



Universidade do Minho  
Escola de Engenharia

Tiago Manuel Martins Vasconcelos

Criação de ferramentas de desenvolvimento  
para uma arquitetura baseada em Microblaze





Universidade do Minho  
Escola de Engenharia

Tiago Manuel Martins Vasconcelos

Criação de ferramentas de desenvolvimento  
para uma arquitetura baseada em Microblaze

Dissertação de Mestrado  
Ciclo de Estudos Integrados Conducentes ao Grau de  
Mestre em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do  
Professor Doutor Adriano Tavares

## DECLARAÇÃO

Nome: Tiago Manuel Martins Vasconcelos

Correio electrónico: tiago.vasconcelos.18@gmail.com

Tel./Tlm.: 915557283

Número do Bilhete de Identidade:13743876

Título da dissertação:

**Criação de ferramentas de desenvolvimento para uma arquitetura baseada em *Microblaze***

Ano de conclusão: 2014

Orientador:

Professor Doutor Adriano Tavares

Designação do Mestrado:

Ciclo de Estudos Integrados Conducentes ao Grau de Mestre em Engenharia Electrónica Industrial e Computadores

Área de Especialização: Sistemas Embebidos

Escola de Engenharia

Departamento de Electrónica industrial

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA DISSERTAÇÃO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE.

Guimarães, \_\_/\_\_/\_\_\_\_

Assinatura: \_\_\_\_\_

## AGRADECIMENTOS

---

Aa primeiras palavras de agradecimento são direcionadas à minha família. Em especial aos meus pais João de Sousa Vasconcelos e Maria Paula Viana Martins, por todo o apoio educacional, psicológico e financeiro prestado durante todo o meu percurso académico.

Ao meu orientador Doutor Adriano Tavares pelo apoio prestado e pela confiança que depositou em mim mesmo quando eu próprio questioneei a minha capacidade de levar a presente dissertação até ao fim.

Ao aluno de doutoramento Paulo Garcia pela disponibilidade para uma revisão cuidada da minha dissertação e pelos ensinamentos transmitidos durante o processo.

Aos “voluntários” Pedro Matos, Rui Machado e Vanessa Cunha pelo tempo gasto a rever o meu documento nas fases iniciais da escrita.

Ao parceiro de projeto João Martins que desenvolveu o processador alvo, pela disponibilidade e pela fácil comunicação que foram essenciais para o sucesso das duas dissertações.

Aos colegas de laboratório Ana Isabel Pedregal, Davide Guimarães, Eduardo Domingues, Eurico Moreira, Filipe Alves e Vasco Lima pela entreaajuda, companheirismo e momentos de descontração essenciais a um bom ambiente de laboratório.

Aos colegas de curso que me acompanharam na fase inicial mas que seguiram outros caminhos na sua formação, em especial ao Hugo Gomes, José Oliveira, Nelson Pereira, Pedro Alves e Rui Rodrigues.

Finalmente, e não menos importante, à minha namorada Matilde Azevedo por me conseguir aturar e até animar nos momentos difíceis durante a realização do trabalho.

A todos, MUITO OBRIGADO!!!



## RESUMO

---

Nos tempos atuais as empresas vivem numa constante corrida para chegar em primeiro ao mercado, para tentar obter uma vantagem em relação à concorrência. Em empresas que produzem soluções de sistemas embebidos a realidade não é diferente, assim as empresas necessitam de ferramentas para agilizar o desenvolvimento de produtos. Quando são utilizados microprocessadores, é obrigatório a existência de diversas ferramentas de desenvolvimento para a plataforma alvo, como compilador, *linker*, simuladores, etc.

Com a crescente complexidade de aplicações que os sistemas embebidos executam, na maioria das vezes é imprescindível a utilização de sistemas operativos no desenvolvimento da aplicação. Com o tempo os sistemas operativos evoluem e por vezes é desejável a migração da aplicação para um sistema operativo mais atual. Esta mudança pode ser morosa, pois envolve a alteração do código fonte da aplicação, nomeadamente, as chamadas ao sistema operativo. Muitas vezes estas alterações não são efetuadas pelo programador que desenvolveu a aplicação inicialmente, assim requer que o programador para além de dominar o novo sistema operativo também necessite de dominar o antigo.

O propósito desta dissertação é criar as ferramentas de desenvolvimento para um novo processador que está a ser desenvolvido no âmbito de outra dissertação de mestrado. Este trabalho é necessário para que se possa desenvolver aplicações para o novo processador numa linguagem de programação de alto nível abstraindo o programador da linguagem máquina de baixo nível, facilitando e agilizando o processo de desenvolvimento. Para além disso, introduzir mecanismos de automatização na migração de aplicações entre sistemas operativos.

Palavras-chave: Compilador, Ferramentas de desenvolvimento, agnosticismo





## ABSTRACT

---

Nowadays companies live in a constant race to reach the market first, trying to get an advantage over the competition. In companies that manufacture solutions for embedded systems, the reality is not different, so they need tools to streamline product development. When microprocessors are used, the existence of several development tools for the target platform, such as compilers, linkers, simulators, etc, is mandatory.

