Prova P2 **Disciplina: Organização de Computadores**

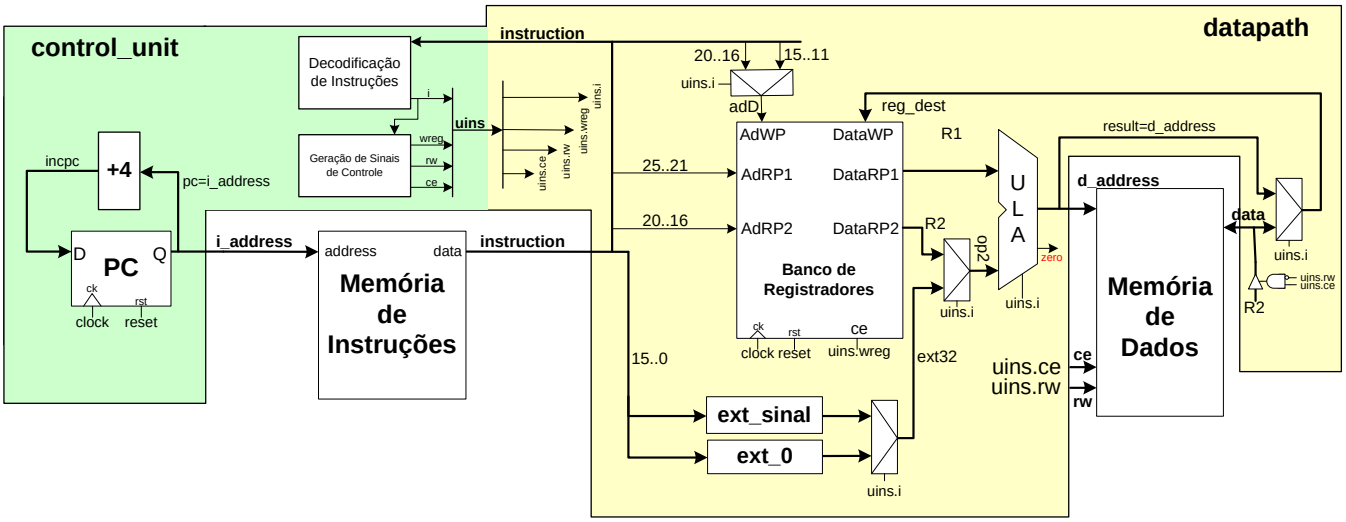
Professor: Ney Laert Vilar Calazans

Aluno:

18/novembro/2016

Para realizar a prova, refira-se à proposta de organização MIPS monociclo vista em aula. O diagrama de blocos deste processador aparece abaixo, e detalha o Bloco de Dados (**datapath**) e o Bloco de Controle (**control_unit**). Assuma que as instruções às quais esta organização monociclo dá suporte de execução são apenas as seguintes (exceto se a questão particular especificar de outra forma): **ADDU, SUBU, AND, OR, XOR, NOR, LW, SW e ORI**.

MIPS_V0



1. [3,0 pontos] Considere o processador **MIPS monociclo** apresentado acima. Observe os trechos de VHDL abaixo, copiados da descrição deste processador:

```
I.    uins.ce    <= '1' when i=SW  or i=LW else '0';
II.   uins.rw    <= '0' when i=SW  else '1';
III.  uins.wreg  <= '0' when i=SW  else '1';
```

   a) Suponha que existe uma falha no hardware que produz a linha II. acima. A falha é que o sinal **`uins.rw`** está eternamente grudado em 0. A pergunta aqui é se esta falha modificará o comportamento de uma ou mais instruções. Se sim, diga qual(is) instrução(ões) é(são) afetada(s), justificando sua resposta. Se não, justifique porque a falha não afeta a operação do hardware.

   b) Observe na documentação do MIPS a funcionalidade da instrução SLTIU e diga como o hardware acima deveria ser alterado para ser capaz de executar esta instrução. Especifique os módulos do código VHDL que precisariam ser alterados e diga como estes devem ser alterados. Diga também se algum módulo de hardware adicional é necessário e porquê. Se achar necessário, ilustre as alterações com desenhos esclarecedores.

