

# **Universidade de São Paulo**

## **Escola Politécnica**



### **PCS3732 - Laboratório de Processadores**

#### **Prova prática - Relatório**

Nome: Luiz Guilherme Budeu

NUSP: 11821639

**São Paulo, 15 de agosto de 2023**

# Projeto: Compilador de Expressões Aritméticas para ARMv7 Assembly

## Introdução:

O projeto escolhido consiste em um compilador de expressões aritméticas para a linguagem de montagem ARMv7, implementado em Python. O objetivo deste projeto era criar uma ferramenta capaz de analisar expressões aritméticas, construir uma árvore de sintaxe abstrata (AST) correspondente e gerar código de montagem ARMv7 que executa as operações aritméticas especificadas nas expressões.

O compilador suporta as operações de adição, subtração, multiplicação e divisão de inteiros *signed*. Por se tratar de um desenvolvimento individual, não foi possível adicionar suporte para números de ponto fixo e racionais.

Para a interação e a execução do projeto, é necessário clonar o repositório <https://github.com/LuizBudeu/LabProc>. Realizado esse passo, basta digitar a expressão aritmética desejada em notação infixa (exemplo:  $1 + 2 * 3 - (4/2)$ ), rodar o arquivo "main.py" e obter o código *assembly* ARMv7 correspondente localizado em "out.s". O resultado final da expressão estará localizado no registrador *r0*.

## Solução:

A solução do projeto foi dividida em três partes principais: o analisador léxico (*Lexer*), o analisador sintático (*Parser*) e, por fim, o gerador de código de montagem ARMv7 (*ARMCodeGenerator*). Primeiro, é acionado o *Lexer*, transformando os caracteres da expressão aritmética em *tokens*. Em seguida, o *Parser* constrói uma Árvore de Sintaxe Abstrata (AST) utilizando os *tokens* fornecidos. E por fim, o gerador de código é responsável por atravessar a AST, seguindo as operações e números, simulando o funcionamento de registradores e imprimindo o código *assembly* equivalente.

### Analizador Léxico (*Lexer*):

O analisador léxico é responsável por transformar a sequência de caracteres de entrada em uma sequência de *tokens* significativos para o analisador sintático. Isso é realizado por meio da identificação e classificação de palavras-chave, operadores e valores numéricos presentes na expressão. No caso deste projeto, os tipos de token incluem INTEGER, PLUS, MINUS, MULTIPLY, DIVIDE, LPAREN (parêntese esquerdo) e RPAREN (parêntese direito).

O lexer é implementado como uma classe Python chamada *Lexer*. Ela percorre a entrada caractere por caractere, construindo os tokens à medida que encontra padrões específicos. A classe fornece um método `get_next_token()` que retorna o próximo *token* da sequência de entrada, permitindo que o analisador sintático processe os tokens adequadamente, e eliminando espaços em branco desnecessários.

```

○ Token(TokenType.INTEGER, 3)
  Token(TokenType.PLUS, +)
    Token(TokenType.INTEGER, 4)
      Token(TokenType.MULTIPLY, *)
        Token(TokenType.LPAREN, ()
          Token(TokenType.INTEGER, 10)
            Token(TokenType.MINUS, -)
              ● Token(TokenType.INTEGER, 5)
                Token(TokenType.RPAREN, ))
                  Token(TokenType.DIVIDE, /)
                    Token(TokenType.INTEGER, 4)
                      Token(TokenType.MULTIPLY, *)
                        Token(TokenType.LPAREN, ()
                          Token(TokenType.MINUS, -)
                            Token(TokenType.INTEGER, 1)
                              Token(TokenType.RPAREN, ))

```

Figura 1: Exemplo de saída do *Lexer* para a expressão aritmética:  
 $3 + 4 * (10 - 5) / 4 * (-1)$

#### Analizador Sintático (*Parser*):

O analisador sintático é responsável por construir a Árvore de Sintaxe Abstrata (AST) a partir da sequência de tokens fornecida pelo *Lexer*. A AST representa a estrutura hierárquica da expressão, capturando as relações entre os operadores e operandos. Nesta etapa, os tokens são agrupados em nós que representam operações binárias (BinOp) e números (Num).