With the increasing complexity of applications that run on embedded systems, the use of operating systems on application development is almost always essential. Over time operating systems evolve, and it is sometimes desirable to migrate the application to a newer operating system. This change can be time consuming because it involves changing the application's source code, including calls to the operating system. Often, these changes are not made by the programmer who developed the first application, so the programmer is required to master the new operating system and also needs to comprehend the former.

The purpose of this dissertation is to create development tools for a new processor that is being developed under another dissertation. This work is necessary so that applications for the new processor can be developed in a high level programming language abstracting the programmer from low-level machine language, facilitating and streamlining the development process. Additionally, to introduce mechanisms to automate the migration of applications across operating systems.

Keywords: Compiler, Development tools, agnostic



# ÍNDICE GERAL

---

<b>Agradecimentos</b> .....	<b>v</b>
<b>Resumo</b> .....	<b>vii</b>
<b>Abstract</b> .....	<b>ix</b>
<b>Índice Geral</b> .....	<b>xi</b>
<b>Abreviaturas e siglas</b> .....	<b>xiii</b>
<b>Índice de Figuras</b> .....	<b>xv</b>
<b>Índice de Tabelas</b> .....	<b>xix</b>
<b>Introdução</b> .....	<b>1</b>
1.1. <i>Contextualização do trabalho</i> .....	1
1.2. <i>Motivação e objetivos</i> .....	2
1.3. <i>Organização do documento</i> .....	3
<b>Estado da Arte</b> .....	<b>5</b>
2.1. <i>Compilador</i> .....	5
2.1.1. ANTRL (ANother Tool for Language Recognition) .....	6
2.1.2. Flex – A fast scanner generator .....	7
2.1.3. Bison .....	8
2.2. <i>Linguagem C</i> .....	9
2.2.1. GCC .....	11
2.2.2. Tiny C Compiler .....	12
2.3. <i>Sistemas operativos</i> .....	12
2.4. <i>Portabilidade e Agnosticismo</i> .....	13
2.5. <i>Arquitetura alvo</i> .....	13
2.6. <i>Conclusões</i> .....	14
<b>Especificação do sistema</b> .....	<b>15</b>
3.1. <i>Funcionalidades e Restrições</i> .....	15
3.2. <i>Compilador</i> .....	16
3.2.1. <i>Front end</i> .....	16
3.2.2. <i>Representação intermédia</i> .....	21
3.2.3. <i>Back end</i> .....	24
3.2.4. <i>ABI (Application Binary Interface)</i> .....	26
3.3. <i>Microkernel</i> .....	29
3.4. <i>Agnosticismo</i> .....	30
<b>Implementação</b> .....	<b>33</b>

4.1.	<i>Compilador</i> .....	33
4.1.1.	<i>Front end</i> .....	33
4.1.2.	Representação intermédia .....	36
4.1.3.	<i>Back End</i> .....	40
4.2.	<i>Microkernel</i> .....	53
4.2.1.	Tarefas .....	53
4.2.2.	Escalonador .....	54
4.2.3.	Comutação de contexto .....	55
4.3.	<i>Agnosticismo</i> .....	56
<b>Resultados</b> .....		<b>59</b>
5.1.	<i>Disassembler</i> .....	59
5.2.	<i>Plataformas de teste</i> .....	60
5.2.1.	Simulador .....	60
5.2.2.	Plataforma alvo .....	62
5.3.	<i>Testes</i> .....	63
5.3.1.	Validação das funcionalidades do compilador .....	63
5.4.	<i>Testes ao microkernel</i> .....	69
5.5.	<i>Teste à componente agnóstica do compilador</i> .....	70
<b>Conclusões e trabalho futuro</b> .....		<b>75</b>
6.1.	<i>Conclusões</i> .....	75
6.2.	<i>Trabalho Futuro</i> .....	76
<b>Referências bibliográficas</b> .....		<b>77</b>
<b>Anexos</b> .....		<b>79</b>

## ABREVIATURAS E SIGLAS

---

ABI	<i>Application Binary Interface</i>
ANSI	<i>American National Standards Institute</i>
ANTRL	<i>ANother Tool for Language Recognition</i>
API	<i>Application Programming Interface</i>
CPU	<i>Central Process Unit</i>
EOF	<i>End of File</i>
FPGA	<i>Field Programmable Gate Array</i>
GCC	<i>GNU compiler collection</i>
Id	<i>Identifier</i>
IEEE	<i>Institute of Electrical and Electronics Engineers</i>
ISA	<i>Instruction Set Architecture</i>
ISR	<i>Interrupt Service Routine</i>
OP	<i>Operation</i>
PC	<i>Program Counter</i>
RAM	<i>Random Access Memory</i>
RISC	<i>Reduced Instruction Set Computing</i>
RTOS	<i>Real Time Operating System</i>
SoC	<i>System on Chip</i>
SP	<i>Stack Pointer</i>
TCB	<i>Task Control Block</i>
TCC	<i>Tiny C Compiler</i>
UART	<i>Universal Asynchronous Receiver/Transmitter</i>
XML	<i>eXtensible Markup Language</i>