2. [3,0 pontos] Observe a linha de código VHDL abaixo, parte da descrição do processador monociclo MIPS_V0.

```
reg_dest <=  data when uins.i=LW else result;
```

   a) (1,5 pontos) Desenhe um hardware correto que seria gerado por uma ferramenta de síntese ao processar esta linha, assumindo o seguinte: uins.i é um sinal que codifica com o mínimo número de bits possível os símbolos associados ao nome de cada instrução do processador e o símbolo **`invalid_instruction`**. A codificação parte do menor valor possível, e segue em ordem crescente para os símbolos ordenados conforme aparece na declaração do tipo inst_type (ou seja, **`ADDU, SUBU, AAND, OOR, XXOR, NNOR, LW, SW, ORI, invalid_instruction`**).

   b) (1,5 pontos) Quer-se modificar o processador MIPS_V0 para lhe dar capacidade de executar as instruções LBU e LHU do MIPS. Estude a funcionalidade destas duas instruções no Apêndice A, e diga como a descrição VHDL do hardware associado ao sinal **`reg_dest`** teria de ser modificada. O objetivo, claro, é que no sinal **`reg_dest`** a execução destas instruções gere o dado correto a ser armazenado no banco de registradores.
   Dica 1: A interface com a memória de dados não deve ser alterada, ou seja, uma leitura da memória sempre traz para dentro do processador o valor de 32 bits armazenado a partir do endereço gerado na saída da ULA.

Dica 2: O formato do resultado é o seguinte:

```
reg_dest <=   data when uins.i=LW       else
              X when uins.i=LHU   else -- diga que código VHDL é X
              Y when uins.i=LBU   else -- diga que código VHDL é Y
              result;
```

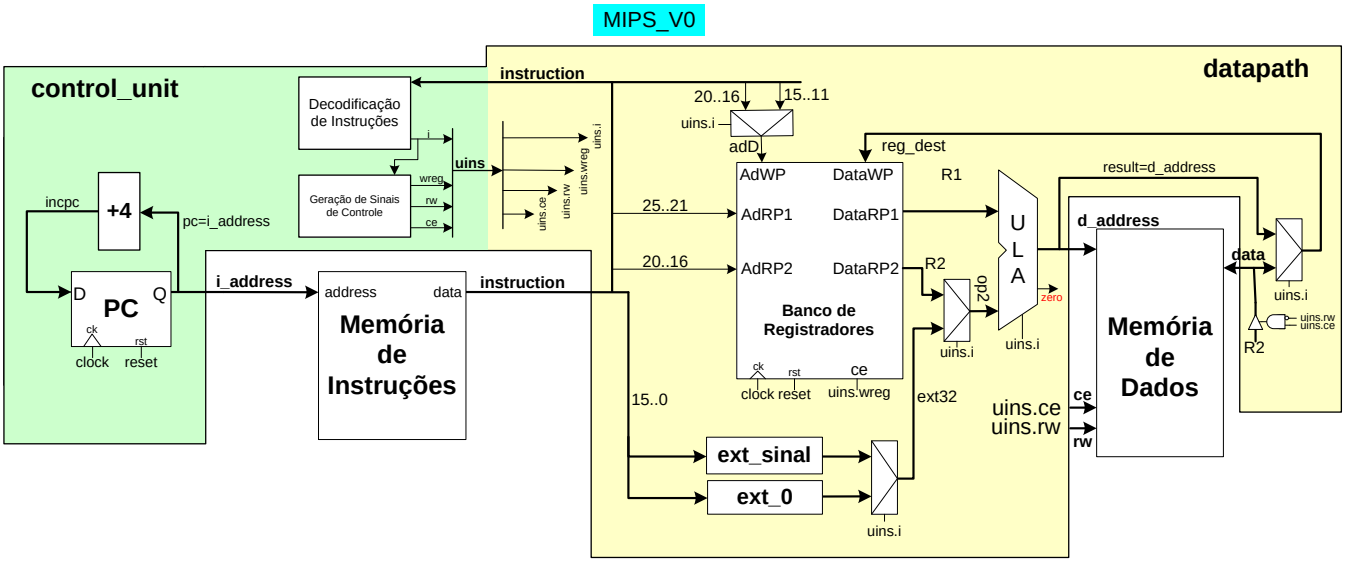
3. [4,0 pontos] Dada uma frequência de operação de 600 MHz para o processador **MIPS monociclo**, assuma que a organização original foi alterada para dar suporte a todas as instruções do programa abaixo, mantendo a característica monociclo. Com estes pressupostos, calcule e diga:
   a) (1,5 pontos) O número de ciclos de relógio que leva p/ executar o programa, com a área de dados fornecida;
   b) (1 ponto) O tempo de execução do programa em segundos (1ns=10^{-9} segundos);
   c) (1 ponto) O que faz este programa, do ponto de vista semântico. Diga o que ele gera como saída;
   d) (0,5 pontos) Se este programa possui subrotina(s). Se sim, diga onde esta(s) se encontra(m), definindo o intervalo de linhas que ela(s) ocupa(m).

