Universidade de Brasília – 2022/1
Departamento de Ciência da Computação – Prof. Marcus Vinicius Lamar
CIC0099 – Organização e Arquitetura de Computadores – Turma 03

$d_0\ d_1$ / $d_2\ d_3\ d_4\ d_5\ d_6\ d_7\ d_8$

Nome: Monitores do 2022/2

Matrícula: 1 9 / 0 0 3 8 6 4 1

Prova 1

→ Gabarito comentado das questões 1-3 e 4)b) ao final do pdf

(4.0) 1) Dado o programa C ao lado,
(2.0) a) Respeitando a convenção do uso de registradores, compile eficientemente para Assembly RISC-V ISA RV32IMF as rotinas `main` e `SOMAX()`.

Dica: comente o código!

```
void main () {
  int A[9]={d0,d1,d2,d3,d4,d5,d6,d7,d8};
  printf("soma=%d", SOMAX(A,9));
}

int SOMAX(int v[], int n) {
  int i, soma=0;
  for (i=0; i<n; i++)
    if (v[i]%2 == 0)
      soma = soma + v[i];
  return soma;
}
```

(2.0) b) Para um processador RISC-V, onde as instruções de acesso à memória (loads e stores) e os desvios (condicionais e incondicionais) possuem CPI=2 e todas as outras instruções possuem CPI=1, com frequência de *clock* de 1 GHz, responda:

(1.0) b.1) Qual a CPI média obtida para o seu procedimento `SOMAX`? $\overline{\text{CPI}} = 15/11 = 1,\overline{36}$

(1.0) b.2) Qual o tempo necessário à execução do seu procedimento `SOMAX`? $t_{\text{exec}} = 91$ ns

(2.0) 2) Considere o mapa de memória de código abaixo.
(1.4) a) Escreva o programa em Assembly abaixo

Endereço	Conteúdo	Instrução
0x00400000	0x00b50c63	beq a0, a1, 24
0x00400004	0x00b55663	bge a0, a1, 12
0x00400008	0x40a585b3	sub a1, a1, a0
0x0040000C	0xff5ff06f	jal zero, -12
0x00400010	0x40b50533	sub a0, a0, a1
0x00400014	0xfedff06f	jal zero, -20
0x00400018	0x00008067	jalr zero, ra, 0

(0.6) b) Se `a0=(`$d_7 d_8$`)+1` e `a1=(`$d_5 d_6$`+1)`, qual deve ser a frequência de *clock* de um Processador RISC-V Uniciclo (CPI=1) para que o procedimento seja executado em $(d_4 + 1)$ μs? $f_{\text{clk}} = 13,5$ MHz

(3.0) 3) A ISA RV32I não possui as instruções de multiplicação, divisão e módulo (resto da divisão). No entanto, isto não significa que uma linguagem de alto nível não possa usar essas operações matemáticas neste processador. Implemente eficientemente a instrução abaixo como um **procedimento MUL** que não modifica **nenhum** registrador além do registrador de destino.

`mul a0, a0, a1 # a0=Low(a0×a1) sendo a0 e a1 números com sinal`

(2.0) 4) As operações em ponto flutuante foram um marco na computação científica, permitindo que números reais em uma ampla faixa dinâmica fossem representados em um número finito de bits. Dada a representação em ponto flutuante **IEEE 754 precisão simples:**

(1.0) a) Quando comparada com a representação de números inteiros de 32 bits em complemento de 2, qual das duas representações possibilita representar a maior quantidade de números diferentes? Por quê?

A de complemento, por não se preocupar com NaN, $\pm\infty$, etc.

(0.5) b) Dado o número `0x40d4d50000` em IEEE754 precisão simples, qual o número final correspondente?

`0x40380000` = 2.875_{10}

(0.5) c) Dada a aproximação do número $\pi = 3.14159265358979323846264338327950288419716939937510$, até que casa decimal conseguimos representá-lo com IEEE754 precisão simples?

π tem expoente 1, de forma que ao converter a mantissa o menor expoente será $2^{-22} = 0,0000002\ldots$
Teremos então representação correta até a 6ª casa decimal.