# ÍNDICE DE FIGURAS

---

FIGURA 2.1 - COMPILADOR VISTO COMO UMA CAIXA NEGRA .....	5
FIGURA 2.2-TEMPLATE APRESENTADA NO LIVRO [6] PARA O DESENVOLVIMENTO DE COMPILADORES.....	9
FIGURA 3.1-DIAGRAMA DE BLOCOS DO FRONT END .....	17
FIGURA 3.2 - FLUXOGRAMA DE VALIDAÇÃO DA ESTRUTURA DE CONTROLO DE FLUXO WHILE .....	19
FIGURA 3.3-DIAGRAMA DE BLOCOS DA INTERAÇÃO DO <i>FRONT END</i> COM A REPRESENTAÇÃO INTERMÉDIA .....	21
FIGURA 3.4-REPRESENTAÇÃO DA <i>STACK</i> DE TABELAS DE SIMBOLOS.....	22
FIGURA 3.5-ASPECTO DE UMA TABELA DE SÍMBOLOS ORDENADA ALFABETICAMENTE .....	22
FIGURA 3.6-ESTRUTURA E RELAÇÕES ENTRE AS CLASSES QUE GUARDA A INFORMAÇÃO DE CADA SÍMBOLO .....	23
FIGURA 3.7-EXEMPLO DA ESTRUTURA DO CÓDIGO INTERMÉDIO .....	24
FIGURA 3.8-DIAGRAMA DE BLOCOS <i>BACK END</i> EM INTERAÇÃO COM A REPRESENTAÇÃO INTERMÉDIA .....	25
FIGURA 3.9 - CLASSE QUE GERE A UTILIZAÇÃO DOS REGISTOS DA MÁQUINA .....	25
FIGURA 3.10 - <i>STACK</i> QUE AUXILIA A RESOLUÇÃO DE EXPRESSÕES MATEMÁTICAS.....	26
FIGURA 3.11 - ÁREAS DE MEMÓRIA USADAS DURANTE A EXECUÇÃO DE UM PROGRAMA .....	27
FIGURA 3.12 - ESTRUTURA DE UMA <i>FRAME</i> NA <i>STACK</i> .....	28
FIGURA 3.13 - DIAGRAMA DE ESTADOS DAS TAREFAS DO <i>MICROKERNEL</i> .....	29
FIGURA 3.14 - FASES DE COMPILAÇÃO DEPOIS DE INTRODUIDA COMPONENTE AGNÓSTICA DO SISTEMA OPERATIVO .....	31
FIGURA 4.1 - ATRIBUIÇÃO DE TIPO A CADA CARATERE DA TABELA ASCII.....	34
FIGURA 4.2 - MÉTODO PARA CONSTRUÇÃO DO <i>TOKEN</i> .....	35
FIGURA 4.3 - MÉTODO QUE DIFERENCIA O TIPO DE <i>STATEMENT</i> .....	36
FIGURA 4.4 - ATRIBUTOS DA CLASSE <i>TSYMTABNODE</i> .....	37
FIGURA 4.5 - MÉTODOS DE INTRODUÇÃO DE <i>TOKENS</i> NA ESTRUTURA DO CÓDIGO INTERMÉDIO.....	38
FIGURA 4.6 - MÉTODOS DE OBTENÇÃO DE <i>TOKENS</i> A PARTIR DO CÓDIGO INTERMÉDIO .....	39
FIGURA 4.7 - DECLARAÇÃO DA CLASSE <i>TREGISTER</i> .....	40
FIGURA 4.8 - DECLARAÇÃO DA CLASSE <i>TREGISTERFILE</i> .....	41
FIGURA 4.9 - PONTO DE ENTRADA PARA GERAÇÃO DO PROGRAMA.....	41
FIGURA 4.10 - GERAÇÃO DO CÓDIGO DO PROGRAMA FONTE .....	42
FIGURA 4.11 - BLOCO DE CÓDIGO QUE CRIA O FICHEIRO FINAL COM O CÓDIGO MÁQUINA.....	42
FIGURA 4.12 - MÉTODO QUE IDENTIFICA O TIPO DE <i>STATEMENT</i> A GERAR .....	44
FIGURA 4.13 – DIAGRAMAS SINTÁTICOS QUE RESOLVEM AS EXPRESSÕES LÓGICAS E ARITMÉTICAS NO COMPILADOR. ....	46
FIGURA 4.14 - EXEMPLO DA RESOLUÇÃO DE UMA EXPRESSÃO ARITMÉTICA .....	46
FIGURA 4.15 - ALGORITMO PARA RESOLVER UMA MULTIPLICAÇÃO À CUSTA DE VÁRIAS SOMAS .....	48
FIGURA 4.16 – REPRESENTAÇÃO DOS CAMPOS QUE COMPÕES O NÚMERO. ....	49
FIGURA 4.17 - CONFIGURAÇÃO DO NÚMERO SEGUINDO A NORMA DO <i>STANDARD</i> IEEE 754 .....	50
FIGURA 4.18 – FÓRMULA MATEMÁTICA PARA RESOLUÇÃO DE EXPRESSÕES ARITMÉTICAS ENVOLVENDO NÚMEROS COM VÍRGULA FLUTUANTE.....	50