Dica: A chamada do sistema número 11, usada neste programa, executa a impressão de exatamente um caractere ASCII na tela, aquele caractere cujo código está contido nos 8 bits menos significativos do registrador $a0 no momento da execução do syscall.

```
1.          .data
2.  n:      .word  0xab83efde
3.
4.          .text
5.  main:   li     $a0,'0'
6.          li     $v0,11
7.          syscall
8.          li     $a0,'x'
9.          li     $v0,11
10.         syscall
11.         li     $t0,8
12.         la     $t1,n
13.         lw     $a1,0($t1)
14. c:      jal    imsh
15.         sll    $a1,$a1,4
16.         addiu  $t0,$t0,-1
17.         beq    $t0,$zero,ed
18.         j      c
19. ed:     li     $v0,10
20.         syscall
21. imsh:   srl    $t2,$a1,28
22.         addiu  $t3,$t2,-10
23.         bltz   $t3,no
24.         ori    $t2,$t2,0x40
25.         addiu  $t2,$t2,-9
26.         j      pr
27. no:     ori    $t2,$t2,0x30
28. pr:     move   $a0,$t2
29.         li     $v0,11
30.         syscall
31.         jr     $ra
```

Gabarito

Para realizar a prova, refira-se à proposta de organização MIPS monociclo vista em aula. O desenho da versão monociclo aparece abaixo, com detalhamento do Bloco de Dados. Assuma que as instruções às quais esta organização monociclo dá suporte de execução são apenas as seguintes (exceto quando a questão particular especificar de outra forma): **ADDU, SUBU, AND, OR, XOR, NOR, LW, SW e ORI**.



1. [3,0 pontos] Considere o processador **MIPS monociclo** apresentado acima. Observe os trechos de VHDL abaixo, copiados da descrição deste processador:

```
   I.    uins.ce    <= '1' when i=SW  or i=LW else '0';
  II.    uins.rw    <= '0' when i=SW  else '1';
 III.    uins.wreg  <= '0' when i=SW  else '1';
```

   a) Suponha que existe uma falha no hardware que produz a linha II. acima. A falha é que o sinal **`uins.rw`** está eternamente grudado em 0. A pergunta aqui é se esta falha modificará o comportamento de uma ou mais instruções. Se sim, diga qual(is) instrução(ões) é(são) afetada(s), justificando sua resposta. Se não, justifique porque a falha não afeta a operação do hardware.

   b) Observe na documentação do MIPS a funcionalidade da instrução SLTIU e diga como o hardware acima deveria ser alterado para ser capaz de executar esta instrução. Especifique os módulos do código VHDL que precisariam ser alterados e diga como estes devem ser alterados. Diga também se algum módulo de hardware adicional é necessário e porquê. Se achar necessário, ilustre as alterações com desenhos esclarecedores.

