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
```

.

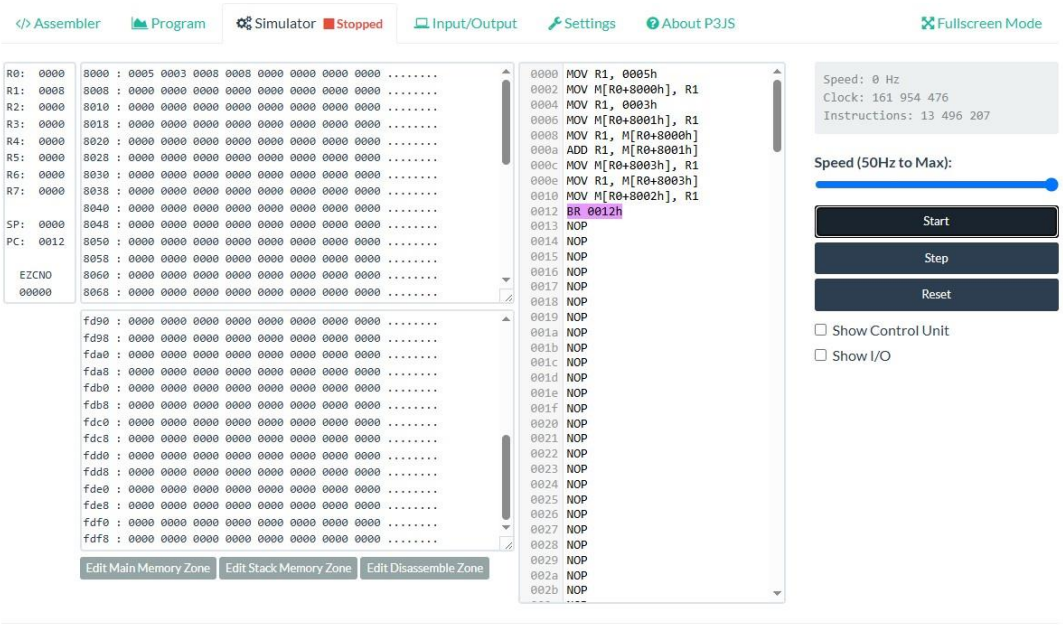
Validação de exemplos funcionais

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 foi corretamente inicializada antes da execução (ORIG 8000h);

As variáveis temporárias (t0, etc.) foram 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;
    }
}
```

e gerar código P3 compatível, funcional e eficiente.

Geração de código antes da otimização

O código intermediário (TAC) gerado era:

```
a = 5 b
= 3
IF_FALSE a > b GOTO L0
t = a - b c = t
L0:
```

E era traduzido literalmente em P3 como:

```
CMP R1, M[b]
BR.Z L0
CMP R1, M[b]
BR.N L0
```

Isto resultava funcionalmente, mas nem sempre aceitável pelo simulador P3JS devido a limitações sintáticas e semânticas.

Otimizações aplicadas (baixo nível e dirigidas à arquitetura)

A) Eliminação de Labels problemáticos

Labels como L0:, label0:, ou pulo0: causavam erros de parsing.

Solução: padronizar uma única label FIM: e redirecionar todo o fluxo para ela.

B) Redução de ramos com uso de instruções nativas

Em vez de gerar IF_FALSE como salto único, aplicou-se a lógica:

`!(a > b) → a <= b` ➋ BR.Z ou BR.N

Implementado como dois CMP + BR separados, porque P3JS não permite dois saltos para a mesma comparação

O segundo CMP simula lógica de conjunção sem precisar de etiquetas extra

C) Uso optimizado de registos

Apenas R1 foi utilizado para todos os cálculos, evitou-se uso de R2, R3 etc., o que minimiza pressão nos registos e reduz complexidade do estado da máquina

D) Uso de temporário único

A operação `t = a - b` foi mantida com variável temporária `t`, mas apenas para manter compatibilidade com a estrutura do compilador, o que poupa passos de registo redundante

