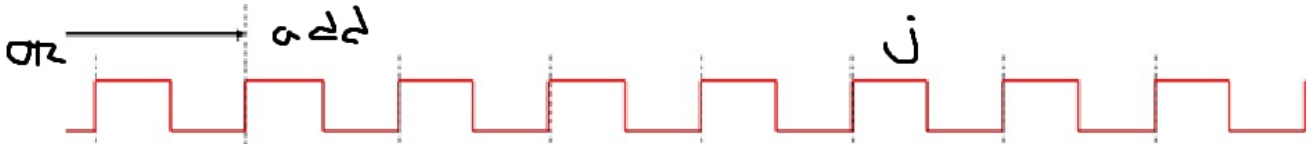
[$t0=0x000013FC],[$t1=0x],[$t2=0x90FFFF64],[$t3=0x00000028] e que na memória [(0x10010000)=0x00000020] e [(0x10010004)=0x00000038]

a. or $t0, $0, $t1 addi $t0,$t1,0x20 j label

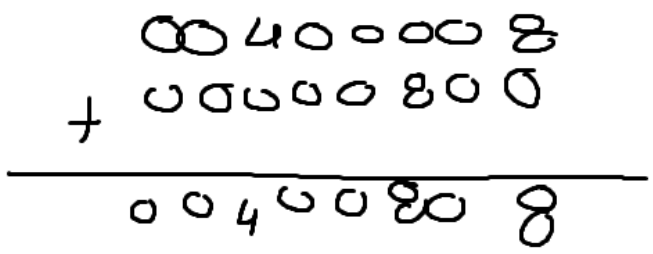
or $t0, $0, $t1
000000 00000 01001 01000 00000 100101 = 0x00094025

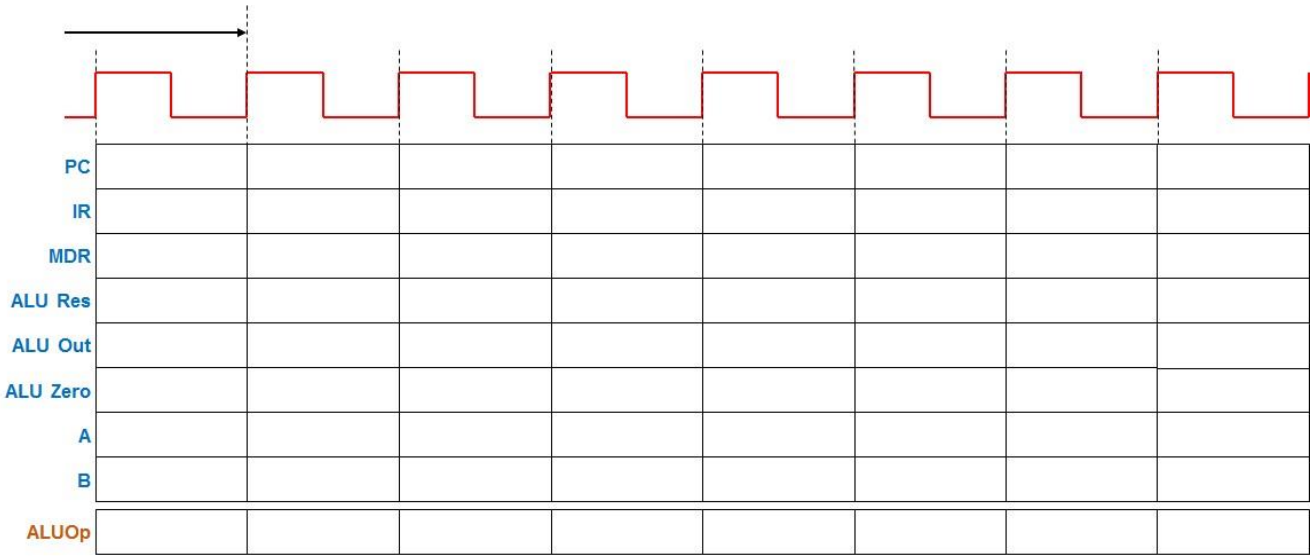
addi $t0,$t1,0x20
001000 01001 01000 0000000000110000 = 0x21280030

J label = WTF?



PC	0x00400004	0x00400004	0x00400008	0x00400008	0x00400008	0x00400008	0x0040000C	0x0040000C
IR	0x00094025	0x00094025	0x21280030	0x21280030	0x21280030	?	?	?
MDR	?	?	0x21280030	?	?	?	?	?
ALU Res	?	0x00400008	0x00400808	10010020	?	0x0040000C	BTA	?
ALU Out	10010000	?	0x00400008	0x00400808	10010020	?	0x0040000C	BTA
ALU Zero	?	0	0	0	?	0	0	0
A	10010000	10010000	10010000	10010000	10010000	10010000	10010000	10010000
B	00000000	00000000	00000000	000013FC	000013FC	000013FC	000013FC	000013FC
ALUOp	xx	00	00	10	xx	xx	00	00