FIGURA 4.19 - API'S DE MANIPULAÇÃO DOS PERIFÉRICOS .....	51
FIGURA 4.20 - CÓDIGO GERADO A PARTIR DE UMA FUNÇÃO DE SERVIÇO À INTERRUPÇÃO .....	51
FIGURA 4.21 - TCB DO <i>MICROKERNEL</i> .....	53
FIGURA 4.22 - CÓDIGO QUE IMPLEMENTA O ALGORITMO DE ESCALONAMENTO BASEADO EM PRIORIDADES .....	55
FIGURA 4.23 - PSEUDO CÓDIGO DA ISR DO <i>SYSTICK</i> ONDE OCORRE COMUTAÇÃO DE CONTEXTO .....	55
FIGURA 4.24 - ESQUEMA DO REPOSITÓRIO DO AGNÓSTICO .....	56
FIGURA 4.25 - ASPETO DO FICHEIRO <i>GENERICAPI.DATA</i> .....	57
FIGURA 4.26 - ASPETO DO FICHEIRO <i>.INTERFACE</i> DE UM SISTEMA OPERATIVO ALVO.....	57
FIGURA 5.1 – PROCESSO DE OBTENÇÃO DO CÓDIGO <i>ASSEMBLY</i> A PARTIR DO CÓDIGO C.....	60
FIGURA 5.2 - <i>LAYOUT</i> INICIAL DO SIMULADOR .....	61
FIGURA 5.3 - <i>LAYOUT</i> FINAL DO SIMULADOR.....	62
FIGURA 5.4 - PLATAFORMA ONDE SERÁ ALOJADO O <i>SOFTCORE</i> .....	62
FIGURA 5.5 - TESTE DE ENTRADA NA FUNÇÃO <i>MAIN</i> E EXPRESSÕES SIMPLES .....	64
FIGURA 5.6 - TESTE DE CHAMADAS DE FUNÇÕES.....	66
FIGURA 5.7 - IMPLEMENTAÇÃO DO ALGORITMO <i>BUBBLE SORT</i> E RESULTADO APÓS EXECUÇÃO .....	67
FIGURA 5.8 - CÓDIGO C COM INTERRUPÇÃO EXTERNA .....	68
FIGURA 5.9 - PERIFÉRICO DE <i>OUTPUT</i> ANTES E APÓS A OCORRÊNCIA DA INTERRUPÇÃO EXTERNA.....	68
FIGURA 5.10 - CÓDIGO EXEMPLO DE UMA APLICAÇÃO DE TESTE DO <i>MICROKERNEL</i> .....	69
FIGURA 5.11 - RESULTADO DA APLICAÇÃO APRESENTADA NA FIGURA 5.10 .....	70
FIGURA 5.12 - COMPARAÇÃO DAS API'S DAS DUAS VERSÕES DO <i>MICROKERNEL</i> .....	70
FIGURA 5.13 - COMPARAÇÃO DOS ALGORITMOS DE ESCALONAMENTO DAS DUAS VERSÕES DO <i>MICROKERNEL</i> .....	71
FIGURA 5.14 - APLICAÇÃO UTILIZANDO AS API'S GENÉRICAS .....	72
FIGURA 5.15 - COMPILAÇÃO PARA <i>MICROKERNEL</i> ALVO .....	72
FIGURA 5.16 - FICHEIROS INTERMÉDIOS GERADOS.....	73
FIGURA 5.17 - RESULTADO DOS DOIS SISTEMAS OPERATIVOS A EXECUTAR A MESMA APLICAÇÃO .....	73
FIGURA 6.1 – DIAGRAMA DE CLASSES DO <i>SCANNER</i> .....	82
FIGURA 6.2 – FLUXOGRAMA DE DECLARAÇÃO DE VARIÁVEIS .....	83
FIGURA 6.3 – FLUXOGRAMA DE DECLARAÇÃO DE FUNÇÕES.....	84
FIGURA 6.4 – FLUXOGRAMA DE DECLARAÇÃO DE ESTRUTURAS.....	85
FIGURA 6.5 – DIAGRAMA DE CLASSES DA CLASSE <i>TPARSER</i> .....	86
FIGURA 6.6 – ALGORITMO PARA OBTER A DIVISÃO À CUSTA DE SUBTRAÇÕES.....	87
FIGURA 6.7 – ALGORITMO PARA OBTER O RESTO DA DIVISÃO À CUSTA DE SUBTRAÇÕES.....	87
FIGURA 6.8 - BIBLIOTECA <i>ASSEMBLY</i> PARA SOMAS E SUBTRAÇÕES DE <i>FLOATS</i> .....	89
FIGURA 6.9 - BIBLIOTECA <i>ASSEMBLY</i> PARA MULTIPLICAÇÃO DE <i>FLOATS</i> .....	90
FIGURA 6.10 - BIBLIOTECA <i>ASSEMBLY</i> PARA DIVISÃO DE <i>FLOATS</i> .....	91
FIGURA 6.11 – CÓDIGO PARA RESTAURAR O CONTEXTO DE UMA TAREFA .....	92
FIGURA 6.12 – CÓDIGO PARA SALVAR O CONTEXTO DE UMA TAREFA.....	93
FIGURA 6.13 – CÓDIGO DA INTERRUPÇÃO DO <i>SYSTICK</i> .....	94