O parser é implementado como uma classe Python chamada *Parser*. Ele usa o *Lexer* para obter os *tokens* e, em seguida, implementa regras gramaticais para construir o AST. O parser segue a técnica de análise descendente recursiva para construir o AST, percorrendo os tokens da entrada e construindo os nós do AST conforme necessário.

```

BinOp(TokenType.PLUS)
  Num(3)
  BinOp(TokenType.MULTIPLY)
    Num(4)
    BinOp(TokenType.MINUS)
      Num(10)
      BinOp(TokenType.DIVIDE)
        Num(5)
        BinOp(TokenType.MINUS)
          BinOp(TokenType.MULTIPLY)
            Num(1)
            Num(2)
            Num(4)

```

Figura 2: Exemplo de saída do *Parser* para a expressão aritmética:  
 $3 + 4 * (10 - 5 / (1 * 2 - 4))$

### Gerador de Código de Montagem ARMv7 (ARMCodeGenerator):

O gerador de código de montagem ARMv7 (ARMCodeGenerator) é responsável por transformar o AST em código de montagem executável compatível com a arquitetura ARMv7. Ele percorre o AST e gera as instruções de montagem apropriadas para executar as operações matemáticas especificadas na expressão.

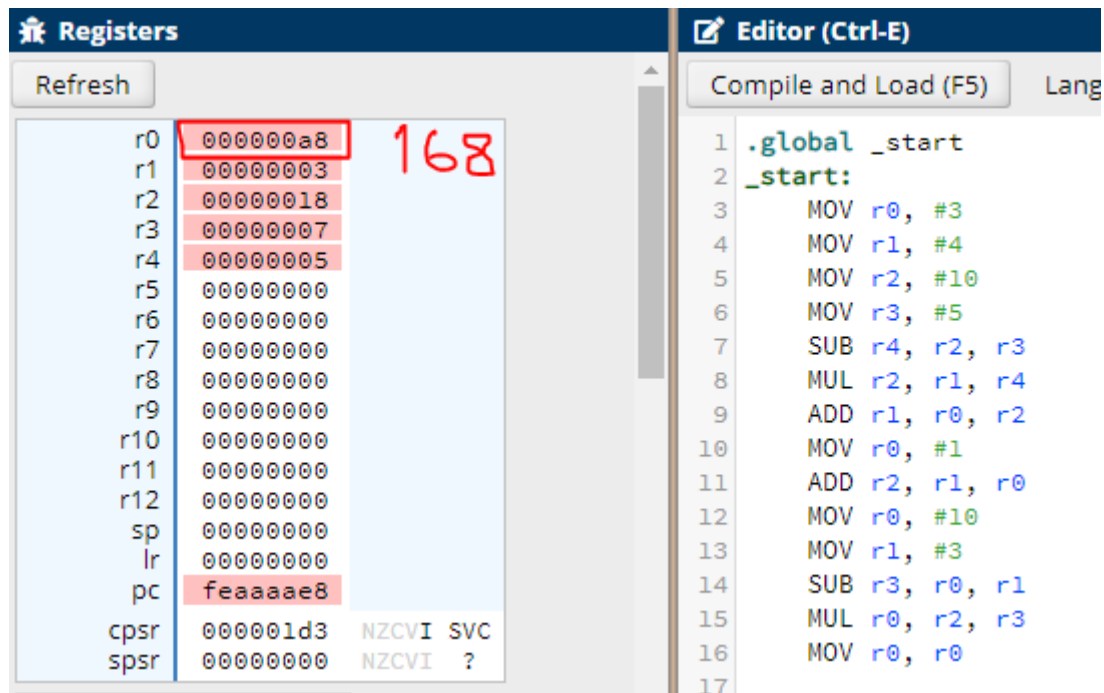
O ARMCodeGenerator é implementado como uma classe Python que percorre o AST usando uma abordagem de percurso em profundidade. Para cada nó do AST, o gerador de código gera as instruções de montagem apropriadas com base no tipo de nó e no valor associado. A classe mantém o controle do uso dos registradores ARM para evitar conflitos e garante a liberação adequada dos registradores após o uso.