Gabarito

1) a) A convenção pede que apenas os registradores temporários sejam modificados numa rotina, com a possível exceção de registradores de saída. Aqui evitou-se alterar `a1`, mas `a0` foi modificado à vontade.

```
    .data
      str:  .string "soma="
      A:    .word 1, 9, 0, 0, 3, 8, 6, 4, 1
    .text
      main: li  a7, 4   # syscall PrintString
            la  a0, str # a0 = "soma="
            ecall       # print("soma=")
            li  a1, 9   # a1 = 9
            la  a0, A   # a0 = A
            jal SOMAX   # chama a funcao, resultado em a0
            li  a7, 1   # syscall PrintInt
            ecall       # print(a0)
            li  a7, 10  # syscall Exit
            ecall       # fim do programa

      SOMAX:  li    t0, 0           # soma = 0
              li    t1, 0           # i = 0
      for:    lw    t2, 0(a0)       # t2 = v[i]
              andi  t3, t2, 1       # t3 = v[i]%2
              bne   t3, zero, impar # impar ? nao soma : soma
              add   t0, t0, t2      # soma += v[i]
      impar:  addi  t1, t1, 1       # i++
              addi  a0, a0, 4       # endereco do proximo valor no vetor
              blt   t1, a1, for     # i < n ? continua o laco : retorna
              mv    a0, t0          # a0 <- soma
              ret                   # retorno
```

b.1) O procedimento `SOMAX` como escrito acima possui 11 instruções, sendo 2 delas pulos condicionais (`bne`, `blt`), 1 delas um pulo incondicional (`ret`) e 1 delas um load. Logo, a CPI média nesta rotina é

$$\overline{\text{CPI}} = \frac{7}{11} \cdot 1 + \frac{4}{11} \cdot 2 = \frac{15}{11}.$$

b.2) Dada a frequência de *clock* $f = 10^9$ Hz, precisamos encontrar a quantidade de ciclos executados para calcular o t_{exec}. Da forma como foi escrito, dentro do laço *for*, o `SOMAX` executa 7 instruções por valor par no vetor, e 6 instruções por ímpar. Além disso, dentro do laço *for* são executadas 2 instruções de desvios condicionais e uma de acesso a memória, independente da paridade do valor do vetor. Fora desse laço, são executadas mais 4 instruções, 2 antes e 2 depois.
Assim, para `A[9] = {1,9,0,0,3,8,6,4,1}`, temos

$$\text{I} = (4 \cdot 6) + (5 \cdot 7) + 4 = 63.$$

Como o laço *for* é executado nove vezes, temos $3 \cdot 9 = 27$ instruções de dois ciclos executadas por ele e uma instrução de dois ciclos executada fora do laço (`ret`). Então 28 instruções possuem CPI=2 e as $63 - 28 = 35$ instruções restantes possuem CPI=1. Logo a quantidade C de ciclos executados é

$$C = 28 \cdot 2 + 35 \cdot 1 = 91.$$

Assim, o tempo de execução dessa rotina é:

$$t_{\text{exec}} = C \times f = 91 \times 10^{-9} s = 91 \text{ ns}.$$

2) a) Questões de montagem/tradução de código de máquina são laboriosas mas diretas. A ideia aqui é converter o hexadecimal para binário, e com o OpCode determinar qual o tipo da instrução. Sabendo o tipo, é imediato obter os registradores da instrução e etc.
Padrão de cores: OpCode, Rd, Rs1, Rs2, funct3, funct7, Imm.