136. Calcule o número de ciclos de relógio que o programa seguinte demora a executar, desde o *Instruction fetch* da 1ª instrução até à conclusão da última instrução:

a. num *datapath single-cycle* 35 ciclos

b. num *datapath multi-cycle* 143 ciclos

```
main:                   # p0 = 0;
    lw    $1,0($0) # p1 = *p0 = 0x10;
    add $4,$0,$0 # v = 0;
    lw    $2,4($0) # p2=*(p0+1)=0x20;
loop:                   # do {
    lw    $3,0($1) #  aux1 = *p1;
    add $4,$4,$3 #   v = v + *p1;
    sw    $4,36($1) #      *(p1 + 9) = v;
    addiu $1,$1,4    #     p1++;
    sltu $5,$1,$2 #
    bne $5,$0,loop # } while(p1 < p2);
    sw    $4,8($0) # *(p0 + 2) = v;
    lw    $1,12($0) # aux2 = *(p0 + 3);
```

Memória de dados

Address	Value
0x0000000	0x10
0x0000004	0x20

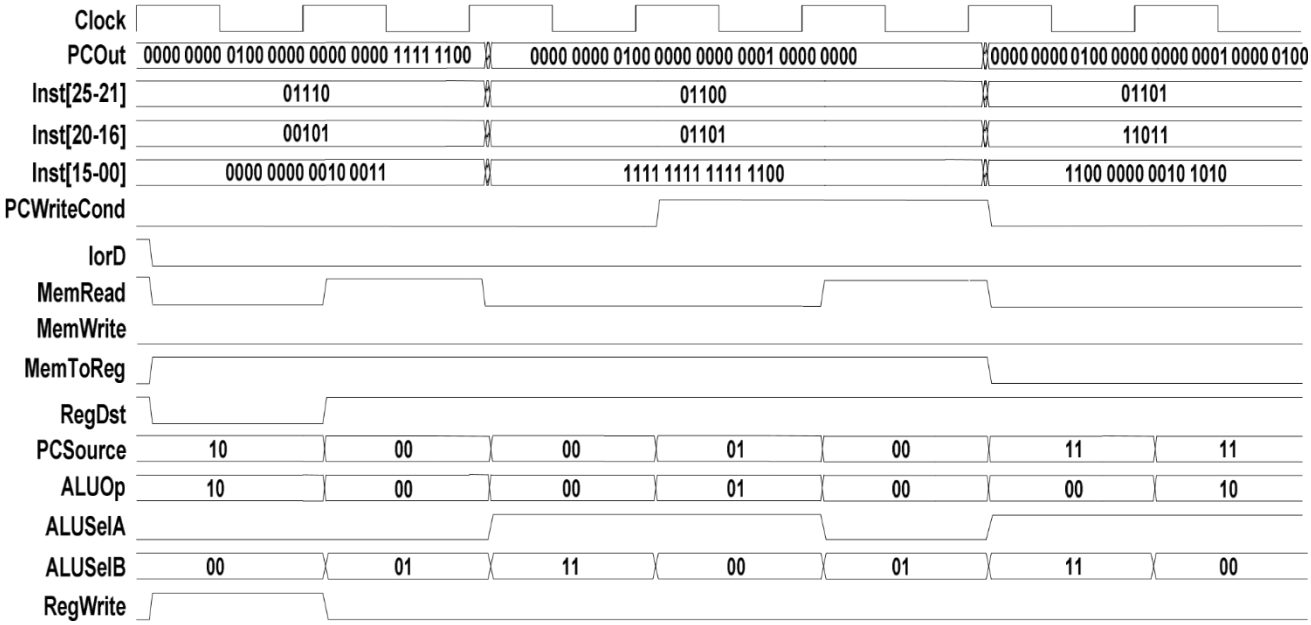
137. Repita o exercício anterior assumindo que o valor armazenado no endereço de memória **0x00000004** é **0x2C**. 53 ciclos para o SC e 215 ciclos para o MC

138. Descreva, sucintamente, as principais diferenças, ao nível estrutural, entre os *datapath single-cycle* e *multi-cycle*. O datapath SC tem duas memorias, enquanto que o MC so tem uma. O SC executa as instruções num ciclo de relógio apenas enquanto que o MC demora vários ciclos de relógio (dependendo da operação). A unidade de controlo do SC é combinatória e no MC é FSM, no MC a ALU realiza todos os cálculos enquanto que no SC existem somadores para calcular o PC+4

139. Indique, para o caso de um *datapath multi-cycle,* quais as operações realizadas pela ALU no decurso dos dois primeiros ciclos de relógio de qualquer instrução. No primeiro ciclo de relogio a ALU calcula o

PC+4 e no segundo calcula o BTA que advem do PC+4 do ciclo anterior mais o valor do campo imediato 16bits LSB extendidos para 32 e com um SLL2

140. Considere o diagrama temporal seguinte relativo à execução de uma sequência de três instruções, das quais apenas a segunda está completamente representada. Obtenha o código assembly desta sequência de três instruções.



