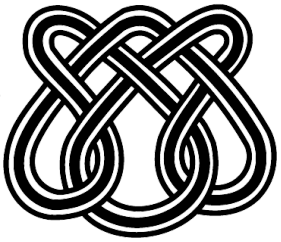

Instituto de Ciências Matemáticas e de Computação

Departamento de Sistemas de Computação

SSC0902 – Organização e Arquitetura de Computadores

Prof(a). Sarita Mazzini Bruschi

Alec Campos Aoki – 15436800
Juan Henriques Passos – 15464826
Jõao Ricardo de Almeida Lustosa – 15463697
Christyan Paniago Nantes – 15635906

A. Especificação

1. A calculadora deverá aceitar os seguintes comandos:

   1.1. + (adição): Soma os dois valores.

   1.2. - (subtração): Subtrai os números.

   1.3. * (multiplicação): Multiplicação dos números.

   1.4. / (divisão): Divisão inteira dos números.

   1.5. u (undo): Desfaz a última operação realizada.

   1.6. f (finalizar): Encerra a execução da calculadora.

2. E/S

   2.1. A entrada dos valores e operações será feita via teclado

   2.2. Os resultados das operações deverão ser exibidos na saída padrão.

   2.3. Após cada operação, deve-se imprimir o resultado.

3. Estrutura:

   3.1. Os resultados devem ser armazenados numa lista encadeada para a operação undo.

   3.2. Deve-se tratar possíveis erros. A operação undo deve imprimir o resultado imediatamente anterior e considerá-lo para operações futuras.

B. Relatório

1. Operações

Todas as operações foram separadas em funções, após receber uma operação é checado se a operação é de controle (*undo* ou *finish*) ou numérica, se for numérica se solicita o número inteiro.

Para cada operação, o endereço da lista é passado como parâmetro pelo registador *a0*, e é recuperado o topo da lista (número no buffer da calculadora) e o número digitado pelo usuário. Esses dois números serão guardados em registradores.

```
# Get number of the last operation
lw a0, 0(s6)                    # a0 = list address
jal list_top                    # a0 = top node number
```

Ao fim de cada operação é criado um novo node para realizar o salvamento na lista de operações (*list_push*) e é printado o resultado da operação (*print_result*).

```
# Creates new node with current list address and the result of the sum
lw a0, 0(s6)                    # a0 = s6(list address)
mv a1, s9                       # a1 = s9(result of current operation)
jal list_push                   # Insert s9 on the top of the list

# Print result of current operation
mv a0, s9                       # a0 = s9(result of current operation)
jal print_result                # Call function format output result

j calculator_on                 # Continue for more operations
```

1.1. Adição

Na operação de adição, a instrução *add* é realizada com os dois registradores que contém os números a serem operados, e o resultado é armazenado em um terceiro.

1.2. Subtração

Na operação de subtração, a instrução *sub* é realizada com os dos dois registradores que contém os números a serem operados, e o resultado é armazenado em um terceiro.

1.3. Multiplicação

Na operação de multiplicação, a instrução *mul* é realizada com os dos dois registradores que contém os números a serem operados, e o resultado é armazenado em um terceiro.

1.4. Divisão

Na operação de divisão, a instrução *div* é realizada com os dos dois registradores que contém os números a serem operados, e o resultado é armazenado em um terceiro.

1.5. Undo

Na operação de *undo* é realizada a remoção da última operação através da função *list_pop*. Nota-se que se a lista tiver tamanho 1 ou 0(caso undo primeira operação), a operação undo dispara erro de nenhuma operação anterior(Seção 3.4.2), tendo em vista a lógica de tratamento da primeira entrada, no qual esse elemento restante não é uma operação realizada, caso contrário, apenas imprime o resultado anterior a operação anterior e continua o loop principal da calculadora.

```
# Remove last operation
lw a0, 0(s6)                    # Load adress of the list
jal list_pop
```

1.6. Finalizar

Na operação de finalizar é realizado o print do histórico de resultados e o encaminhamento para o fim do programa.

```
# Prepare output to print history of results
li a7, 4                        # Syscall 4: print string
la a0, history_of_res           # Output message
ecall                           # Syscall

# Print results
lw a0, 0(s6)                    # a0 = address of list
jal list_print                  # Print list elements (numbers)

# Finish loop
j calculator_off                # Jump to calculator_off
```

2. E/S

2.1. Entrada de valores e operações

Recebemos como primeira entrada um inteiro, uma operação e outro inteiro. Para tratar a primeira entrada de forma consistente e evitar repetição de código, o primeiro inteiro fornecido é armazenado em uma lista como se fosse o resultado de uma operação prévia. Dessa maneira, desde o início, o programa já executa o loop principal processando as entradas de forma unificada.