Solução:

a) A análise desta falha leva à conclusão de que a instrução SW continuará funcionando sem alteração, pois a falha fixa uins.rw no valor que faz com que SW execute. Quanto às instruções que não operam com a memória (todas as demais, exceto LW) estas devem continuar operando normalmente, pois para elas **uins.ce** será igual a 0, fazendo com que o valor de uins.rw seja irrelevante. Já a instrução LW irá sofrer alteração na sua operação. Ela ativa o sinal ce (**uins.ce**=1), mas **uins.rw** grudado em 0 forca uma escrita na memória. Ou seja, a ULA calcula o endereço de leitura corretamente, mas neste endereço será feita uma escrita e não uma leitura. O valor escrito (erradamente) na posição certa de memória é o que há no sinal **data**, que será o conteúdo do registador que deveria ser sobre-escrito com o dado vindo da memória (que aparece no sinal **R2**). Ao mesmo tempo este registrador vai ser escrito com seu próprio valor, que chegará no sinal **reg_dest** via o mux mais à direita no desenho. Em resumo, a falha afeta apenas a operação da instrução LW.

b) A instrução **SLTIU** possui a seguinte sintaxe em linguagem de montagem: **SLTIU $Rt, $Rs, imm**. Sua semântica consiste em escrever a constante 0 ou a constante 1 (representada em 32 bits) no registrador $Rt, dependendo do resultado do teste $Rs < ext_sinal(imm)? Este teste considera o conteúdo de $Rs e ext_sinal(imm) como números sem sinal. Se $Rs for menor que ext_sinal(imm), $Rt recebe 1, senão recebe 0. Existem várias formas de realizar esta funcionalidade, dá-se a seguir um dos mais simples conjuntos de alterações na MIPS_V0 que faz com que o processador possa executar a instrução SLTIU. A idéia é que trata-se de uma instrução tipo I. Assim, cria-se uma nova operação na

ULA (SLTIU) que gera na saída desta ULA ou a constante 0 ou a constante 1, depois de testar se a entrada superior da ULA é menor que a entrada inferior. As alterações são:

- Alterações no Package – acrescentar o valor SLTIU ao conjunto de valores que o tipo inst_type pode assumir, ou seja:

```
type inst_type is (ADDU, SUBU, AAND, OOR, XXOR, NNOR, LW, SW, ORI, SLTIU, invalid_instruction);
```

- Alterações no Bloco de Controle – acrescentar o hardware de decodificação da SLTU, ou seja:

```
i <= ADDU   when instruction(31 downto 26)="000000" and instruction(10 downto 0)="00000100001" else
     SUBU   when instruction(31 downto 26)="000000" and instruction(10 downto 0)="00000100011" else
     AAND   when instruction(31 downto 26)="000000" and instruction(10 downto 0)="00000100100" else
     OOR    when instruction(31 downto 26)="000000" and instruction(10 downto 0)="00000100101" else
     XXOR   when instruction(31 downto 26)="000000" and instruction(10 downto 0)="00000100110" else
     NNOR   when instruction(31 downto 26)="000000" and instruction(10 downto 0)="00000100111" else
   SLTIU when instruction(31 downto 26)="001011" else
     ORI    when instruction(31 downto 26)="001101" else
     LW     when instruction(31 downto 26)="100011" else
     SW     when instruction(31 downto 26)="101011" else
     invalid_instruction ;       -- IMPORTANTE: condicao "default" eh invalid instruction;
```

- Alterações no Bloco de Dados

Os multiplexadores que geram o sinal adD, op2 e reg_dest e o controle tristate do sinal data não precisam ser alterados, pois SLTIU usa seus valores *default* e não faz acesso à memória de dados. No entanto, a extensão de sinal não é o valor *default* de saída do mux que corresponde ao sinal ext32. Assim, é necessário alterar seu código, que fica:

```
ext32 <= x"FFFF" & instruction(15 downto 0) when (instruction(15)='1'
        and (uins.i=LW or uins.i=SW or uins.i=SLTIU)) else
              x"0000" & instruction(15 downto 0);
```