No primeiro ciclo temos o final de uma lw, depois temos , no segundo ciclo temos uma instrução do tipo branch condicional e a ultima instruçãoé do tipo R

Label beq = 0xFFFC Funct = 101010 => slt

lw $14, 35($5)

beq $12, $14, 0xFFFC

slt $24 $13, $27

141. Considere a seguinte sequência de três instruções a serem executadas num *datapath muti-cycle*:

```
lw  $6,0($7)
and $8, $6, $5
beq $8, $0, L1
```

No diagrama temporal seguinte, relativo à execução desta sequência, identifique o nome dos sinais de controlo representados. (Note: o IorD não faz parte destes sinais)

Não há MemRead, MemWrite

1Linha - PCSrc

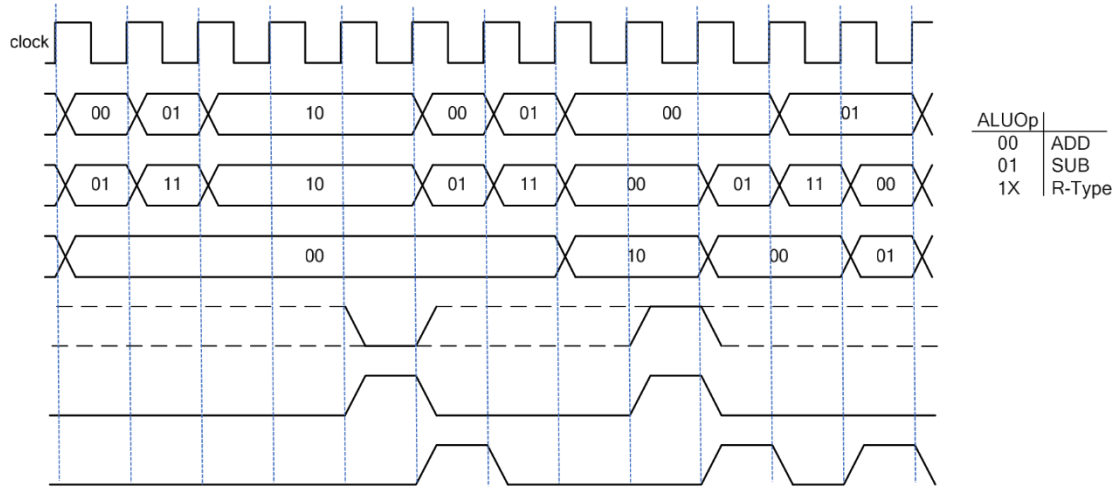
2Linha - ALUSelB

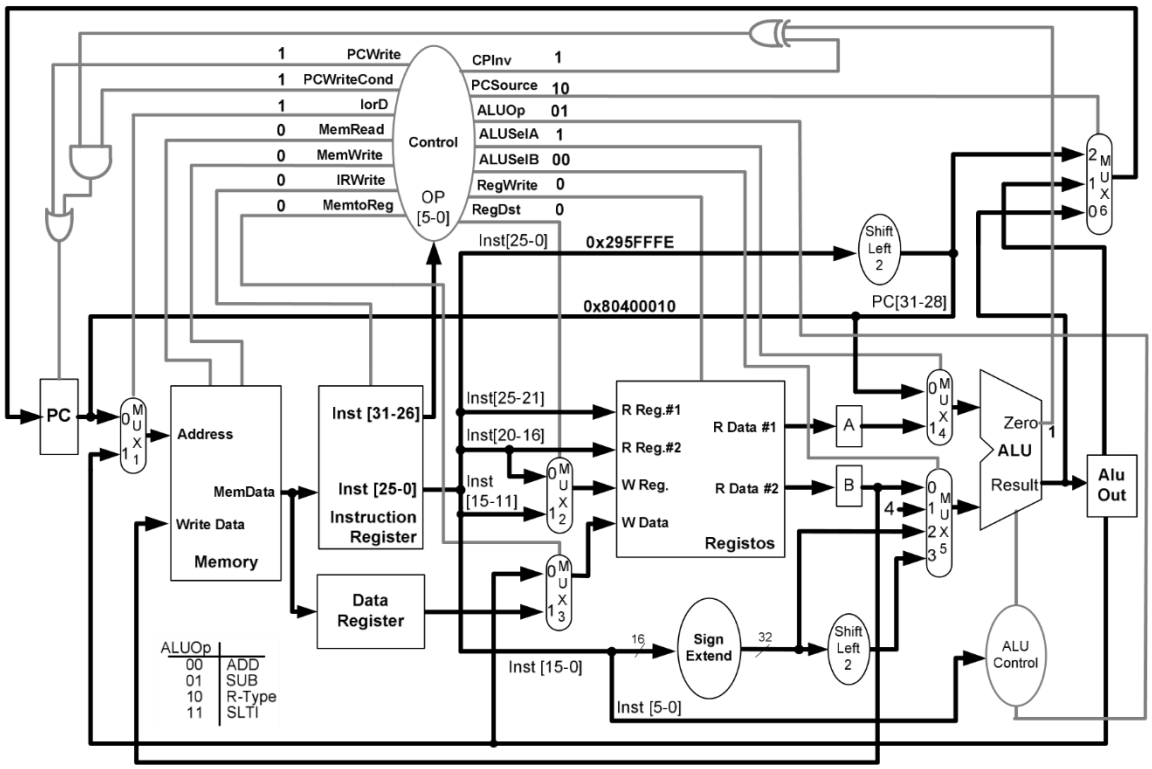
3Linha - ALUOp

4Linha - RegDst

5Linha - RegWrite

6Linha – PCWrite Condicional





142. Considere o *datapath multi-cycle* e a unidade de controlo fornecidos na figura acima. Admita que os valores indicados no *datapath* fornecido correspondem à "fotografia" tirada no decurso da execução de uma instrução armazenada no endereço **0x8040000C**. Tendo em conta todos os sinais, identifique, em *assembly*, a instrução que está em execução e a respetiva fase. Instrucao Jump
0x295FFFE = 10 1001 0101 1111 1111 1111 1110 (26bits)
1010 0101 0111 1111 1111 1111 1000 (28bits) c/ sll 2 (0xA97FFF8) com os 4 bits MSB de PC+4 (1000)
0x8

j label # label é 0x8A97FFF8 Na fase de Execute (fase final do jump)

