COMPILADOR P3

AUTOR: Ricardo Farinha Gomes da Silva

Objectivo

Este projecto consiste na construção de um compilador para a linguagem **MOCC (Mini-Orientada a Código Compilado)**, definida no livro *Computer Architecture: Digital Circuits to Microprocessors* de Guilherme Arroz e José Monteiro.

O compilador foi implementado em Java com recurso à ferramenta **ANTLR v4.13.2**.

Realiza a análise léxica, sintática e a geração de código intermédio (**TAC – Three Address Code**), suportando variáveis, expressões aritméticas, leitura, escrita e estruturas de controlo (`if`, `while`, `for`).

O destino final é a geração de **código Assembly P3**, totalmente compatível com o simulador P3JS um assembler e simulador open source para o processador P3 que corre diretamente no browser ou em Node.js.

Principais Funcionalidades

- Análise léxica e sintática com ANTLR.

- Análise semântica via Visitor.

- Geração de **Three Address Code (TAC)**.

- Otimização básica de código.

- Backend para **Assembly P3** (simulado no P3JS).

Estrutura

O compilador P3 incluí:

Analisador léxico e sintático gerado por ANTLR4 a partir da gramática MOC.g4;

Gramática da Linguagem MOCC

A linguagem MOCC (Mini-Orientada a Código Compilado) foi formalizada em ANTLR v4.13.2.

A gramática define a sintaxe da linguagem, suportando variáveis, expressões aritméticas, estruturas condicionais e ciclos (`if`, `while`, `for`).

Excerto de `MOC.g4`###

```antlr
prog        : (decl |
func)* EOF ;

decl        : tipo ID (‘=’ expr)? (‘,’ ID (‘=’ expr)?)* ‘;’ ; tipo
: 'int' | 'double' ;

stmt        : assignStmt
            | ifStmt
            | whileStmt
            | forStmt
            | block
            ;

ifStmt      : 'if' '(' expr ')' stmt ('else' stmt)? ; whileStmt   :
'while' '(' expr ')' stmt ; forStmt     : 'for' '(' assignStmt
expr ';' assignStmt ')' stmt ;

expr        : expr op=('*'|'/') expr
            | expr op=('+'|'-') expr
            | INT
            | ID
            | '(' expr ')'
            ;
```

Está implementado um analisador semântico com detecção de erros e tabela de símbolos;

Geração de código intermédio (TAC – Three Address Code) implementada em CodeGenFinal.java;

Programa principal MainMOC.java que compila ficheiros .moc e gera TAC.

Criação do módulo CodeGenP3.java, que:

Traduz o TAC para instruções da linguagem assembly P3 ;

Implementa separação correcta entre zona de dados (`ORIG 8000h`) e zona de código (`ORIG 0000h`);

Gera instruções válidas como `MOV, ADD, BR, CMP, JMP`, etc.

Gera o código utilizável no ficheiro `programa.as`

Integração com MainMOC.java

Após a geração de TAC, o MainMOC invoca automaticamente o CodeGenP3 para emitir o ficheiro `programa.as` dentro da pasta;

O ficheiro gerado no ficheiro `programa.as` é directamente carregável no simulador P3JS.

Ficheiro Executável

O ficheiro **`compilador-moc.jar`**, que constitui a versão empacotada e executável do compilador. Este ficheiro .jar permite compilar programas .moc diretamente, sem necessidade de recompilar o código fonte. O ficheiro pode ser executado através do terminal com o comando: `java -jar compilador-moc.jar ProgramasExemplo/exemplo_if.moc`

.

