Fundamentos de Arquitetura de Computadores
Lista de Exercícios 01

Prof. Tiago Alves

Programação em Linguagem de Montagem Assembly MIPS

Entrada e Saída Básica usando MARS

Para resolver as questões abaixo, será necessário um estudo sobre as *syscall*s disponíveis no MARS.

1) Escreva um programa em linguagem de montagem MIPS que imprima na saída em console a mensagem “Alo mundo!”

2) Escreva um programa em linguagem de montagem MIPS que leia um inteiro pelo console e imprima na saída em console o valor lido.

3) Escreva um programa em linguagem de montagem MIPS que leia uma string pelo console e imprima na saída a quantidade de espaços encontrados na string inserida pelo usuário. Limite o bloco de memória onde será guardada a string em 80 bytes (80 caracteres).

Lógica Bit-a-bit

Instruções necessárias para as implementações abaixo:

`and   andi  nor   or    ori   sll   srl   xor   xori`

1) Escreva o seguinte padrão (hexadecimal) no registrador $1: 0x**`DEADBEEF`**.

2) In each register $1 through $7 set the corresponding bit. That is, in register 1 set bit 1 (and clear the rest to zero), in $2 set bit 2 (and clear the rest to zero), and so on. Use only one ori instruction in your program, to set the bit in register $1.

```
ori   $1,$0,0x01
```

Don't use any `ori` instructions other than that one. Note: bit 1 of a register is the second from the right, the one that (in unsigned binary) corresponds to the first power of two.

3) Start out a program with the instruction that puts a single one-bit into register one:

```
ori   $1,$0,0x01
```

Now, by using only shift instructions and register to register logic instructions, put the pattern 0xFFFFFFFF into register $1. Don't use another ori after the first. You will need to use more registers than $1. See how few instructions you can do this in.

4) Put the bit pattern 0x55555555 in register $1. (Do this with three instructions.)

Now shift the pattern left one position into register $2 (leave $1 unchanged).

Put the the bit-wise OR of $1 and $2 into register $3. Put the the bit-wise AND of $1 and $2 into register $4. Put the the bit-wise XOR of $1 and $2 into register $5.

Examine the results.

5) Put the bit pattern 0x0000FACE into register $1. This is just an example pattern; assume that $1 can start out with any pattern at all in the low 16 bits (but assume that the high 16 bits are all zero).

Now, using only register-to-register logic and shift instructions, rearrange the bit pattern so that register $2 gets the bit pattern 0x0000CAFE.

Write this program so that after the low 16 bits of $1 have been set up with any bit pattern, no matter what bit pattern it is, the nibbles in $2 are the same rearrangement of the nibbles of $1 shown with the example pattern. For example, if $1 starts out with 0x00003210 it will end up with the pattern 0x00001230

A. Moderately Easy program: do this using ori instructions to create masks, then use and and or instructions to mask in and mask out the various nibbles. You will also need rotate instructions.

B. Somewhat Harder program: Use only and, or, and rotate instructions.

6) Start out register $1 with any 32-bit pattern, such as 0x76543210. Now, put the reverse of that pattern into register $2, for the example, 0x01234567.

Operações Aritméticas Inteiras

Instruções necessárias para as implementações abaixo:

```
add  sll
addi srl
addiu     sub
addu subu
and  nor
andi xor
or   xori
ori
```

1) Write a program that adds up the following integers:

 456
-229
 325
-552
Leave the answer in register $8.

2) Initialize the sum in register $8 to zero. Then add 4096_10 (radix 10) to $8 sixteen times. You don't know how to loop, yet, so do this by making 16 copies of the same instruction. The value 4096_10 is 0x1000.

Next initialize register $9 to 4096_10. Shift $9 left by the correct number of positions so that registers $8 and $9 contain the same bit pattern.

Finally, initialize aregister $10 to 4096_10. Add $10 to itself the correct number of times so that it contains the same bit pattern as the other two registers.

3) Initialize register $9 to 0x7000. Shift the bit pattern so that $9 contains 0x70000000. Now use addu to add $9 to itself. Is the result correct?

Now use the add instruction and run the program again. What happens?

This may be the only time in this course that you will use the add instruction.

4) Let register $8 be x and register $9 be y. Write a program to evaluate:

```
3x – 5y
```

Leave the result in register $10. Inspect the register after running the program to check that the program works. Run the program several times, initialize x and y to different values for each run.

5) Let register $8 be x. Write a program that computes 13x. Leave the result in register $10. Initialize x to 1 for debugging. Then try some other positive values.