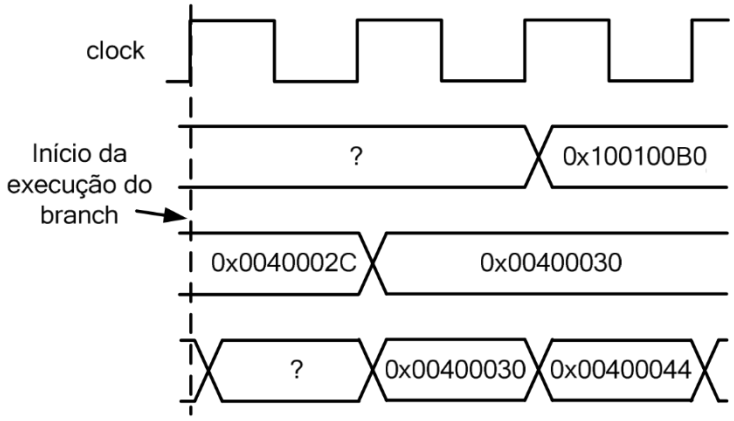
143. Considere a instrução **`beq $5 $6,L2`** armazenada no endereço **`0x0040002C`**. Admita que **`$5=0x1001009C`** e **`$6=0x100100B0`**. Identifique os registos representados na figura seguinte e obtenha o código máquina, em hexadecimal, da instrução indicada.

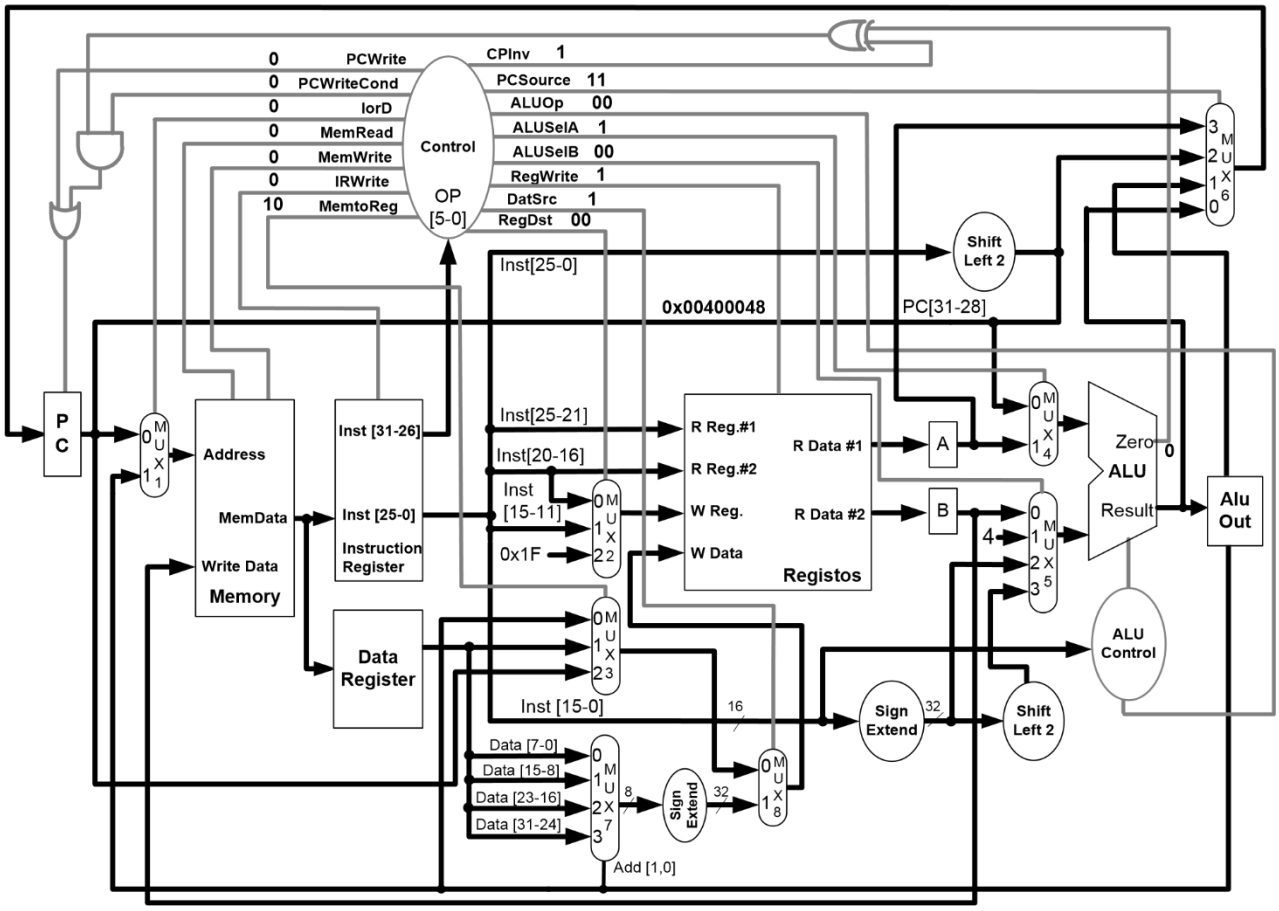
A primeira linha é o registo B

A segunda linha é o PC

A última linha é o ALUOut e o campo immediate é 5 (0x00400044 – 0x00400030) a dividir por 4