Vale ressaltar que devido a essa abordagem, foi necessário implementar uma lógica específica para tratar o primeiro input, conforme detalhado na Seção 1.5.

```
# Insert first number into list to avoid repeated code
mv a1, a0               # Move input to a1
lw a0, 0(s6)            # Move list adress a0
jal list_push           # Add number (int) to the list
```

2.2. Saídas

2.2.1. Mensagens sobre inserção ou saída de dados

Ocasião	Mensagem
Introdução ao usuário (*help_msg*)	Calculator with operations '+', '-', '*', '/', 'u', 'f' Special Operations: u (undo last op), f (finish) First input: <number> <op> <number> Next inputs: <op> <number> Performs operation between last result and input
Requisitar número ao usuário (*read_int_msg*)	Type number:
Requisitar operação ao usuário (*operation_msgs*)	Type the operation:
Imprimir resultado da operação realizada (*result*)	Result =
Imprimir histórico de resultados (*history_of_res*)	History of results =

2.2.2. Mensagens de erro

Ocasião	Mensagem
Ponteiro para lista nulo (*msg_null_list*)	Error: null list.
Divisão por zero (*msg_div_by_zero*)	Error: division by zero is not allowed.
Overflow (*msg_overflow*)	Error: overflow has occurred.
Undo antes de a primeira operação não foi realizada (*msg_no_previous_op*)	No previous operation.
Undo quando não houverem mais operações a serem desfeitas (*msg_no_previous_op_remains*)	No previous operation remains.
Operação inválida (*msg_invalid_op*)	Invalid operation, try again.

3. Estrutura (Lista Encadeada)

3.1 Funcionamento

Tratando-se de uma lista simplesmente encadeada, a estrutura "lista" consiste de um espaço de 8 bytes na memória heap, apontado por um ponteiro (espaço que armazena um endereço) na *RAM*. Os primeiros 4 bytes desse espaço guardam o endereço e guardam o endereço do "nó" no início da lista. Os próximos 4 bytes armazenam o tamanho atual da lista como um inteiro.

Um nó consiste de outro espaço de 8 bytes na memória heap. Os primeiros 4 bytes do nó guardam o endereço do nó seguinte. Os outros 4 bytes armazenam o valor daquele nó, ou seja, o inteiro resultante de uma operação realizada pelo usuário.

Nas funções da lista, seu endereço deve ser passado como parâmetro pelo registrador *a0*.

3.2 Funções

3.2.1 *list:* Função que cria a estrutura da lista, armazenando o endereço para um nó(4 bytes) e o tamanho atual da lista (4 bytes), esse espaço é alocado

dinamicamente na heap.

```
# Control structure consists of a pointer to the first node
li a7, 9                    # Syscall code 9: allocate memory on heap
li a0, 8                    # Size to be allocated = 4 bytes(adress) + 4 bytes(list size)
ecall                       # Syscall to allocate memory on the heap
```

3.2.2 *list_push*: Função que adiciona um novo elemento no início da lista por meio da alocação dinâmica. É alocado espaço para um nó, que consiste em um endereço para o próximo e um valor a ser armazenado.

```
# Create new node, and save data(adress of next node and number)
lw t1, 0(t0)                # Loading to t1 the address of the first/top node on the list
sw t1, 0(a0)                # Storing the address of the first/top node on the list into the new node
sw a1, 4(a0)                # Storing the value into the new node
sw a0, 0(t0)                # Making the new node the first/top node of the list
```

3.2.3 *list_pop*: Função responsável por remover o elemento do início da lista encadeada. Dado o endereço da lista, se ela estiver vazia, encerra-se a função, caso contrário o início da lista será o nó apontado pelo nó.

```
# Update list head next node
lw t1, 0(t0)                # t1 = next node address
sw t1, 0(a0)                # Update list head
```

3.2.4 *list_top*: Função responsável por retornar valor do topo(início) da lista.

```
# Get the first element(number) in the list
lw t0, 0(a0)                # t0 = address to top node
lw t1, 4(t0)                # t1 = top node number
mv a0, t1                   # a0 = top node number
```

3.2.5 *list_print*: Função que percorre a lista e imprime os valores de cada elemento.