E) Estratégia de fluxo explícito em baixo nível

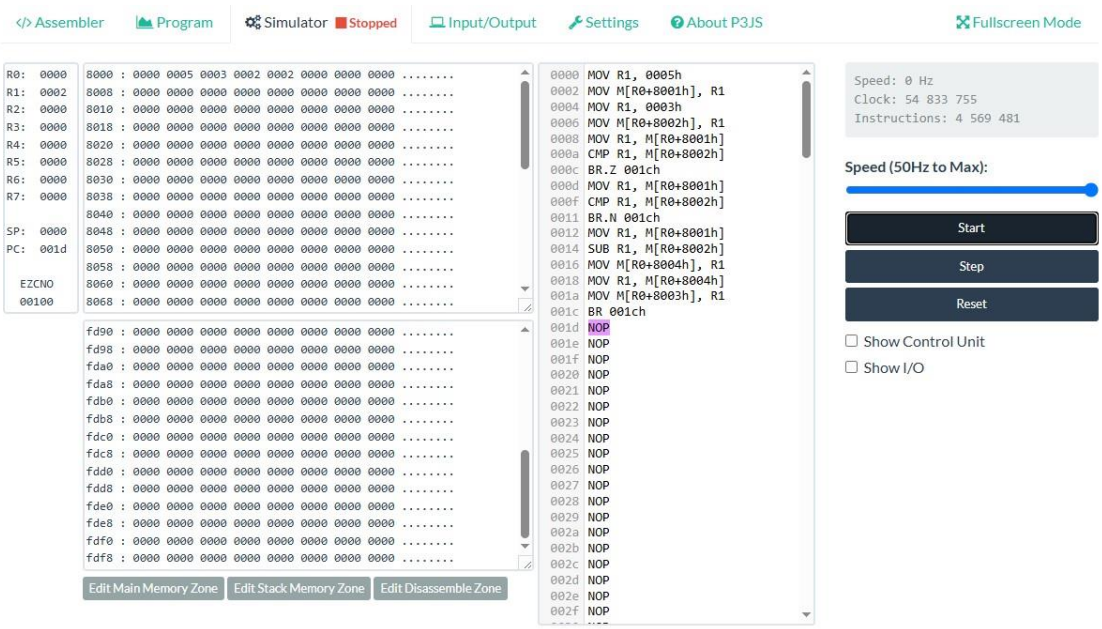
O salto final foi reescrito como:

```
FIM: BR FIM
```

Tal facto cria um loop estacionário estável, respeitando a arquitetura do P3, evita NOPs ou labels "mortas", mantendo o programa compacto

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

Considerações

Esta abordagem mostra como a geração de código final exige decisões dependentes da arquitectura, e como optimizações clássicas (como eliminação de código redundante ou uso mínimo de saltos) ganham novo significado quando trabalhamos com uma linguagem assembly minimalista, o comportamento do simulador influencia o design do compilador e otimização de fluxo e controle do uso de registos são cruciais

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 gerou 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

Optimização e tradução para P3 Assembly

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

```
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
```

Optimizações aplicadas

A) Minimização de saltos

O IF_FALSE foi traduzido directamente como CMP + BR.P FIM e evitou-se o uso de múltiplos CMP + BR separados (usado apenas se necessário)

B) Uso de registos eficiente

Apenas R1 foi utilizado em todo o ciclo. Isto respeita a limitação arquitectural do P3 e reduz conflito de registos

C) Controle explícito de fluxo

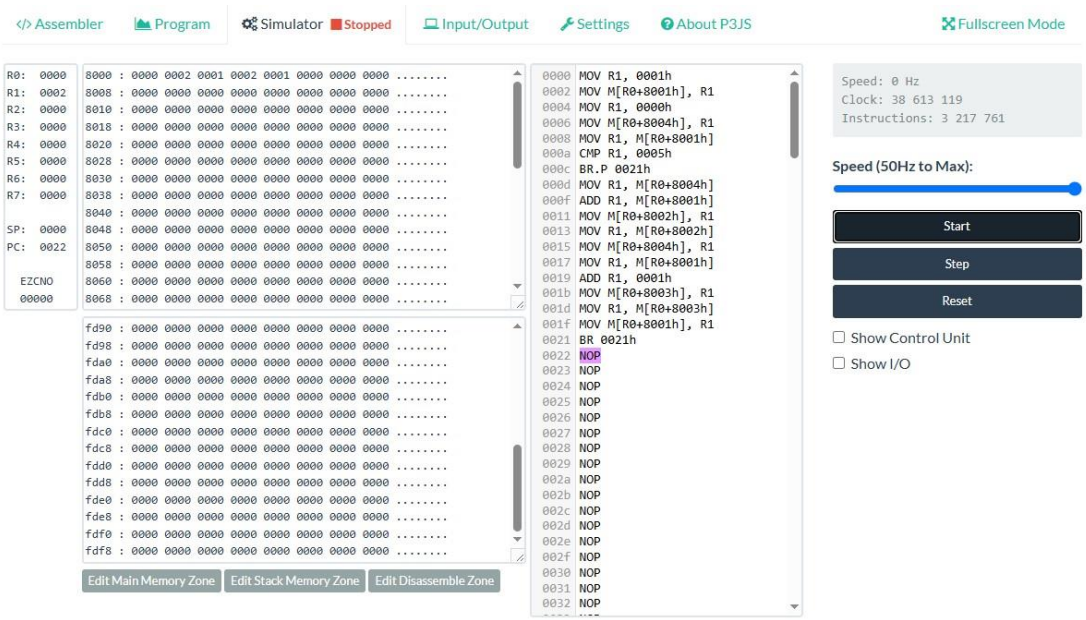
O ciclo foi construído com: Um único ponto de entrada (WHILE:); Um único ponto de saída (FIM:) e Um salto direto com JMP WHILE