000100 00101 00110 0000000000000101 = 0x10A60005;





144. Considere o *datapath* e a unidade de controlo fornecidos na figura acima (com ligeiras alterações relativamente à versão das aulas teórico-práticas). Analise cuidadosamente as alterações introduzidas e identifique quais são as novas instruções que este *datapath* permite executar quando comparado com a versão fornecida nas aulas TP.
Conseguimos fazer um bne (vem do Xor com PCinv), instrução jr (vem do novo input do Mux com PCSrc), jal porque tens o mux do RegDst a escolher o registo 31 a ser escrito que vem do PC e o sb (que vem do novo mux que recebe a data de 8 em 8 bits)

145. Descreva, justificando, as principais características da unidade de controlo numa implementação pipelined da arquitetura MIPS, incluindo a sua natureza (combinatória ou síncrona) os sinais que constituem as variáveis independentes de entrada e as suas saídas.
Numa implementação pipelined a unidade de controlo é combinatória e oss sinais de controlo relevantes avançam no pipeline a cada ciclo de relógio (assim como os dados) estando, portanto, sincronizados com a instrução.
Os sinais que a constituem são:
PCWrite, PCWriteCond, ALUOp, RegDst, MemRead, MemWrite, RegWrite, PCSrc, Branch, Jump, MemToReg

146. Indique o que determina a máxima frequência de relógio de uma implementação *pipelined* da arquitetura MIPS com base nos principais elementos operativos que a constituem. A frequência maxima é definida pelo atraso de propagação do elemento operativo mais lento

147. Calcule, numa implementação *pipelined* da arquitetura MIPS em que a operação de *Write Back* é executada a meio do ciclo de relógio, a frequência máxima de operação, assumindo que os elementos operativos apresentam os seguintes atrasos de propagação:

    a. Memórias externas: Leitura: 10 ns, Escrita: – 8ns; File register: Leitura – 2ns, Escrita – 2ns; Unidade de Controlo: 2ns; ALU (qualquer operação): 6ns; Somadores: 4ns; Outros: 0ns.

    1/10ns = 100Mhz

    b. Memórias externas: Leitura: 5 ns, Escrita: – 7ns; File register: Leitura – 1ns, Escrita – 1ns; Unidade de Controlo: 1ns; ALU (qualquer operação): 8ns; Somadores: 1ns; Outros: 0ns.

    1/8ns = 125Mhz

    c. Memórias externas: Leitura: 8 ns, Escrita: – 10ns; File register: Leitura – 2ns, Escrita – 4ns; Unidade de Controlo: 2ns; ALU (qualquer operação): 6ns; Somadores: 2ns; Outros: 0ns.

    1/10 ns = 100Mhz

148. Identifique os principais tipos de *hazard* que podem existir numa implementação *pipelined* de um processador. Estrutural(+ do que uma instrução precisa de aceder ao mesmo h/w -> acontece quando existe apenas 1 memória(não acontece no MIPS) ou quando existem instruções com diferentes tempos de execução), De Controlo(Acontece nos Branches, quando é necessário fazer o IF de uma nova instrução e existe numa etapa mais avançada uma instrução que pode alterar o fluxo de execução e que

ainda não terminou(não sabe o address que fazer a seguir) e de Dados(existe uma dependência entre o resultado calculado e o operando usado por uma instrução seguinte, ambos guardados no mesmo registo)

149. Numa arquitetura *pipelined*, como se designa a técnica que permite utilizar como operando de uma instrução um resultado produzido por outra instrução que se encontra numa etapa mais avançada do mesmo. Forwarding

150. Explique por palavras suas em que circunstâncias pode ocorrer um *hazard* de dados numa implementação *pipelined* de um processador. Quando existe uma dependência entre o resultado calculado e o operando usado por uma instrução seguinte, ambos guardados no mesmo registo)

151. A existência de *hazards* de controlo pode ser resolvida por diferentes técnicas dependendo da arquitetura em causa. Identifique a técnica usada para o efeito numa arquitetura MIPS com *datapath pipelined*, como se designa essa técnica e em que consiste. Branch Delayed (consiste em correr sempre a instrução seguinte ao beq)

152. Em certas circunstâncias relacionadas com *hazards* de dados, não é possível resolver o problema sem recorrer a uma paragem parcial do *pipeline,* através do atraso de um ou mais ciclos de relógio no início da execução de uma instrução. Indique como se designa essa técnica e em que consiste ao nível do controlo do *pipeline* Stalling, consiste em inserir uma operação NOP(No operation que é meter todos os registos em ID/EX a 0)