- Alterações na ULA - definir instrução SLTIU com a funcionalidade necessária, por exemplo (note que isto funcionará se e somente se na entidade da ULA houver a inclusão do pacote IEEE.std_logic_unsigned.all, o que já é o caso na ULA da MIPS_V0):

```
int_alu <=
    op1 - op2      when  op_alu=SUBU                  else
    op1 and op2    when  op_alu=AAND                  else
    op1 or  op2    when  op_alu=OOR  or op_alu=ORI    else
    op1 xor op2    when  op_alu=XXOR                  else
    op1 nor op2    when  op_alu=NNOR                  else
   x"00000001"   when op_alu=SLTIU and op1 < op2 else
   x"00000000"   when op_alu=SLTIU               else
    op1 + op2;      --- default eh a soma
```

2. [3,0 pontos] Observe a linha de código VHDL abaixo, parte da descrição do processador monociclo MIPS_V0.

```
reg_dest <=  data when uins.i=LW else result;
```

a) (1,5 pontos) Desenhe um hardware correto que seria gerado por uma ferramenta de síntese ao processar esta linha, assumindo o seguinte: uins.i é um sinal que codifica com o mínimo número de bits possível os símbolos associados ao nome de cada instrução do processador e o símbolo **`invalid_instruction`**. A codificação parte do menor valor possível, e segue em ordem crescente para os símbolos ordenados conforme aparece na declaração do tipo inst_type (ou seja, **`ADDU, SUBU, AAND, OOR, XXOR, NNOR, LW, SW, ORI, invalid_instruction`**).

b) (1,5 pontos) Quer-se modificar o processador MIPS_V0 para lhe dar capacidade de executar as instruções LBU e LHU do MIPS. Estude a funcionalidade destas duas instruções no Apêndice A, e diga como a descrição VHDL do hardware associado ao sinal **`reg_dest`** teria de ser modificada. O objetivo, claro, é que no sinal **`reg_dest`** a execução destas instruções gere o dado correto a ser armazenado no banco de registradores.
Dica 1: A interface com a memória de dados não deve ser alterada, ou seja, uma leitura da memória sempre traz para dentro do processador o valor de 32 bits armazenado a partir do endereço gerado na saída da ULA.
Dica 2: O formato do resultado é o seguinte:

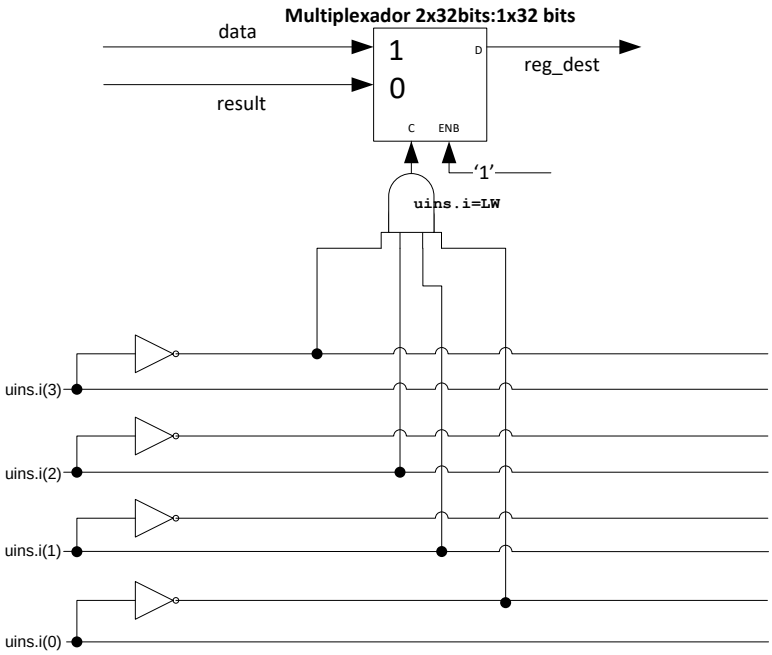
```
reg_dest <=   data when uins.i=LW       else
              X when uins.i=LHU   else -- diga que código VHDL é X
              Y when uins.i=LBU   else -- diga que código VHDL é Y
              result;
```