Um registrador é representado por uma *dataclass* que contém dados como *id*, *content* (conteúdo do registrador), e *in\_use* (indicando se ele está em uso naquele instante). O ARMCodeGenerator cria uma lista de registradores para simular um banco de registradores, sendo configurável o número máximo de registradores disponíveis.

```
MOV r0, #3
MOV r1, #4
MOV r2, #10
MOV r3, #5
SUB r4, r2, r3
MUL r2, r1, r4
ADD r1, r0, r2
MOV r0, #1
ADD r2, r1, r0
MOV r0, #10
MOV r1, #3
SUB r3, r0, r1
MUL r0, r2, r3
MOV r0, r0
```

**Figura 3: Exemplo do código de saída do ARMCodeGenerator para a expressão aritmética:  $(3 + 4 * (10 - 5) + 1) * (10 - 3)$**

Na figura de exemplo acima, é possível observar que o resultado da expressão aritmética é igual a 168. Colocando o código *assembly* gerado no *CPUlator* (<https://cpulator.01xz.net/?sys=arm>), nota-se que o resultado final das operações é colocado em *r0*, correspondendo ao valor hexadecimal “A8”, que é igual a 168 em decimal:



**Figura 4: Saída do código *assembly* executado no CPUlator**

### Observações adicionais:

Uma observação adicional importante a se fazer é que não existe no conjunto de instruções ARMv7 uma instrução própria de divisão. Por causa disto, foi utilizada a implementação de uma subrotina de divisão inteira realizada em sala de aula, disponível no fórum (<https://www.wise-ware.com.br/pcs3732/d/68-divisao-binaria-bis/2>):

```
;@ r0 - Divisor
;@ r1 - Dividendo
;@ r2 - Resto
;@ r3 - Resultado
;@ r4 - Contador de shifts
_divide:
    cmp r0, #0
    beq _end_divisao    ;@ Retorna se divisor for zero
    mov r3, #0          ;@ Zera quociente
    mov r2, #0          ;@ Zera parcela
    clz r4, r1           ;@ r4 recebe o número de zeros a esquerda do dividendo e será o ocntador
de deslocamento
    movs r1, r1, lsl r4 ;@ Desloca dividendo para remover zeros a esquerda
_desloca:
    cmp r4, #32
    beq _end_divisao
    mov r2, r2, lsl #1   ;@ Desloca parcela
    mov r3, r3, lsl #1   ;@ Desloca quociente
    movs r1, r1, lsl #1 ;@ Desloca dividendo
    add r4, r4, #1
    bcc _carry_zero
    add r2, r2, #1       ;@ Soma 1 a parcela quando o carry eh 1
_carry_zero:
```