Validação de exemplos funcionais

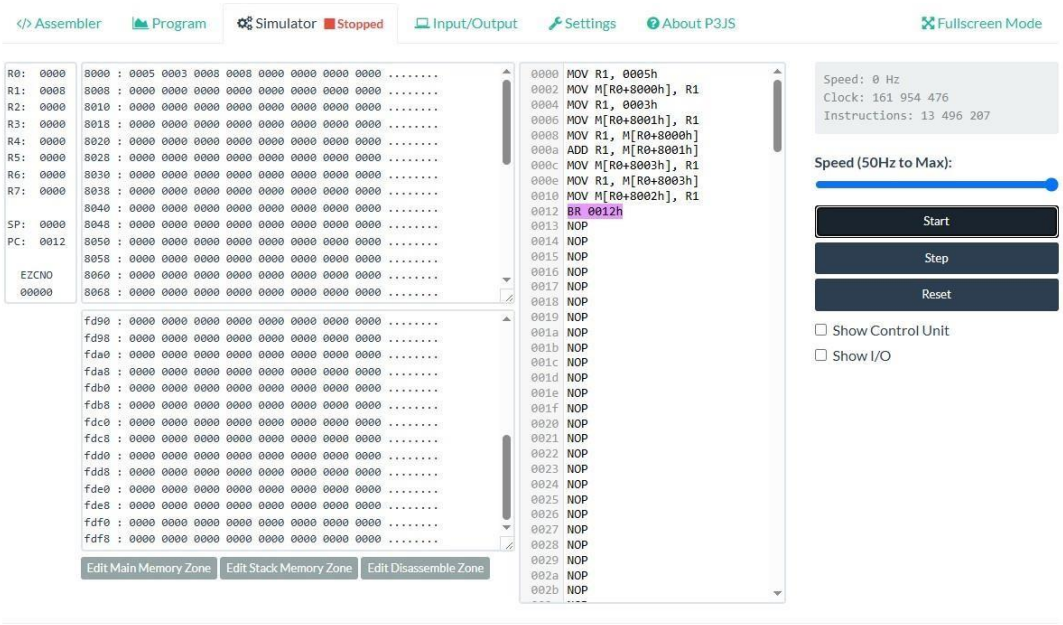
1-Exemplo.moc

O programa `exemplo.moc` de teste onde primeiro e é alocada memória e depois corre o programa:

```
int a;
int b;
int c;
void main(void) {
        a = 5;
        b = 3;
        c = a + b;
```

```
}
```

Foi correctamente traduzido para o seguinte código P3:




```
ORIG 8000h a
WORD 0 b
WORD 0 c
WORD 0 t0
WORD 0
ORIG 0000h
MOV R1, 5
MOV M[a], R1 MOV
R1, 3
MOV M[b], R1
MOV R1, M[a]
ADD R1, M[b]
MOV M[t0], R1
MOV R1, M[t0] MOV
M[c], R1
Fim: BR Fim
```

A simulação no P3JS confirmou que a = 5, b = 3 e c = 8, com execução correta e sem erros.

Notas sobre o funcionamento

A memória é corretamente inicializada antes da execução (ORIG 8000h);

As variáveis temporárias (t0, etc.) são tratadas como células de memória;

O programa termina com um loop fixo (Fim: BR Fim) para garantir estado estável após execução.

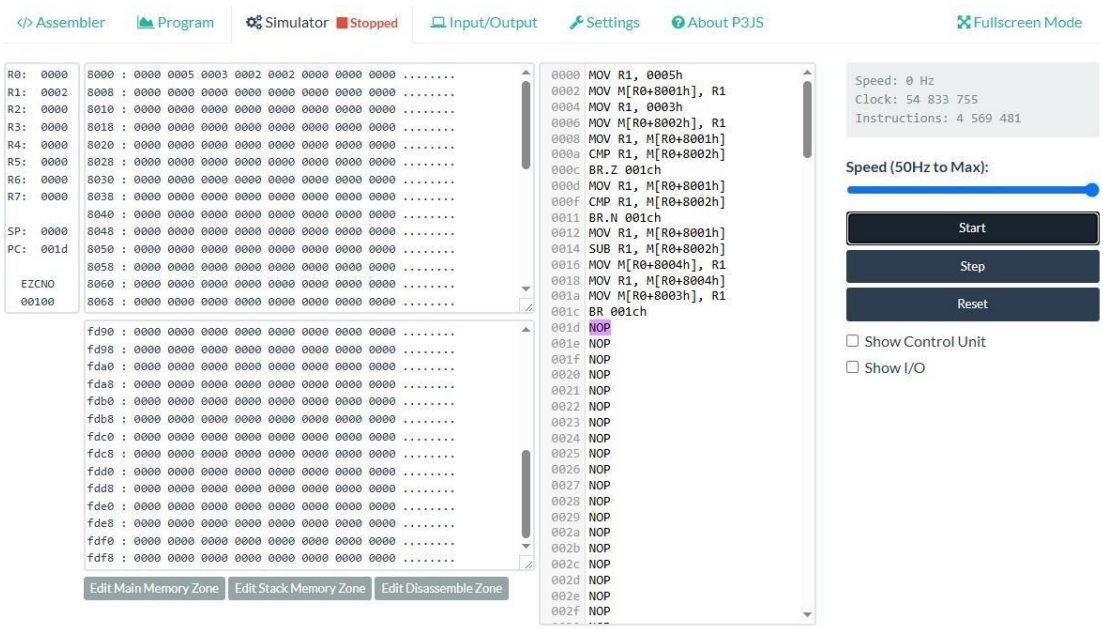
2-exemplo_if.moc

Compilar corretamente um bloco condicional com o script:

```
int a;
int b;
int c;
void main(void) {
        a = 5;
        b = 3;
        if (a > b) {
                c = a-b;
    }
}
```