FIGURA 6.14 – CÓDIGO PARA CONFIGURAR O PERIFÉRICO <i>SYSTICK</i> E HABILITAR A SUA INTERRUPÇÃO NO <i>DVIC</i> .....	94
FIGURA 6.15 – FUNÇÃO PARA OBTER O ENDEREÇO DE SALTO DA PRIMEIRA EXECUÇÃO DE UMA TAREFA.....	95
FIGURA 6.16 – FUNÇÃO QUE ESCALONA AS TAREFAS.....	95
FIGURA 6.17 – FUNÇÃO QUE FAZ O <i>UPDATE</i> DA INFORMAÇÃO DAS TAREFAS QUE ESTÃO NO ESTADO DE <i>DELAY</i> .....	95
FIGURA 6.18 – FUNÇÃO QUE CRIA UMA <i>STACK</i> VIRTUAL PARA A PRIMEIRA EXECUÇÃO DE UMA TAREFA.....	96
FIGURA 6.19 – FUNÇÃO QUE INICIALIZA O ESCALONADOR E CEDE O PROCESSADOR À PRIMEIRA TAREFA A EXECUTAR .....	96
FIGURA 6.20 – API PARA CRIAR UMA NOVA TAREFA .....	97
FIGURA 6.21 – API PARA COLOCAR UMA TAREFA NO ESTADO DE <i>DELAY</i> .....	97
FIGURA 6.22 – API PARA COLOCAR UMA TAREFA NO ESTADO DE <i>SUSPEND</i> .....	97



## ÍNDICE DE TABELAS

---

TABELA 1 - REGISTOS DEFINIDOS NO ABI .....	27
TABELA 2 - TAMANHO DOS TIPOS DE DADOS PRIMITIVOS .....	29
TABELA 3 - PRIORIDADES DOS OPERADORES.....	47
TABELA 4 - LISTA DE INTERRUPÇÕES SUPORTADAS PELO PERIFÉRICO DVIC .....	52
TABELA 5 - INSTRUÇÕES LÓGICAS E ARITMÉTICAS .....	79
TABELA 6 - INSTRUÇÕES DE SALTO E SALTO CONDICIONAL.....	79
TABELA 7 - INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS .....	80
TABELA 8 - OUTRAS INSTRUÇÕES.....	81



# Capítulo 1

## INTRODUÇÃO

---

Neste capítulo é feita uma contextualização do trabalho desenvolvido na presente dissertação e enumerados os objetivos a atingir e a motivação para os alcançar. Por fim é apresentada a estrutura organizacional do presente documento.

### 1.1. Contextualização do trabalho

Nos tempos atuais existe um elevado número de sistemas computacionais que fazem parte do nosso dia-a-dia. Estes sistemas desempenham diversas tarefas, umas triviais e outras bastante complexas. Com a crescente complexidade das tarefas, cresce também a complexidade da programação destes sistemas. Assim são necessárias ferramentas para agilizar o desenvolvimento de aplicações para estes sistemas, poupando tempo e esforço ao programador, aumentando a sua produtividade.

Para agilizar o desenvolvimento de aplicações para o microprocessador usa-se um compilador de uma linguagem de alto nível, para que este converta um programa escrito numa linguagem de programação de alto nível (linguagem orientada à compreensão humana) para código máquina interpretável pelo microprocessador.

Com o aumento da complexidade das aplicações e com o facto de nos tempos atuais estas serem encarregues de executar várias atividades em simultâneo, o uso de sistemas operativos para explorar o paralelismo das atividades torna-se crucial.

Existem diversos requisitos de aplicações que exigem diferentes serviços dos sistemas operativos. Com esta variedade de necessidades e com a variedade de sistemas operativos disponíveis, é frequente a necessidade de transportar uma aplicação para outro sistema operativo. Esta mudança de sistema operativo implica uma reestruturação do código desenvolvido, nomeadamente a substituição de todas as chamadas ao sistema operativo por parte da aplicação que se pretende migrar. A necessidade de realizar o *porting* da aplicação entre sistemas operativos implica um esforço de desenvolvimento adicional para o programador. Esta problemática abre assim a possibilidade ao aparecimento de ferramentas que realizem este *porting* de forma rápida e automática, sem a intervenção do programador.

No grupo de sistemas embebidos do departamento de Engenharia Electrónica Industrial da Universidade do Minho está a ser desenvolvido um novo microprocessador, no âmbito de outra dissertação de mestrado, composto por um *pipeline* de cinco estágios, conectado a um controlador de interrupções com prioridades programáveis e fontes de interrupção externas e por *software*, um bloco de *timers* programáveis em termos de prioridades, contagem e frequência, um periférico PARIO com pinos digitais de *input* e *output* para uso de propósito geral e uma UART que garante ao microprocessador a capacidade de comunicação série com outros dispositivos.

## 1.2. Motivação e objetivos

Desde os princípios da computação se percebeu que não era viável programar em linguagem máquina. Com o intuito de resolver este problema foram criadas linguagens mais orientadas à percepção humana e programas de computador que traduzem aplicações escritas numa linguagem de alto nível perceptível por humanos, para código máquina interpretável pelo processador. Este programa de tradução é chamado de compilador.

Com o aparecimento de novos processadores é necessário o desenvolvimento de um novo compilador ou adaptar o *backend* de um existente. O processador a ser desenvolvido no âmbito de outra dissertação de mestrado, necessita das respetivas ferramentas de desenvolvimento, para facilitar a criação de aplicações.