Solução:

a) Como se têm 10 valores possíveis para o tipo inst_type, do qual uins.i é uma instância, é necessário usar um mínimo de 4 bits para codificar estes. Com isto, a codificação em ordem crescente de códigos para os valores só pode ser a dada abaixo:

ADDU-0000, SUBU-0001, AAND-0010, OOR-0011, XXOR-0100, NNOR-0101, LW-0110, SW-0111, ORI-1000, invalid_instruction-1001

A solução para o hardware é que reg_dest será a saída de um multiplexador (mux) 2:1 com duas entradas de 32 bits e saída de 32 bits. Assumindo que a condição verdadeira corresponde à entrada 1 do mux e que a condição falsa corresponde a entrada 0 do mux, o controle resume-se a apenas um detector de LW implementável com uma porta AND de 4 entradas, conforme ilustra a Figura abaixo:



b) As instruções LBU e LHU funcionam de forma similar à instrução LW. Ambas calculam o endereço de leitura exatamente da mesma forma que LW (via a soma do endereço base em Rs e do deslocamento com sinal estendido) e escrevem um valor de 32 bits no registrador destino (Rt). Contudo, ao invés de ler dados de 32 bits da memória, leem respectivamente 8 e 16 bits apenas. Como o valor escrito deve ser de 32 bits, estas instruções preveem que se complete os 32 bits com zeros suficientes à esquerda do valor vindo da memória. Como o pressuposto do hardware estabelecido na questão é que sempre se leem 32 bits da memória, os bits adicionais lidos devem ser descartados. Dada esta discussão, os valores de X e Y são dados abaixo:

```
reg_dest <=    data          when uins.i=LW      else
               x"0000" & data(15 downto 0)
                     when uins.i=LHU     else -- X = 16 zeros
               x"000000" & data(7 downto 0)
                     when uins.i=LBU     else -- Y = 24 zeros
               result;
```

3. [4,0 pontos] Dada uma frequência de operação de 600 MHz para o processador **MIPS monociclo**, assuma que a organização original foi alterada para dar suporte a todas as instruções do programa abaixo, mantendo a característica monociclo. Com estes pressupostos, calcule e diga:
   a) (1,5 pontos) O número de ciclos de relógio que leva p/ executar o programa, com a área de dados fornecida;
   b) (1 ponto) O tempo de execução do programa em segundos (1ns=10^{-9} segundos);
   c) (1 ponto) O que faz este programa, do ponto de vista semântico. Diga o que ele gera como saída;
   d) (0,5 pontos) Se este programa possui subrotina(s). Se sim, diga onde esta(s) se encontra(m), definindo o intervalo de linhas que ela(s) ocupa(m).