153. Determine o número de ciclos de relógio que o trecho de código seguinte demora a executar num pipeline de 5 fases, desde o instante em que é feito o *Instruction Fetch* da 1ª instrução, até à conclusão da última.

```
add  $1,$2,$3
lw   $2,0($4)
sub  $3,$4,$3
addi $4,$4,4
and  $5,$1,$5     #"and" em ID, "add" já terminou
sw   $2,0($1)     #"sw" em ID, "add" e "lw" játerminaram
```

5 + (6-1) = 10 Ciclos de Relogio

154. Num datapath *single-cycle* o código da pergunta anterior demoraria 6 ciclos de relógio a executar. Por que razão é a execução no *datapath pipelined* mais rápida? A execução em Pipeline é mais rápida porque a frequência do relogio para pipelined depende apenas do componente que tem maior atraso enquanto que para SC a frequência depende do maior atraso para uma instrução (normalmente lw)

155. Quantos ciclos de relógio demora a execução do mesmo código num *datapath multi-cycle*?
25 Ciclos de relogio

156. Admita uma implementação *pipelined* da arquitetura MIPS com unidade de *forwarding* para EX e ID. Identifique, para as seguintes sequências de instruções, de onde e para onde deve ser executado o *forwarding* para que não seja necessário realizar qualquer *stall* ao pipeline:

a.

```
add $t0,$t1,$t2
lw $t1,0($t3)
beq $t3,$t0,LABEL
```

Há um forwarding the EX/MEM para ID

b.

```
sub $t0,$t1,$t2
addi $t3,$t0, 0x20
```

Há um forwarding the EX/MEM para EX

c.

```
lw $t0,0($t2)
sll $t2, $t2, 2
sw $t3,0($t0)
```

Há um forwarding the MEM/WB para EX

d.

```
lw $t3,0($t6)
xori $t0,$t4,0x20
sw $t3,($t0)
```

Há um forwarding the MEM/WB para EX

157. Descreva, por palavras suas, a função da unidade de *forwarding* de uma implementação *pipelined* da arquitetura MIPS. Permite enviar dados que ainda não foram escritos nos seus registos partes do pipeline que necessitem desse mesmo registo já atualizado

158. Admita o seguinte trecho de código, a executar sobre uma implementação *pipelined* da arquitetura MIPS com *delayed branches*, e unidade de *forwarding* de MEM e WB para o estágio EX.

```
LABEL: lw   $t3, 0($t4) # 1
sub  $t7, $t5, $t6 # 2
ori  $t2, $0, $0 # 3
beq  $t2, $0, LABEL # 4
add  $t4, $t7, $t7 # 5
```

a. Identifique os vários *hazards* neste código e determine se os mesmos podem ser resolvidos por *forwarding*. Tem hazard de dados em na linha 4.

b. Identifique as situações em que é necessário executar *stalling* do pipeline e o respetivo número de *stalls* É necessário fazer 1 stall depois de ORI

c. Resolva o problema anterior supondo que a arquitetura suporta *forwarding* de MEM para ID. Trocamos a ordem das instruções ori e sub, executando primeiro a instrução de ori e depois de sub, e usando um forwarding the EX/MEM para ID

159. Para o trecho de código seguinte identifique todas as situações de *hazard* de dados e de controlo que ocorrem na execução num pipeline de 5 fases, com *branches* resolvidos em ID.

```
main:
    lw      $1,0($0)
    add     $4,$0,$0
   lw       $2,4($0)
loop:
    lw      $3,0($1)
    add     $4,$4,$3
    sw      $4,36($1)
    addiu $1,$1,4
    sltu $5,$1,$2
    bne     $5,$0,loop
    sw      $4,8($0)
    lw  $1,12($0)
```

Memória de dados

Addr	Value
0x0000000	0x10
0x0000004	0x20

Tem hazard de Dados na 5ª instrução
Tem hazard de Dados na 6ª instrução
Tem hazard de Dados na 8ª instrução
Tem hazard de Dados na 9ª instrução
Tem hazard de Controlo na 9ª instrucao

160. Apresente o modo de resolução das situações de *hazard* de dados do código da questão 159, admitindo que o pipeline não implementa *forwarding*.
Para resolver o primeiro hazard fazemos um stall de 2 ciclos de relogio
Para resolver o segundo hazard fazemos um stall de 2 ciclos de relogio
Para resolver o terceiro hazard fazemos um stall de 2 ciclos de relogio
Para resolver o quarto hazard fazemos um stall de 2 ciclos de relogio
Para resolver o quinto hazard fazemos um stall de 1 ciclo de relogio