Com esta dissertação pretende-se desenvolver um compilador de linguagem C para tornar possível o desenvolvimento de aplicações para a nova arquitetura. Após o seu desenvolvimento será criado um pequeno *microkernel*, compilável no compilador produzido na fase anterior. Este *microkernel* simplista deverá implementar um conjunto mínimo de recursos que compreendem um sistema operativo embebido. Quando estes dois objetivos forem concluídos com sucesso será desenvolvida e integrada no compilador uma ferramenta capaz de migrar de forma automática aplicações entre sistemas operativos ou *microkernel*. A resultante deste trabalho será um ecossistema de ferramentas que garantirão suporte e agilização no processo de desenvolvimento de aplicações para a nova arquitetura que será desenvolvida em paralelo com a presente dissertação de mestrado.

Compiladores e sistemas operativos são a base dos sistemas informáticos atuais, assim o seu estudo e compreensão permitem o entendimento das bases sobre as quais se assentam os sistemas informáticos. Estas temáticas são as áreas de interesse do autor no âmbito dos

sistemas informáticos. O estudo destas áreas de conhecimento origina desafios interessantes e motivadores.

### **1.3. Organização do documento**

O presente documento encontra-se dividido em seis capítulos: introdução, estado da arte, especificação do sistema, implementação, resultados e conclusões.

No capítulo 1 é feita uma introdução e contextualização do trabalho, e são enumerados os objetivos estipulados para a presente dissertação e a estrutura da mesma.

No capítulo 2 é retratado o estado atual das tecnologias que suportam o desenvolvimento do presente trabalho. O capítulo é introduzido com uma descrição das vantagens do uso de compiladores no processo de desenvolvimento de *software*. De seguida explica-se ferramentas e metodologias que auxiliam no desenvolvimento de compiladores. E por fim é feita uma pequena descrição da arquitetura alvo do presente trabalho.

No capítulo 3 são abordadas as especificações do sistema a implementar. O capítulo está dividido em três tópicos, no primeiro tópico são apresentadas as especificações do compilador e as diversas partes que o compõem, no segundo tópico são apresentados os requisitos e as especificações para o *microkernel* a desenvolver, por fim, no terceiro tópico são apresentadas as especificações para a ferramenta de agnosticismo ao sistema operativo a incorporar no compilador.

No capítulo 4 é apresentada a implementação do sistema proposto. Aqui são apresentados os diversos passos tomados na construção do ecossistema de ferramentas. O capítulo à semelhança do anterior está dividido em três subcapítulos, em cada subcapítulo é explicado o desenvolvimento de uma ferramenta do ecossistema.

No capítulo 5 são apresentados os testes realizados e os resultados obtidos a todas as ferramentas desenvolvidas. Numa fase inicial são apresentados testes ao compilador por forma a validar o seu correto funcionamento, exigido pelas ferramentas. Depois de validado o compilador foi a vez do *microkernel*, cujo correto funcionamento é mais uma vez crucial para o desenvolvimento da ferramenta de agnosticismo ao sistema operativo a incorporar no compilador. Ao testar a ferramenta de agnosticismo, esta funciona também como teste de integração de todo o ecossistema de ferramentas.

No capítulo 6 são apresentadas as conclusões desta dissertação e sugestões para trabalho futuro por forma a melhorar o ecossistema de ferramentas desenvolvido.

No final do documento é apresentada a bibliografia e os anexos com informação relevante à compreensão do presente trabalho.



## Capítulo 2

### ESTADO DA ARTE

---

#### 2.1. Compilador

Os compiladores são uma das bases da computação moderna, uma vez que são as ferramentas que permitem a tradução de um programa escrito numa linguagem dita de alto nível (linguagem orientada à compreensão humana) para um programa escrito numa linguagem de baixo nível (linguagem orientada à compreensão da máquina). O uso de uma linguagem de alto nível para o desenvolvimento de aplicações é de extrema utilidade, pois como esta linguagem não é dependente da máquina, promove a reutilização de código em aplicações futuras. Esta característica permite deste modo diminuir o tempo e custos necessários ao desenvolvimento de uma aplicação, aumentando assim a produtividade no desenvolvimento de *software*.

Para o programador, o compilador pode ser visto como uma caixa negra que traduz os seus programas escritos numa linguagem de ordem superior para a linguagem máquina equivalente, permitindo ao programador abstrair-se dos vários detalhes inerentes às especificidades do *hardware*.

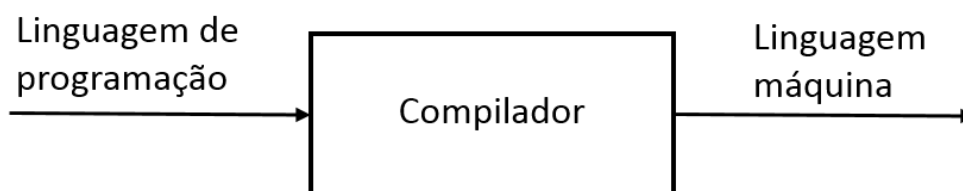


Figura 2.1 - Compilador visto como uma caixa negra

A linguagem de programação de alto nível torna também os programas menos dependentes de computadores ou ambientes computacionais específicos de modo a facilitar a sua portabilidade entre máquinas diferentes.