3.2.6 *list_empty*: Função que verifica se a lista está vazia, retorna verdadeiro se vazia e falso caso contrário.

```
# Get the first byte from the address of the first node
lw t0, 0(a0)                # t0 = address of top node
# if t0(adress of 1st node) = 0, list is empty
seqz a0, t0                 # t0 == 0 ? 1:0
```

3.3 Erros

3.3.1 *error null list*: Ocorre erro caso usuário forneça uma lista com endereço 0(NULL, por definição), erro fatal e encerra o programa.

```
# Catch possible error(dont try to acess null pointer)
beqz t0, error_null_list# t0 = 0, list dont exist(null pointer)
```

3.3.2 *error div by zero*: Não é possível realizar uma divisão por zero, isto é um erro fatal e o programa é encerrado (s8 = divisor)

```
# Check if the divider is zero
beqz s8, error_div_by_zero     # Cant div by zero
```

3.3.3 *error no previous op*: Ocorre ao tentar realizar a operação undo como primeira operação, ou quando não houver operações anteriores ao realizar undo (a0 = tamanho).

```
# Because the logic of implementation if a0(size) <= 1,
# there is no last operation,
# because this one element is the 1st input.
slti t0, a0, 2                # Case occurs when all the operations is removed
bnez t0, error_no_previous_op  # and only remains the 1st input or none
```

3.3.4 *error overflow*: Erro ocorre quando o resultado da operação não pode ser guardado em um registrador de 32 bits.

- Lógica soma: o overflow da soma de dois números ocorre quando eles possuem o mesmo sinal e o seu resultado possui o sinal oposto, dessa forma, foi feito uma verificação implícita desses quesitos.

```
# Overflow occurs when:
# 1. Two positive numbers are added together and the result is negative
# 2. Two negative numbers are added together and the result is positive
# Check overflow
slti t0, a0, 0                # t0 = (a0 < 0) - is a0 neg? 1:0
slt t1, s9, s8                # t1 = (s8 + a0 < s8) - sum result lower number? 1:0
bne t0, t1, error_overflow    # overflow if (a0 < 0) && (s8 + a0 >= s8)
                              # || (a0 >= 0) && (s8 + a0 < s8)
```

- Lógica subtração: o overflow da subtração de dois números ocorre quando eles possuem sinais opostos e o resultado possui o mesmo sinal do subtraendo, nesse caso, foi implementado uma lógica direta de análise de sinais.

```
# Overflow occurs when:
# 1. Subtracting a negative from a positivea and result is negative
# 2. Subtracting a positive from a negative and result is positive
# Check overflow
slti t0, a0, 0                # t0 = (a0 < 0) - is a0 neg? 1:0
slti t1, s8, 0                # t1 = (s8 < 0) - is s8 neg? 1:0
# t1 = number have opposite signs
xor t0, t1, t0                # if the numbers have opposite signs, possible overflow
# t2 = result(s9) has the same sign s8
slti t2, s9, 0                # s9 = (s9 < 0) - is s9 neg? 1:0
# xnor t2, t2, t1
xor t2, t2, t1
xori t2, t2, 1
# overflow if t1 = 1 and t2 = 1
and t0, t0, t2
li t2, 1
beq t0, t2, error_overflow
```

- Lógica multiplicação: ocorre overflow na multiplicação quando a parte alta do resultado não é uma extensão do sinal da parte baixa, pois nesse caso, foi necessário mais de 32 bits para armazenar o número.

```
# Overflow occurs when:
# The high part of the product operation is different of the sign extended
# Check overflow
mulh t2, s8, a0               # t2 = high part of mul operation (signed)
mul t3, s8, a0                # t3 = low part of mul
srai t4, t3, 31               # t4 = sign extended from low part (0 or -1)
xor t2, t2, t4                # If t2 != t4, have overflow
bnez t2, error_overflow       # t2 != 0, overflow
```

- Lógica divisão: só há um caso de overflow na divisão, tendo em vista que o número sempre diminui. Esse caso é quando dividimos o inteiro mínimo por -1, pois obtemos um valor que é 1 unidade maior que o inteiro máximo, dessa forma, analisamos o dividendo e o divisor para esse caso.

```
# Overflow occurs when:
# if INT_MIN / -1 = 2^31, but INT_MAX is 2^(31) - 1
li   t0, -1                           # t0 = -1
lui  t1, 0x80000                      # t1 = INT_MIN (0x80000000 to 32 bits)
bne  s8, t0, no_overflow      # If divider != -1, dont have overflow
beq  a0, t1, error_overflow   # INT_MIN / -1 -> Overflow
```