Laboratório 1 Introdução ao *Assembly*

Linguagem *Assembly* do Microprocessador MIPS

A linguagem *assembly*, ou linguagem de montagem, é uma representação dos comandos executados por um microprocessador. Uma instrução de *assembly* clássico possui dois componentes:

```
add    $s0,$s1,$s2
```
mnemônico (nome para lembrar o propósito) *operandos (registradores, constantes, memória)*

Um registrador é formado por flip-flops em paralelo e é usado como uma variável, guardando um valor numérico. Inicialmente, usaremos apenas os registradores $s0 a $s7.

O utilitário que transforma um programa em código *assembly* para o código de máquina (digamos o "executável", o binário) se chama *montador (assembler),* e não compilador.

Instrução de Soma

Para o MIPS, a instrução básica é a soma dos valores vindos de dois registradores, com o resultado armazenado em um terceiro. Sintaxe: `add <destino>,<fonte1>,<fonte2>`. Por exemplo:

```
add $s3,$s7,$s2  #realiza s3 ← s7+s2
```

Note-se que praticamente todos os montadores não diferenciam maiúsculas de minúsculas, porém podem ser bastante chatos com espaços e tabulações (usualmente, um espaço depois ou antes de uma vírgula faz um montador não reconhecer o comando). Tudo após o # é comentário.

Carga de Constantes

A instrução `addi <destino>,<fonte>,<constante>` irá somar o valor do registrador fonte à constante e armazenar no registrador destino. Já que o registrador $zero ou $0 possui a constante zero armazenada nele (e é não modificável), podemos usar algo como:

```
addi $s0,$zero,345  # carrega a constante 345 em $s0
```

As constantes podem ser negativas. Note ainda que a constante obrigatoriamente deve ser o terceiro operando, não o segundo.

Montador-Simulador Integrado

O MARS da Missouri State University é bastante amigável (mais até do que seria saudável pra disciplina) e em Java. Rodem o bicho, googando "mars mips" se preciso; programem e executem o seguinte:

▶	Carregue as constantes 123, 456 e 789 em $s0, $s1 e $s2. Some tudo e coloquem o resultado em $s3. Confirme na janela de registradores o resultado.

Nota: sempre dê um ou dois TABs no início de cada linha, fazendo um recuo para os comandos. Alguns montadores *exigem* isso, e é boa prática de programação assembly. Ex.:

```
→    →   addi $s0,$zero,43
→    →   add  $s2,$s0,$zero
```

▶	Calcule 30*1234 com o mínimo de instruções que você conseguir. Confirme na janela de registradores o resultado. Não use instruções de multiplicação nem de *branch*.

Saltos Condicionais

A execução das instruções é, a princípio, sequencial. Para quebrarmos a sequência, podemos usar um comando de salto condicional (*branch*), útil para implementar *ifs, elses* e *whiles.*

Inicialmente, usaremos apenas `bne <reg1>,<reg2>,<destino>`, bastante simples: se `<reg1>` é diferente de `<reg2>`, o programa é desviado para `<destino>`; caso contrário, prossegue na execução normal para a próxima instrução. `bne` significa *branch if not equal*.

O destino é dado por um *label* (rótulo) que identifica a linha-alvo. O *label* usualmente começa na primeira coluna e é seguido por dois pontos. Exemplo didático:

```
                addi $s0,$zero,43
                add  $s2,$s0,$zero
                bne  $s2,$zero,pralah
                add  $s0,$s0,$s0      # não será executada
pralah:         addi $s2,$s2,-1       # vai repetir 43 vezes
                bne  $s2,$zero,pralah
                add  $s5,$zero,$s0    # $s5 termina com 43
```

▶	Faça um *loop* de 30 vezes repetindo a soma de 23 em um registrador. Ao final, o valor 30*23 deverá ser carregado em $s5.

▶	Calcule a soma dos 40 primeiros números ímpares e a coloque no registrador $s1.

Exercícios

▶	Calcule $\sum i^2$ para i de 1 a 20. Resultado no registrador $s3. Rode e confira. Não use instruções de multiplicação.

▶	Faça um programa que calcula o fatorial de 8.

Questões Extras

1. O número de instruções disponíveis num *assembly* (números que devem ser decodificados pelos circuitos do processador) pode variar grandemente, indo tipicamente de poucas dezenas a muitas centenas. Cite uma vantagem e uma desvantagem de termos muitas instruções. (Dica: pense tanto no programador quanto no projetista do processador).
2. Atualmente, a tendência é por menos instruções, mais flexíveis e que realizam tarefas mais pontuais. Nos anos oitenta, a situação era invertida. Cite uma vantagem e uma desvantagem de termos instruções mais complexas.
3. Quanto à aridade (número de operandos), as linguagens *assembly* podem variar desde o uso quase exclusivo de operandos implícitos (*stack machines*), as de um operando (um registrador especial, o Acumulador, é sempre fonte e destino implicitamente), e aquelas com instruções de 0 a 3 operandos, mais comuns. Outras que podem incluir múltiplos operandos de endereçamento (e.g., mov eax, [Table + ebx + esi*4] no 8086/Pentium, sendo ebx, eax e esi registradores e Table um endereço). Cite uma desvantagem e uma vantagem de a) uma máquina com apenas um operando e b) uma máquina com um número variável (digamos de zero a quatro) de operandos.
4. Há diversas possibilidades de modificar a estrutura clássica das linguagens *assembly*, incluindo dispositivos na instrução para execuções simultâneas ou execução condicional, ou assemblys que possuem apenas uma instrução. (Veja OISC na Wikipedia).
   Pode-se ter apenas uma instrução `subleq a,b,destino`, por exemplo, que faz [b] ← [a]-[b] e salta para o destino se o resultado não for positivo. Embora o interesse seja apenas acadêmico, pode-se construir um computador completo com apenas esta instrução. Como exercício mental, visualize uma ordenação bolha construída apenas com instruções *subleq*.