Os compiladores têm vindo a auxiliar o *hardware* a obter o seu máximo desempenho; um exemplo é o papel importante que o compilador tem no auxílio à resolução de problemas associados ao *pipeline* dos processadores modernos. Os processadores com *pipeline* apresentam três tipos de *hazards* (conflitos): *hazards* estruturais, *hazards* de dados e *hazards* de controlo [1].

Os *hazards* estruturais ocorrem quando uma instrução no *pipeline* necessita de um recurso que está a ser utilizado por outra instrução e deste modo o hardware não pode admitir a combinação de instruções que se pretende executar no mesmo ciclo.

Os *hazards* de dados acontecem quando uma instrução tem a sua execução dependente de um valor ou resultado produzido por uma instrução anterior que ainda esteja em execução, e neste caso torna-se necessário interromper o *pipeline* até a instrução anterior se apresentar concluída.

Os *hazards* de controlo resultam quando existe a necessidade de executar instruções de salto onde o fluxo de endereços das instruções não é o esperado pelo *pipeline*.

Os problemas acima mencionados podem ser amenizados pelo compilador durante a geração de código. O compilador pode incluir algoritmos para rearranjar a ordem das instruções sem que altere a lógica do programa para que seja possível reduzir e evitar estes tipos de *hazards* acelerando assim a execução do programa no processador alvo.

Inicialmente os compiladores eram desenvolvidos para traduzir aplicações de uma determinada linguagem para um processador específico, o que implicava que sempre que aparecia uma nova linguagem ou um novo processador, fosse necessário desenvolver um novo compilador.

Desenvolver um compilador de raiz é uma tarefa complicada e morosa, de modo a combater este problema, a solução foi a divisão da arquitetura do compilador em duas partes: *front end* e *back end*.

O *front end* é responsável por todo o trabalho dependente da linguagem de programação a traduzir, enquanto o *back end* consiste na fase de geração de código e por todos os detalhes dependentes do processador alvo. Esta divisão entre *front-end* e *back-end*, permite através de pequenas alterações, criar uma família de compiladores para novas linguagens ou processadores [2].

Nos subcapítulos seguintes são apresentados alguns dos *front end generators* mais utilizados no desenvolvimento de compiladores.

### **2.1.1. ANTRL (ANother Tool for Language Recognition)**

Segundo [3] ANTRL é um *parser generator* para ler, processar, executar e traduzir texto estruturado ou ficheiros binários. Esta ferramenta é utilizada para construir linguagens

de programação, ferramentas e *frameworks*. A partir de uma determinada gramática, ANTRL é capaz de gerar um *parser* que consegue construir e percorrer árvores sintáticas.

ANTRL aceita como *input* uma gramática que especifica uma linguagem e gera código fonte para um *parser* que reconheça essa linguagem.

A versão 3 desta ferramenta possibilita a geração de código de *parsers* para as seguintes linguagens de programação:

- Ada95
- ActionScript
- C
- C#
- Java
- JavaScript
- Objective-C
- Pearl
- Python
- Ruby
- Standard ML

A última versão lançada tem como linguagens alvo apenas java e C#.

### **2.1.2. Flex – A *fast scanner generator***

Segundo [4] Flex é uma ferramenta para gerar *scanners*. *Scanners* são porções de *software* que reconhecem padrões léxicos num determinado texto. A descrição do *scanner* a ser construído pela ferramenta Flex é feita sob a forma de pares de expressões regulares com uma mistura de rotinas em código C, denominadas de regras. A partir destas regras o Flex gera, por omissão, um ficheiro com código fonte em linguagem C, denominado *lex.yy.c*, que define a rotina *yylex()*. Ao adicionarmos o ficheiro *lex.yy.c* a um programa, o *scanner* vai procurar por ocorrências onde sejam respeitadas as expressões regulares definidas, aquando da sua execução.

### 2.1.3. Bison

Segundo [5] Bison é uma ferramenta de geração de *parsers* de uso genérico. Esta ferramenta cria um ficheiro C ou C++ que implementa um *parser* a partir de uma gramática livre de contexto específico.

O Bison é utilizado para desenvolver uma grande variedade de *parsers*, desde simples calculadoras até linguagens de programação complexas. Esta ferramenta à semelhança da ferramenta Flex, gera um ficheiro de código C produzido à custa de regras definidas num ficheiro que é dado como *input* ao Bison. O ficheiro produzido implementa rotinas em código C que analisa a sequência de *tokens* fornecidos pelo *scanner* e verifica se estes respeitam a gramática definida na construção do *parser*.

Com as ferramentas *generator generator* sempre que se pretenda direccionar o compilador para uma nova linguagem de programação, será necessário redefinir todas as regras para a nova gramática e construir um novo *front end*. De forma a contornar este problema apareceram *frameworks* para o desenvolvimento de código que aceleram o desenvolvimento de compiladores. No livro “Writing compilers and interpreters” [6] é apresentada uma *framework* para o desenvolvimento de compiladores. Neste livro é apresentada e explicada de forma simples e bem documentada a *framework*. Esta *framework* divide o compilador em três partes: *front end*, *Intermediate representation* e *back end*.