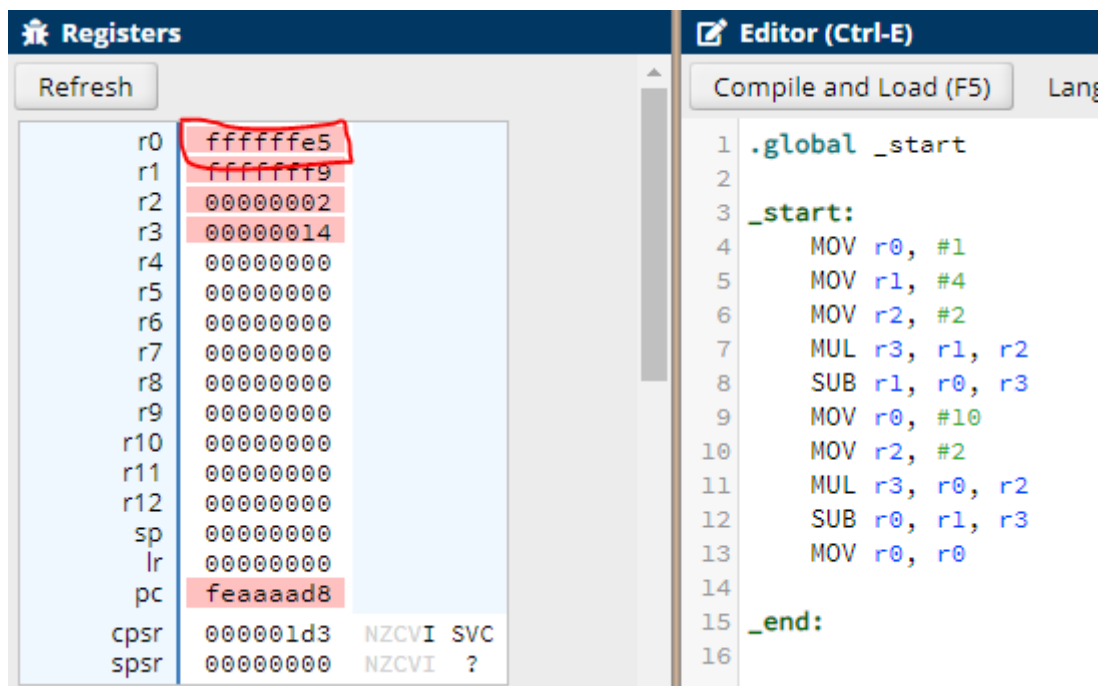
```

    cmp r2, r0
    bge _subtracao    ;@ Parcela maior ou igual ao divisor, subtrai o valor do divisor dele
    b _desloca
_subtracao:
    sub r2, r2, r0
    add r3, r3, #1
    b _desloca
_end_divisao:
    mov pc, lr        ;@ Retorno

```

Dessa maneira, para as expressões aritméticas que contêm operações de divisão, é inserido a subrotina de divisão ao começo do programa, e realizando um *branch-link (bl)* para a subrotina. Para acomodar essa solução, caso haja operações de divisão na expressão aritmética, alguns dos registradores são reservados para realizar a execução da subrotina corretamente, sendo estes *r0-r4*.

Outra observação importante é que o valor do resultado da execução do código *assembly* é *signed*. Isso significa que é possível representar os resultados de expressões que dão negativo. Toma-se o exemplo:  $(1 - 4 * 2) - 10 * 2 = -27$



**Figura 5: Saída do código *assembly* no CPULator**

É possível notar que o resultado da execução do programa foi “**FFFFFFE5**”, valor hexadecimal equivalente ao valor binário “**11100101**”, o que, em complemento de 2, converte para o valor **-27** decimal, mostrando o correto funcionamento do código.

## Testes e verificação:

A seguir, alguns testes, variando de mais simples para mais complexos, a fim de demonstrar o correto funcionamento do programa:

1)  $2 + 1 = 3$

The screenshot shows the ARM debugger interface. On the left, the 'Registers' window displays the state of registers r0 through r10. Register r0 is highlighted with a red box and contains the value 00000003. On the right, the 'Editor (Ctrl-E)' window shows the assembly code for the first test:

```
1 .global _start
2
3 _start:
4     MOV r0, #2
5     MOV r1, #1
6     ADD r2, r0, r1
7     MOV r0, r2
8
9 _end:
```