1. 0x00b50c63 = 0b 0000000 01011 01010 000 11000 $\underbrace{1100011}_{\text{tipo B}}$

   = `beq x10, x11, 0b000000011000`
   = `beq a0, a1, 24`

2. 0x00b55663 = 0b 0000000 01011 01010 101 01100 $\underbrace{1100011}_{\text{tipo B}}$

   = `bge x10, x11, 0b000000001100`
   = `bge a0, a1, 12`

3. 0x40a585b3 = 0b 0100000 01010 01011 000 01011 $\underbrace{0110011}_{\text{tipo R}}$

   = `sub x11, x11, x10`
   = `sub a1, a1, a0`

4. 0xff5ff06f = 0b 1111 1111 0101 1111 1111 00000 $\underbrace{1101111}_{\text{tipo J}}$

   = `jal x0, 0b11111111111111110100`
   = `jal zero, -12`

5. 0x40b50533 = 0b 0100000 01011 01010 000 01010 $\underbrace{0110011}_{\text{tipo R}}$

   = `sub x10, x10, x11`
   = `sub a0, a0, a1`

6. 0xfedff06f = 0b 1111 1110 1101 1111 1111 00000 $\underbrace{1101111}_{\text{tipo J}}$

   = `jal x0, 0b11111111111111101100`
   = `jal zero, -20`

7. 0x00008067 = 0b 0000 0000 0000 00001 000 00000 $\underbrace{1100111}_{\text{tipo I (}\texttt{jalr}\text{)}}$

   = `jalr x0, x1, 0b000000000000`
   = `jalr zero, ra, 0`

Resumindo, o código Assembly é

```
1    L1:  beq  a0, a1, L2
2         bge  a0, a1, L3
3         sub  a1, a1, a0
4         j    L1
5    L3:  sub  a0, a0, a1
6         j    L1
7    L2:  ret
```

b) Queremos encontrar frequência para o qual o programa acima tenha $t_{\text{exec}} = 4\mu$s para `a0`$= 5$ e `a1`$= 49$. Visto que o processador é Uniciclo, basta determinar quantas instruções serão necessárias para a execução.

Primeiro, o programa irá descontar `a0` de `a1` até que `a1`<`a0`. Essa operação deve ser repetida 9 vezes, resultando em `a1`$= 4$. Até aqui, foram realizadas 36 instruções, 9 vezes os 4 comandos entre `L1` e `L3`.

Seguindo, passa-se pelo `beq` e pula-se no `bge`, subtraindo `a1` de `a0`, ou seja, 4 de 5 – mais 4 instruções.

Agora, com `a0`$= 1$ e `a1`=4, repete-se novamente as instruções entre `L1` e `L3`, só que apenas 3 vezes, acumulando-se 12 instruções.

Finalmente, temos `a0`=`a1`=1, e o `beq` nos leva ao `ret`, 2 instruções.

Ao todo: $\text{I} = 36 + 4 + 12 + 2 = 54$. Conclusão:

$$t_{\text{exec}} = \text{I} \times \text{CPI} \times T_{\text{clk}} \overset{\text{CPI}=1}{\Longrightarrow} f_{\text{clk}} = \frac{\text{I}}{t_{\text{exec}}} = \frac{54}{4 \cdot 10^{-6}}\ \text{Hz} = 13,5\ \text{MHz}.$$

3) Aqui é mais adequado usar shifts ao invés de somas.

```
1            addi  sp, sp, -12     # reserva espaco na pilha
2            sw    a1, 0(sp)       # salva a1
3            sw    t0, 4(sp)       # salva t0
4            sw    t1, 8(sp)       # salva t1
5            mv    t0, a0          # t0 <- a0
6            li    a0, 0           # zera o resultado
7            beq   t0, zero, FIM   # verifica se t0 eh zero
8    LOOP:   andi  t1, a1, 1       # verifica bit menos significativo de t1
9            srli  a1, a1, 1       # desloca t1 para direita
10           beq   t1, zero, PULA  # se lsb for 0, PULA
11           add   a0, a0, t0      # se lsb for 1 soma t0 ao resultado
12   PULA:   slli  t0, t0, 1       # desloca t0 para esquerda
13           bne   a1, zero, LOOP  # verifica se a1 eh diferente de 0
14   FIM:    lw    a1, 0(sp)       # recupera a1
15           lw    t0, 4(sp)       # recupera t0
16           lw    t1, 8(sp)       # recupera t1
17           addi  sp, sp, 12      # libera espaco da pilha
```

4) b) A *tool* Floating Point Representation do RARS nos dá o resultado final da conversão assim como os passos necessários para encontrá-lo.