Resultados obtidos

O simulador P3JS aceitou o .as gerado sem erros



Executou corretamente o fluxo condicional

Registo R1 e zonas de memória confirmaram a precisão da execução

3- exemplo_while.moc

O código original em C/MOC:

```
int i;
int total;
void main(void) {
     i = 1;
     total = 0;
     while (i <= 5) {
          total = total + i;
          i = i + 1;
    }
}
```

Representação em TAC (Three Address Code)

O compilador gera corretamente o seguinte código intermédio:

```
i = 1
total = 0
L0:
IF_FALSE i <= 5 GOTO L1
```

```
t2 = total + i
total = t2
t3 = i + 1
i = t3
GOTO L0
L1:
```

L0 marca o início do ciclo

IF_FALSE representa a condição de saída as instruções

dentro do ciclo usam temporários t2, t3

GOTO L0 reitera o ciclo

Tradução para P3 Assembly

O código final gerado no `programa.as` é:

```
MOV R1, 1
MOV M[i], R1 MOV
R1, 0 MOV
M[total], R1


; condição do while
MOV R1, M[i]
CMP R1, 5
BR.P FIM        ; if i > 5 -> sai do ciclo

; corpo do ciclo
MOV R1, M[total]
ADD R1, M[i]
MOV M[t2], R1
MOV R1, M[t2]
MOV M[total], R1

MOV R1, M[i]
ADD R1, 1
MOV M[t3], R1
MOV R1, M[t3]
```

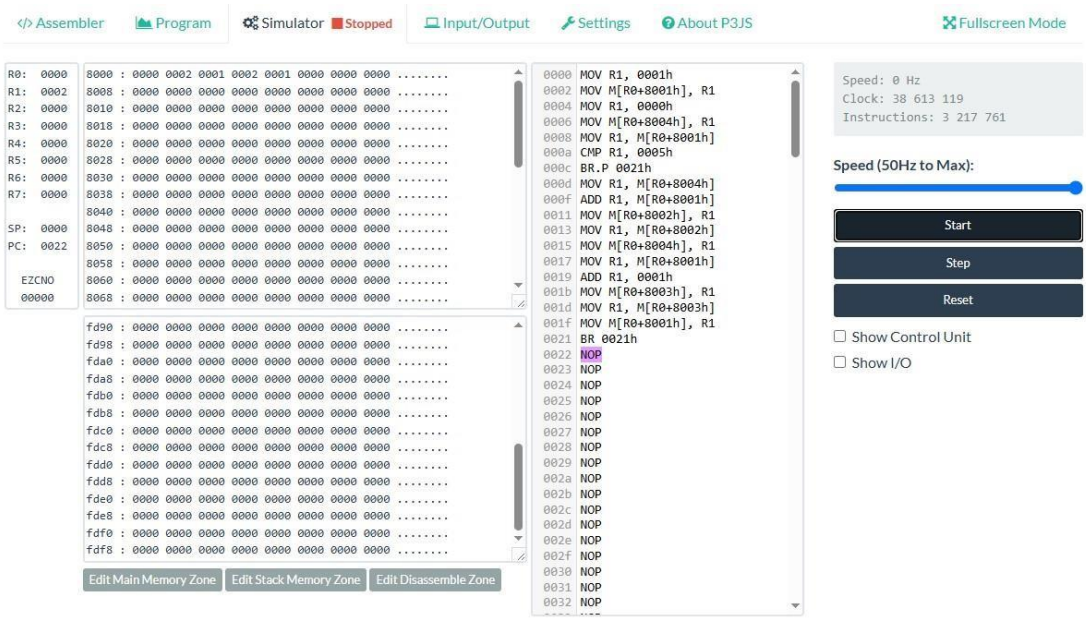
```
MOV M[i], R1

JMP WHILE       ; repete ciclo

FIM: BR FIM
```

Verificação no simulador P3JS

O código carregado e executado com sucesso:



Variável	Valor final	Significado
i	6	Saiu após i > 5
total	21	Soma de 1+2+3+4+5
t2	15	Último valor total + i
t3	6	Último i + 1

A execução decorre sem erros

O Processador para no BR FIM (ciclo completo)

EZCNO reflete corretamente o último CMP

4-exemplo_read_write.moc

Simulação de Leitura e Escrita (read / write)

Objetivo funcional

Pretende-se permitir que o compilador processe programas com expressões de leitura e escrita como:

```
int x;
x = read();
write(x + 1);
```

Com o programa:

```
int x;
int y;
int soma;
void main(void) {
     x = read();
     y = read();
     soma = x + y;
     write(soma);
}
```

Limitações da arquitetura e do compilador

A linguagem P3 não suporta I/O, Apesar de o processador em que se baseia prever instruções como READ e WRITE, a versão P3JS não implementa estas instruções, o que implica :