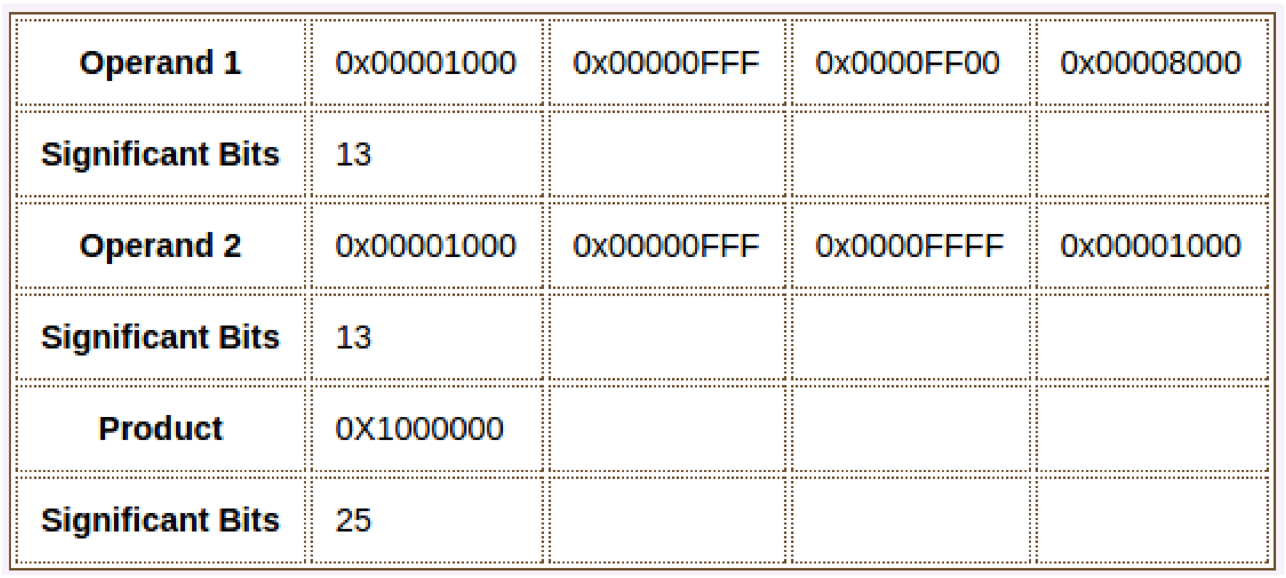
Extend your program so that it also computes -13x and leaves the result in register $11 (This will take one additional instruction.) Now initialize x to -1. Look at your result for 13x. Is it correct?

Operações Aritméticas Inteiras 2

Instruções necessárias para as implementações abaixo:

```
add  mfhi sll
addi mflo sra
addiu     mult srl
addu multu     sub
and  nor  subu
andi or   xor
div  ori  xori
divu
```

1) Write a program that multiplies the contents of two registers which you have initialized using an immediate operand with the ori instruction. Determine (by inspection) the number of significant bits in each of the following numbers, represented in two's complement. Use the program to form their product and then determine the number of significant bits in it.



Operand 1	0x00001000	0x00000FFF	0x0000FF00	0x00008000
Significant Bits	13			
Operand 2	0x00001000	0x00000FFF	0x0000FFFF	0x00001000
Significant Bits	13			
Product	0X1000000			
Significant Bits	25			

2) Write a program that determines the value of the following expression:

`(x*y)/z`

Use x = 1600000 (=0x186A00), y = 80000 (=0x13880), and z = 400000 (=61A80). Initialize three registers to these values. Since the immediate operand of the ori instruction is only 16 bits wide, use shift instructions to move bits into the correct locations of the registers.

Choose wisely the order of multiply and divide operations so that the significant bits always remain in the lo result register.

Operações de Acesso à Memória

Instruções necessárias para as implementações abaixo:

```
add  lw   sll
addi mfhi sra
addiu     mflo srl
addu mult sub
and  multu     subu
andi nor  sw
div  or   xor
divu ori  xori
lui
```

1) Evaluate the expression:

```
17xy - 12x - 6y + 12
```

Use symbolic addresses x, y, and answer. Assume that the values are small enough so that all results fit into 32 bits. Since load delays are turned on in SPIM be careful what instructions are placed in the load delay slot.

Verify that the program works by using several initial values for x and y. Use x=0, y=1 and x=1, y=0 to start since this will make debugging easy. Then try some other values. As an option, follow the precaution for multiplication, as above.

2) Evaluate the polynomial:

```
6x3 - 3x2 + 7x + 2
```

Get the value for x from symbolic addresses x. Store the result at symbolic address poly. Assume that the values are small enough so that all results fit into 32 bits. Since load delays are turned on in SPIM be careful what instructions are placed in the load delay slot.

Evaluate the polynomial by using Horner's Method. This is a way of building up values until the final value is reached. First, pick a register, say $7, to act as an accumulator. The accumulator will hold the value at each step. Use other registers to help build up the value at each step.