Dica: A chamada do sistema número 11, usada neste programa, executa a impressão de exatamente um caractere ASCII na tela, aquele caractere cujo código está contido nos 8 bits menos significativos do registrador $a0 no momento da execução do syscall.

```
1.          .data
```

```
2.  n:     .word  0xab83efde
3.
4.         .text
5.  main:  li     $a0,'0'       # (1) $a0 contém o caracter ASCIIE '0'
6.         li     $v0,11        # (1) prepara chamada de print char
7.         syscall              # (1) chama print char
8.         li     $a0,'x'       # (1) $a0 contém o caracter ASCIIE 'x'
9.         li     $v0,11        # (1) prepara chamada de print char
10.        syscall              # (1) chama print char
11.        li     $t0,8         # (1) $t0 recebe contador de nibbles
12.        la     $t1,n         # (2)
13.        lw     $a1,0($t1)    # (1) lê n para $a1, parâmetro a passar
14. c:     jal    imsh          # (1) chama sub-r que imprime valor hexa mais significativo
15.        sll    $a1,$a1,4     # (1) gera novo nibble mais significativo
16.        addiu  $t0,$t0,-1    # (1) decrementa contador de nibbles
17.        beq    $t0,$zero,ed  # (1) sai do laço se chegou ao fim
18.        j      c             # (1) senão repete o laço
19. ed:    li     $v0,10        # (1) Depois do laço, cai fora
20.        syscall              # (1) do programa
21. imsh:  srl    $t2,$a1,28    # (1) Primeira linha da subrotina, coloca nibble no fim
22.        addiu  $t3,$t2,-10   # (1) subtrai 10 para ver se é dígito ou letra em hexa
23.        bltz   $t3,no        # (1) salta se for número
24.        ori    $t2,$t2,0x40  # (1) Senão é letra entre A e F, duas linhas
25.        addiu  $t2,$t2,-9    # (1) que geram o ASCIIE da letra em $t2
26.        j      pr            # (1) e pula para imprimir
27. no:    ori    $t2,$t2,0x30  # (1) no caso de número, uma linha que gera seu ASCIIE
28. pr:    move   $a0,$t2       # (1) $a0 contém o caracter ASCIIE a imprimir (letra/dígito)
29.        li     $v0,11        # (1) prepara chamada de print char
30.        syscall              # (1) chama print char
31.        jr     $ra           # (1) e retorna para quem chamou.
```

Solução:

a) O programa principal tem uma parte com instruções que só executam uma vez (linhas 5-13 e linhas 19-20). Estas linhas tomam um total de 12 ciclos. Além destas, existe um laço entre as linhas 14-18, executado exatamente 8 vezes, uma para cada nibble (4 bits) da palavra n de 32 bits. Cada execução do laço, com exceção da última, passa por todas suas linhas, gastando 5 ciclos, o que corresponde a 5*7=35 ciclos. Na última passagem pelo laço, executam-se apenas as linhas 14-17, gastando-se 4 ciclos. Assim, o programa principal consome um total de 12+35+4=51 ciclos para executar. A subrotina imsh é chamada exatamente 8 vezes e ocupa as linhas 21-31 do código. Sua estrutura não possui laços, executando sequencialmente, exceto por um código equivalente a um if-then-else. Existem duas possibilidades de execução, dependendo do valor do nibble mais significativo da palavra passada como parâmetro para a subrotina em $a1: (1-then) Se o nibble é um valor entre 0000 e 1001, as linhas não são executadas e a rotina completa sua função em 8 ciclos; (2-else) Se o nibble é um valor entre 1010 e 1111, apenas a linha 27 não é executada e a rotina completa sua função em 10 ciclos. Com o valor de n fornecido, o caso then acontece 2 vezes e o caso else 6 vezes. Logo, gasta-se executando a subrotina 2*8+6*10=76 ciclos. Consequentemente, o programa como um todo executa então em 51+76= 127 ciclos de relógio.

b) Como a frequência é de 600MHz, o período (ou o tempo de um ciclo) é (1/(600*10^6)) segundos, ou ~1,67 x10^{-9}s. Logo, o tempo de execução do programa será de

$$127*1,67\ x10^{-9}s = \sim 2,12x10^{-7}s.$$

c) O programa imprime o valor de 32 bits armazenado em **n** como um número hexadecimal, no formato **0X**zzzzzzzz, onde os primeiros dois caracteres indicam uso da base 16 e os 8 caracteres z são os valores de cada um dos nibbles de **n** expressos como números hexadecimais (0-9 e A-F).

d) Sim, como mencionado nos itens anteriores, o programa possui uma subrotina, **imsh**, que ocupa as linhas 21-31 do código fornecido.