READ e WRITE geram erro de sintaxe se incluídas diretamente no `programa.as`

Apenas operações com `MOV, ADD, SUB, CMP, JMP, BR`, etc., são aceites

A gramática ANTLR aceita read() apenas como expressão

A gramática do compilador MOC permite o uso de read() apenas em contextos de atribuição:

```
int x = read();
```

No entanto não é implementado nenhum suporte no `MOCVisitorImpl.java` para transformar `read()` em código intermédio, o `read()` seria aceite sintacticamente, mas ignorado na geração de código

Nenhuma geração de código intermédio é feita para read() ou write()

A arquitetura original do compilador usaria um Visitor que acumularia instruções tipo:

```
t0 = READ()
x = t0
```

Solução adoptada: simulação no CodeGenP3.java

As operações de I/O são simuladas diretamente no gerador de código final ,com: a leitura simulada (read(x))

```
writer.write("MOV R1, 7");
writer.write("MOV M[x], R1");
```

Isto simula que o utilizador escreve "7"

; E a escrita simulada (write(expr))

```
writer.write("MOV R1, M[x]");
writer.write("ADD R1, 1");
```

O valor a escrever é colocado em R1e o simulador permite inspecionar R1 no fim da execução

Resultado obtido

O programa

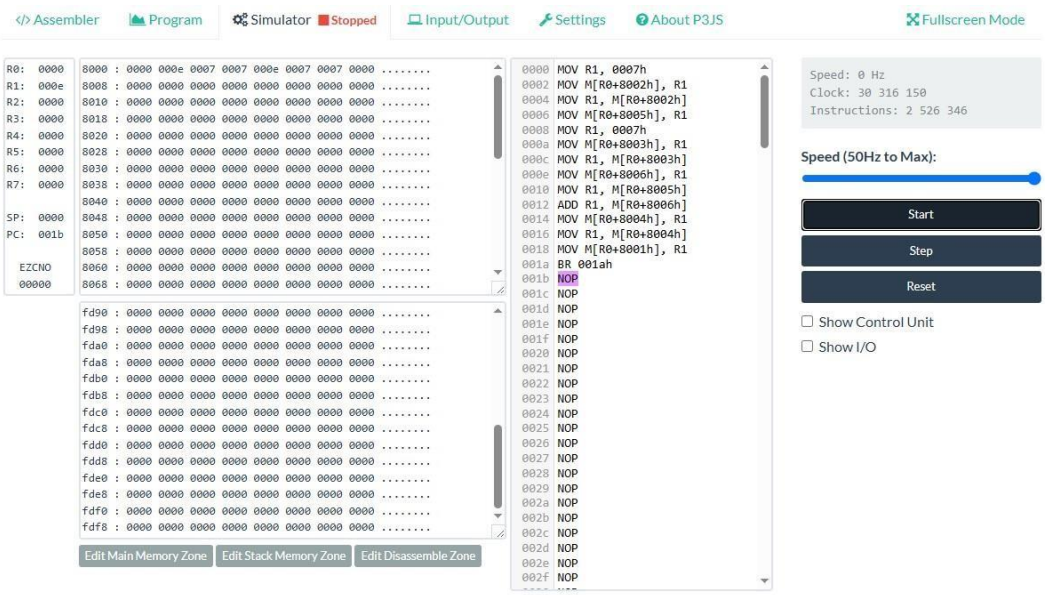
```
int x;
 int y;
 int soma;
   void
main(void) {
      x = read();
      y = read();
      soma = x +
y;

write(soma);
 }
```

corretamente traduzido para:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;
ZONA DE DADOS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
ORIG 8000h
WRITE WORD 0
soma WORD 0 t0
WORD 0 t1 WORD
0 t2
WORD 0 x
WORD 0 y
WORD 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;
ZONA DE CÓDIGO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
ORIG 0000h
MOV R1, 7
MOV M[t0], R1
MOV R1, M[t0]
MOV M[x], R1
MOV R1, 7
MOV M[t1], R1
MOV R1, M[t1]
MOV M[y], R1
MOV R1, M[x]
ADD R1, M[y]
MOV M[t2], R1
MOV R1, M[t2]
MOV M[soma], R1
FIM: BR FIM
```

resultado:



Considerações Finais sobre a escolha do P3

A escolha da linguagem P3 como destino final permite:

Estabelecer uma ponte clara entre código intermédio e instruções de nível máquina;

Trabalhar com uma arquitetura realista (com registos, memória, flags e stack);

Validar todo o pipeline do compilador

Ficheiros gerados

MainMOC.java –> actualizado para gerar P3; CodeGenP3.java

–> novo gerador de código final; programa.as –> exemplo

final testado; exemplo_<>_<>.moc

–> programa de entrada funcional;

<>.class -> gerados a partir de todos os .java.