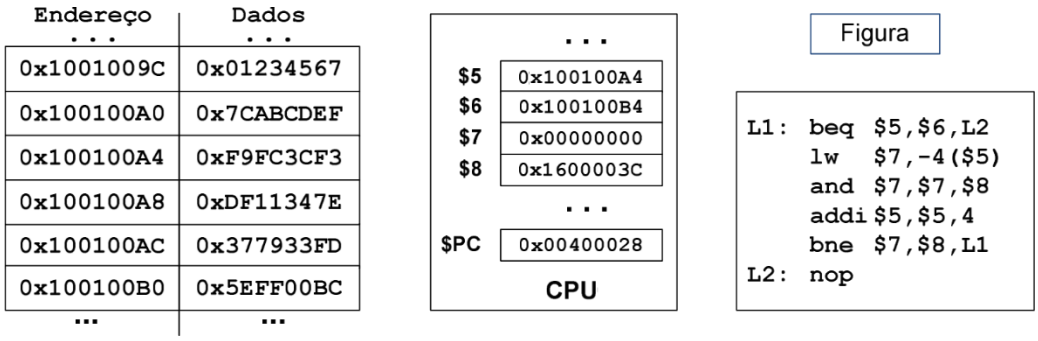
161. Calcule o número de ciclos de relógio que o programa anterior demora a executar num pipeline de 5 fases, sem *forwarding*, com *branches* resolvidos em ID e *delayed branch*, desde o IF da 1ª instrução até à conclusão da última instrução.
5+ 4 +5 +5 + (5 * (5+4+4+2+4+2+4+2 + 3 +4)) = 188 Ciclos

162. Resolva o problema anterior, considerando agora que o pipeline implementa *forwarding* para EX e para ID.
5 + 4 + 5 + 4 + 5 + (5 * (5+4+ 1 +4+4+4+ 1 + 3 + 3)) = 164 Ciclos

163. Calcule finalmente o número de ciclos de relógio que o programa do problema 159 demora a executar num pipeline de 5 fases, com *forwarding* para EX e para ID, com *branches* resolvidos em ID e *delayed branch*, desde o IF da 1ª instrução até à conclusão da última instrução.

5+4+5+5 + (5 * (5+4+4+4+4+ 1 + 3 +1))+ 4 = 153 Ciclos

164. Considere o trecho de código apresentado na figura seguinte, bem como as tabelas e os valores dos registos que aí se apresentam. Admita que o valor presente no registo **$PC** corresponde ao endereço da primeira instrução, que nesse instante o conteúdo dos registos é o indicado, e que vai iniciar-se o *instruction fetch* dessa instrução. Considere, para já, o *datapath* e a unidade de controlo fornecidos na pergunta 132 (Fig. 2), correspondentes a uma implementação *multi-cycle* simplificada da arquitetura MIPS. WTF



165. Determine o valor presente à saída do registo **ALUOut** durante a terceira fase de execução da segunda instrução (`lw $7,-4($5)`).
Em beq PC é 0x00400028
Em beq o PC+4 é 0x0040002C
Em lw o PC é 0x0040002C
Em lw o PC + 4 é 0x00400030

1111 1111 1111 1100 = -4 = 0xFFFC
Com SLL2 e SignExtend de 32 bits fica 0xFFFFFFC0

00400030
+FFFFFFC0
1|003FFFF0

ALUOut contem o valor calculado do BTA que é PC+4 mais o immediate. Fica 0x003FFFF0

166. Face aos valores presentes no segmento de dados (tabela da esquerda) e nos registos, calcule o número total de ciclos de relógio que demora a execução completa do trecho de código apresentado, numa implementação *multi-cycle* do MIPS (desde o instante inicial do *instruction fetch* da primeira instrução até ao momento em que vai iniciar-se o *instruction fetch* da instrução presente em "**L2:**").

FACIL

167. Suponha agora que o mesmo código é executado numa versão *pipelined* do *datapath* do MIPS semelhante à abordada nas aulas teórico-práticas de AC1. Admita que este *datapath* suporta apenas *forwarding* para EX. Determine o número total de ciclos de relógio que demora a execução completa do trecho de código apresentado, até ao instante inicial do *instruction fetch* da instrução imediatamente a seguir ao **nop**.

168. Continue a considerar a execução do código numa versão *pipelined* do *datapath* do MIPS. Admita que no instante zero, correspondente a uma transição ativa do sinal de relógio, vai iniciar-se o *instruction fetch* da primeira instrução. Determine o valor à saída da ALU na conclusão do sexto ciclo de relógio contando a partir do instante zero.

169. Repita as questões 165 a 168 para os dados da figura seguinte:

Endereço	Dados
...	...
0x10010028	0x31434120
0x1001002C	0x31303220
0x10010030	0x00000032
0x10010034	0xE0DE0AC1
0x10010038	0x0000FFFF
0x1001003C	0x00000000
...	...

...	
$6	0x00000000
$7	0x10010038
$8	0x10010028
$9	0xE0DE0AC1
...	
$PC	0x004000C4

CPU

Figura

```
lb    ,$
lw
addi  ,$
xor
addi  ,$
sw    ,$
bne
xor     ,$
nop
```

170. Considere a versão com *pipeline* do *datapath* apresentado na Fig. 3. Identifique todas as combinações de *forwarding* disponíveis neste *datapath* e, <u>para cada uma delas</u>, escreva uma curta sequência de instruções que desencadeie esse tipo específico de *forwarding*. Nos casos em que tal se aplique, identifique igualmente os casos em que é preciso gerar *stalling* e o número de ciclos de *stalling* necessários.