First, put the coefficient of the first term into the accumulator:

6
Next, multiply that value by x:

6x
Add the coefficient of the next term:

6x - 3
Next, multiply that sum by x:

6x2 - 3x
Add the coefficient of the next term:

6x2 - 3x + 7
Next, multiply that sum by x:

6x3 - 3x2 + 7x
Finally, add the coefficient of the last term:

```
6x3 – 3x2 + 7x + 2
```

Evaluating the polynomial in this way reduces the number of steps (and the amount of code). If you want to save time, write and debug each step before moving on to the next. When you reach the final step you should have a working, debugged program.

Verify that the program works by using several initial values for x. Use x=1 and x=-1 to start since this will make debugging easy. Then try some other values. As an option, follow the precaution for multiplication.

3) Evaluate the following polynomial using Horner's method:

```
ax3 + bx2 + cx + d
```

Now the values for the coefficients a, b, c, d as well as for x come from the .data section of memory:

```
       .data
x:     .word    1
a:     .word   -3
b:     .word    3
c:     .word    9
d:     .word  -24
```

Load a base register with the address of the first byte of the .data section. Calculate (by hand) the displacement needed for each of the values in memory and use it with a lw instruction to get values from memory. In a later chapter you will find a much more convenient way to load and store values using symbolic addresses. But don't do this now.

Operações de Acesso à Memória 2

Instruções necessárias para as implementações abaixo:

```
add  lhu  sb
addi lui  sh
addiu     lw   sll
addu mfhi sra
and  mflo srl
andi mult sub
div  multu     subu
divu nor  sw
lb   or   xor
lbu  ori  xori
```

lh

1) Your program has a data section declared as follows:

```
.data
.byte   12
.byte   97
.byte  133
.byte   82
.byte  236
```

Write a program that adds the values up, computes the average, and stores the result in a memory location. Is the average correct?

Hint: there are two easily-made errors in this program.

Operações de Controle de Fluxo de Execução

Instruções necessárias para as implementações abaixo:

```
add  divu mflo sll
addi j    mult sra
addiu     lb   multu     srl
addu lbu  nor  sub
and  lh   or   subu
andi lhu  ori  sw
beq  lui  sb   xor
bne  lw   sh   xori
div  mfhi
```

1) Write a program that computes the sum:

1 + 2 + 3 + 4 + 5 + …

Do this by using the j instruction to implement a non-ending loop. Before the loop, initialize a register to zero to contain the sum, and initialize another register to one to be the counter. Inside the loop add the counter to the sum, increment the counter, and jump to the top of the loop.

Execute the program by single-stepping. After you have done this enough to confirm that the program works, enter a number of steps (such as 40) into the window and click "OK".

2) Write a program that adds $8 to itself inside a non-ending loop. Initialize $8 before the loop is entered. Use the add instruction so that when overflow is detected the program ends with a trap.

Now change the add instruction to addu. Now when overflow occurs, nothing happens. Run the program and observe the difference.

3) Write a program that computes the sum:

1 + 2 + 3 + 4 + 5 + ... + 98 + 99 + 100
Do this, as above, by using the j instruction to implement a non-ending loop. Before the loop, initialize a register to zero to contain the sum, initialize another register to one to be the counter, and another register to 10110. Inside the loop add the counter to the sum, increment the counter, and jump to the top of the loop.

However, now, at the top of the loop put in a beq instruction that branches out of the loop when the counter equals 10110. The target of the branch should be something like this:

```
exit:    j    exit       #  sponge for excess machine cycles
         sll  $0,$0,0
```
Now run the program by setting the value of the PC to 0x400000 (as usual) and entering 500 or so for the number of Multiple Steps (F11). Your program will not need so many steps, but that is OK. The excess steps will be used up repeatedly executing the statment labeled "exit".

4) Write a program that computes terms of the Fibonacci series, defined as:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
Each term in the series is the sum of the preceeding two terms. So, for example, the term 13 is the sum of the terms 5 and 8.

Write the program as a counting loop that terminates when the first 100 terms of the series have been computed. Use a register for the current term and a register for the previous term. Each execution of the loop computes a new current term and then copies the old current term to the previous term register.

Notice how few machine language instruction this program takes. It would be hard for a compiler to produce such compact code from a program in a high level language. Of course, our program is not doing any IO.

Operações de Controle de Fluxo de Execução 2

Instruções necessárias para as implementações abaixo:

```
add  div  mflo slt, slti
addi divu mult sltu, sltiu
addiu     j    multu     sra
addu lb   nor  srl
and  lbu  or   sub
andi lh   ori  subu
beq  lhu  sb   sw
bgez lui  sh   xor
bltz lw   sll  xori
bne  mfhi
```

1) Write a program that computes three sums:

1 + 2 + 3 + 4 + ... + 99 + 100

1 + 3 + 5 + 7 + ... + 97 + 99

2 + 4 + 6 + 8 + ... + 98 + 100
Use a register (say $8) for the sum of evens, a register (say $9) for the sum of odds, and another (say $10) for the sum of all numbers.