Saída: hex: 3, decimal: 3

2)  $(3 + 4 * (10 - 5) + 1) * (10 - 3) = 168$

The screenshot shows the ARM debugger interface for the second test. On the left, the 'Registers' window displays the state of registers r0 through r12, sp, lr, pc, cpsr, and spsr. Register r0 is highlighted with a red box and contains the value 000000a8. On the right, the 'Editor (Ctrl-E)' window shows the assembly code for the second test:

```
1 .global _start
2
3 _start:
4     MOV r0, #3
5     MOV r1, #4
6     MOV r2, #10
7     MOV r3, #5
8     SUB r4, r2, r3
9     MUL r2, r1, r4
10    ADD r1, r0, r2
11    MOV r0, #1
12    ADD r2, r1, r0
13    MOV r0, #10
14    MOV r1, #3
15    SUB r3, r0, r1
16    MUL r0, r2, r3
17    MOV r0, r0
18
19 _end:
```

Saída: hex: A8, decimal: 168

$$3) 1 - 3 \cdot 4 - (3 \cdot 3) = -20$$

The screenshot shows a debugger interface with two main panels. The left panel, titled 'Registers', displays the state of various registers. The right panel, titled 'Editor (Ctrl-E)', shows the assembly code being executed.

**Registers Panel:**

Register	Value	Flags
r0	ffffffec	
r1	ffffff5	
r2	00000003	
r3	00000009	
r4	00000000	
r5	00000000	
r6	00000000	
r7	00000000	
r8	00000000	
r9	00000000	
r10	00000000	
r11	00000000	
r12	00000000	
sp	00000000	
lr	00000000	
pc	feaaaaad8	
cpsr	000001d3	NZCVI SVC
spsr	00000000	NZCVI ?

**Editor Panel:**

```

1  .global _start
2
3  _start:
4      MOV r0, #1
5      MOV r1, #3
6      MOV r2, #4
7      MUL r3, r1, r2
8      SUB r1, r0, r3
9      MOV r0, #3
10     MOV r2, #3
11     MUL r3, r0, r2
12     SUB r0, r1, r3
13     MOV r0, r0
14
15  _end:
16
  
```

**Saída:** hex: FFFFFFFEC, decimal (complemento de dois): -20

$$4) (1 - 4 / 2) - 10 / 2 = -6$$

The screenshot shows a debugger interface with two main panels. The left panel, titled 'Registers', displays the state of various registers. The right panel, titled 'Editor (Ctrl-E)', shows the assembly code being executed.

**Registers Panel:**

Register	Value	Flags
r0	fffffffa	
r1	00000000	
r2	00000000	
r3	00000005	
r4	00000020	
r5	fffffffa	
r6	fffffffa	
r7	00000002	
r8	00000005	
r9	00000000	
r10	00000000	
r11	00000000	
r12	00000000	
sp	00000000	
lr	00000088	
pc	00000098	
cpsr	600001d3	NZCVI SVC
spsr	00000000	NZCVI ?

**Editor Panel:**

```

34
35  _start:
36      MOV r5, #1
37      MOV r6, #4
38      MOV r7, #2
39      MOV r1, r6 ;@ dividendo
40      MOV r0, r7 ;@ divisor
41      bl _divide
42      MOV r8, r3 ;@ resultado
43      SUB r6, r5, r8
44      MOV r5, #10
45      MOV r7, #2
46      MOV r1, r5 ;@ dividendo
47      MOV r0, r7 ;@ divisor
48      bl _divide
49      MOV r8, r3 ;@ resultado
50      SUB r5, r6, r8
51      MOV r0, r5
52
53  _end:
  
```

**Saída:** hex: FFFFFFFFA, decimal (complemento de dois): -6



**Conclusão:**

Em conclusão, o desenvolvimento deste projeto resultou em um compilador de expressões aritméticas para a linguagem de montagem ARMv7, destacando as etapas fundamentais do processo de compilação, desde a análise léxica até a geração de código de montagem. A implementação bem-sucedida do lexer, parser e gerador de código de montagem permitiu a criação de uma ferramenta versátil capaz de interpretar e executar expressões matemáticas complexas.