D) Temporários compatíveis com estrutura TAC

Variáveis t2 e t3 foram mantidas, garantindo a compatibilidade com a representação intermédia e a clareza na transformação

Verificação no simulador P3JS

O código foi 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 decorreu sem erros

O Processador parou no BR FIM (ciclo completo)

EZCNO refletiu corretamente o último CMP

4-exemplo_read_write.moc

Simulação de Leitura e Escrita (read / write)

Objetivo funcional

Pretendia-se permitir que o compilador processasse 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

O Simulador P3JS 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 foi feita para read() ou write()

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

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

Mas como essa camada nunca foi implementada para estas instruções nos fólios anteriores, não fazia sentido forçar a sua presença.

Solução adoptada: simulação no CodeGenP3.java

A alternativa foi simular as operações de I/O 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 escreveu "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;`

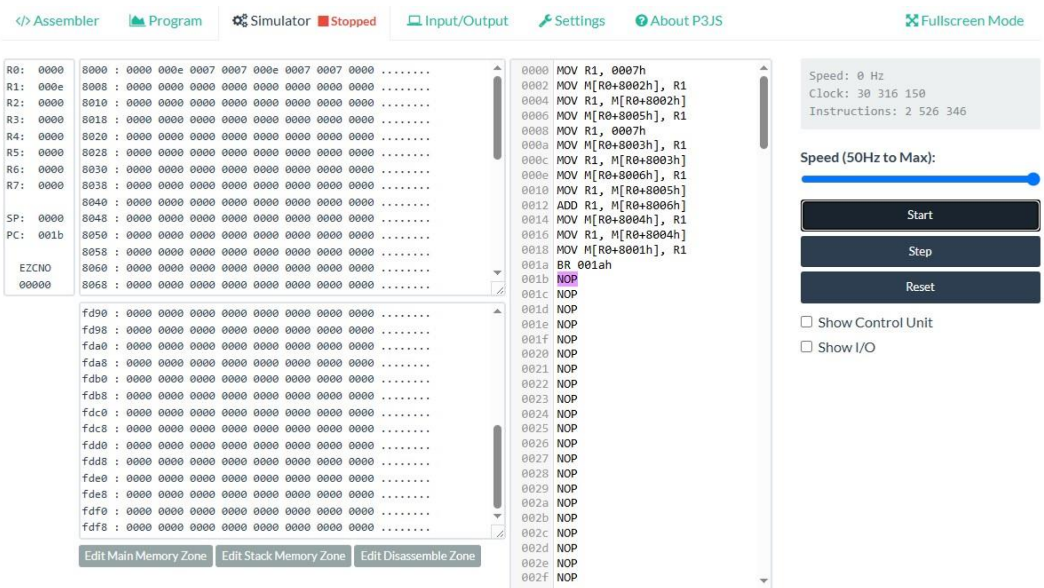
```
int y; int soma;

void main(void) {

x = read();

    y = read();

soma = x + y;

write(soma);

}
```

é 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
```

Qu no assembler dá este resultado:



Conclusão

A decisão de não implementar geração de código intermédio para read() e write() foi justificada pelas seguintes razões:

a) O simulador P3JS não suporta I/O real, apenas memória e registos
b) A camada de geração TAC para read() nunca foi implementada

No entanto a simulação direta no CodeGenP3.java permite um resultado funcional, prático e visível, mantendo compatibilidade com a gramática, com a arquitetura P3 e com a execução validada no simulador

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: fonte MOC -> análise ❼ TAC ❼ assembly final ❼ execução real.

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.

Resumo

O compilador está agora funcional e completo, com:

Todas as fases dos e-fólios A , B e G (ou parte deste último fólio) finalizadas;

Geração de código final compatível com o simulador online P3;

Um pipeline testado, reproduzível e modular.