2) With an ori instruction, initialize $8 to a bit pattern that represents a positive integer. Now the program determines how many significant bits are in the pattern. The significant bits are the leftmost one bit and all bits to its right. So the bit pattern:

0000 0000 0010 1001 1000 1101 0111 1101
has 22 significant bits. (To load register $8 with the above pattern, 0x00298D7D, use an ori followed by a left shift followed by another ori.)

3) A temperature in $8 is allowed to be within either of two ranges: 20 <= temp <= 40 and 60 <= temp <= 80. Write a program that sets a flag (register $3) to 1 if the temperature is in an allowed

range and to 0 if the temperature is not in an allowed range.

It would be helpful to draw a flowchart before you start programming.

Operações em Vetores e Strings

Instruções necessárias para as implementações abaixo:

```
add   div   mflo slt, slti
addi divu mult sltu, sltiu
addiu     j     multu     sra
addu lb    nor   srl
and   lbu   or    sub
andi lh    ori   subu
beq   lhu   sb    sw
bgez lui   sh    xor
bltz lw    sll   xori
bne   mfhi
```

1) Declare a string in the data section:

```
          .data
string:   .asciiz     "ABCDEFG"
```

Write a program that converts the string to all lower case characters. Do this by adding 0x20 to each character in the string. (Examine a tabela ASCII para entender porque essa abordagem funciona.)

2) Declare an array of integers, something like:

```
          .data
  size:   .word 8
  array:  .word 23, -12, 45, -32, 52, -72, 8, 13
```

Write a program that determines the minimum and the maximum element in the array. Assume that the array has at least one element (in which case, that element will be both the minimum and maximum.) Leave the results in registers.

3) In this program data comes in pairs, say height and weight:

```
          .data
  pairs:  .word 5                    # number of pairs
          .word 60, 90               # first pair: height, weight
          .word 65, 105
          .word 72, 256
          .word 68, 270
          .word 62, 115
```

Write a program that computes the average height and average weight. Leave the results in two registers.

Operações em Ponto Flutuante

Atenção: será necessário o uso de syscalls para a resolução de algumas das questões propostas.

1) Write a program that computes value of the following arithmetic expression for values of x and y entered by the user:

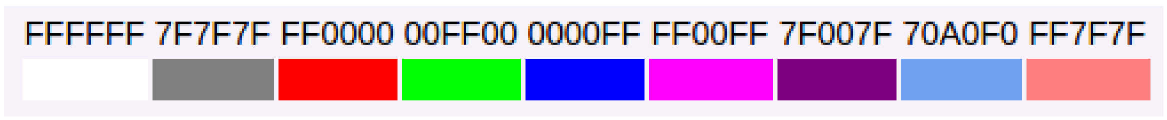
```
5.4xy - 12.3y + 18.23x - 8.23
```

2) Write a program that computes the sum of the first part of the harmonic series:

1/1 + 1/2 + 1/3 + 1/4 + ...
This sum gets bigger and bigger without limit as more terms are added in. Ask the user for a number of terms to sum, compute the sum and print it out.

3) Colors on a Web page are often coded as a 24 bit integer as follows:

RRGGBB
In this, each R, G, or B is a hex digit 0..F. The R digits give the amount of red, the G digits give the amount of green, and the B digits give the amount of blue. Each amount is in the range 0..255 (the range of one byte). Here are some examples:



Another way that color is sometimes expressed is as three fractions 0.0 to 1.0 for each of red, green, and blue. For example, pure red is (1.0, 0.0, 0.0), medium gray is (0.5, 0.5, 0.5) and so on.

Write a program that has a color number declared in the data section and that writes out the amount of each color expressed as a decimal fraction. Put each color number in 32 bits, with the high order byte set to zeros:

```
          .data
color:    .word  0x00FF0000     # pure red, (1.0, 0.0, 0.0)
```

For extra fun, write a program that prompts the user for a color number and then writes out the fraction of each component.

4) Write a program that computes the value of a polynomial using Horner's method. The coefficients of the polynomial are stored in an array of single precision floating point values:

```
.data
n:        5
a:        .float 4.3, -12.4, 6.8, -0.45, 3.6
```

The size and the values in the array may change. Write the program to initialize a sum to zero and then loop n times. Each execution of the loop, with loop counter j, does the following:

```
sum = sum*x + a[j]
```

To test and debug this program, start with easy values for the coefficients.