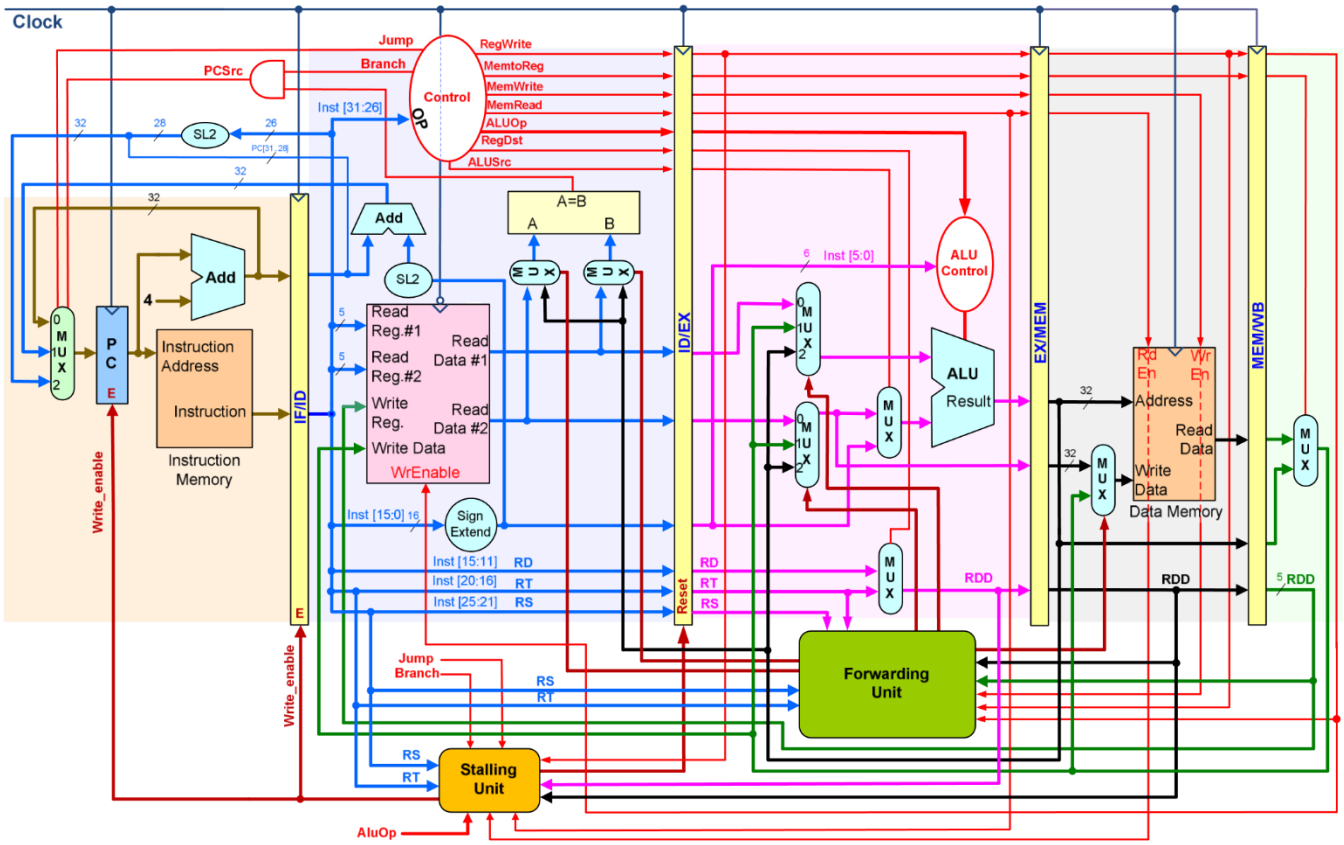


Fig. 3 – Piplined Datapath

Tabela de códigos de função (funct) e códigos de operação (OpCode) das principais instruções do MIPS

Arithm / Logical Instructions			Comparison Instructions	
Instruction	(funct)		Instruction	(OpCode)
add	100000 (0x20)		slti	001010 (0x0A)
addu	100001 (0x21)		sltiu	001001 (0x09)
and	100100 (0x24)			
div	011010 (0x1A)			
divu	011011 (0x1B)			
mult	011000 (0x18)		Branch Instructions	
multu	011001 (0x19)		beq	000100 (0x04)
nor	100111 (0x27)		bne	000101 (0x05)
or	100101 (0x25)		bgtz	000111 (0x07)
sll	000000 (0x00)		bgez	000001 (0x01)[1]
sra	000011 (0x03)		bltz	000001 (0x01)
srl	000010 (0x02)		blez	000110 (0x06)
sub	100010 (0x22)			
subu	100011 (0x23)		Jump Instructions	
xor	100110 (0x26)		j	000010 (0x02)
slt	101010 (0x2A)		jal	000011 (0x03)
sltu	101001 (0x29)		jalr	001001 (0x09)
			jr	001000 (0x08)
Arithm / Logical Imm				
Instruction	(OpCode)		Load/Store Instructions	
addi	001000 (0x08)		lb	100000 (0x20)
addiu	001001 (0x09)		lbu	100100 (0x24)
andi	001100 (0x0C)		lw	100011 (0x23)
ori	001101 (0x0D)		sb	101000 (0x28)
xori	001110 (0x0E)		sw	101011 (0x2B)
			Data Movement Instructions	
			mfhi	010000 (0x10)
			mflo	010010 (0x12)
			mthi	010001 (0x11)

			mtlo	**010011 (0x13)**