Cada uma destas partes pode ainda ser divididas em subpartes mais específicas, mas ainda assim comum a todas as linguagens e arquiteturas. A Figura 2.2 apresenta as diversas partes da *framework* e os componentes inerentes a cada parte. A vantagem da utilização de uma *framework* face às ferramentas *generator generator* é que com a *framework* obtemos uma implementação com uma granularidade inferior, isto é, é possível modificar pequenas partes do compilador para o adaptar para uma nova linguagem ou para uma nova arquitetura. Um aspeto negativo das ferramentas *generator generator* é o facto de a forma de definir a gramática, ser à custa de uma sintaxe própria da ferramenta, que por si só tem a complexidade de uma linguagem de programação, fazendo assim com que o tempo de aprendizagem da ferramenta seja demorado.

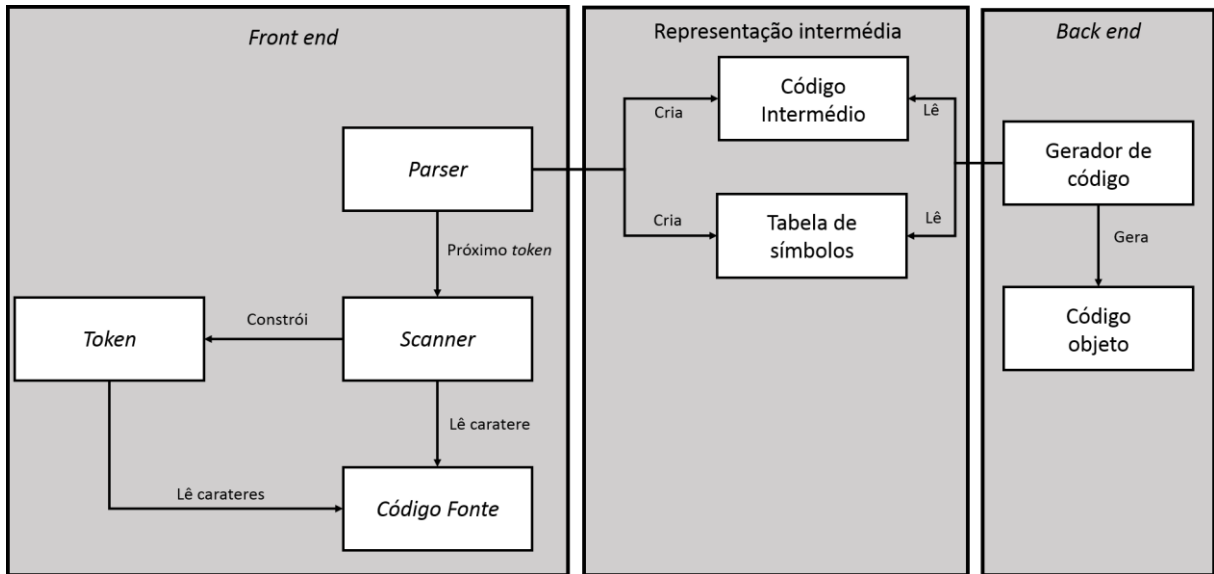


Figura 2.2-Template apresentada no livro [6] para o desenvolvimento de compiladores

Com o desenvolvimento da tecnologia e do conhecimento começou também a aparecer uma diversa gama de linguagens de programação. Cada linguagem tem os seus pontos fortes e fracos, mas de entre todas, houve uma linguagem que desde o seu aparecimento teve grande sucesso entre os programadores, essa linguagem é chamada de linguagem C.

## 2.2. Linguagem C

De acordo com [7] a linguagem C nasceu nos laboratórios da empresa Bell Telephones entre 1969 e 1973, e foi criada por Dennis Ritchie. O nome da linguagem (e a própria linguagem) resulta da evolução de uma outra linguagem desenvolvida por Ken Thompson também nos laboratórios da empresa Bell Telephones, chamada de B. A linguagem nasceu com o objetivo de reescrever o sistema operativo UNIX de forma a tornar este sistema portátil entre várias arquiteturas de *hardware*. A linguagem foi usada mais tarde para o desenvolvimento de uma nova versão do sistema operativo UNIX, que inicialmente tinha sido escrito em *assembly*. Depois do grande sucesso que a linguagem obteve com o sistema operativo UNIX, esta começou a ser utilizada em várias universidades.

A linguagem C é uma linguagem poderosa e flexível, que impõe poucas restrições ao programador, pois tem uma sintaxe extremamente simples, com poucas palavras reservadas, reduzindo o tempo e esforço necessários à aprendizagem da linguagem [8]. Além disso, C é uma linguagem que permite operações de baixo nível enquanto mantém instruções de alto

nível, permitindo assim a portabilidade da aplicação entre máquinas e sistemas operativos diferentes [8]. Talvez o ponto mais forte da linguagem é o facto de ser uma linguagem modular, isto é conseguido com a divisão do código em bibliotecas e funções, o que permite a sua reutilização em projetos futuros [8].

Em 1978, Ritchie e Kernighan publicaram a primeira edição do livro “The C Programming Language” Este livro serviu durante muitos anos como uma especificação informal da linguagem. A versão da linguagem C que o livro descreve é habitualmente referida como “C de K&R”. Nesta versão foram introduzidas as seguintes características na linguagem [9]:

- Tipo de dados *struct*
- Tipo de dados long int
- Tipo de dados unsigned int
- O operador += foi alterado para +=
- O operador -= foi alterado para -=

C de K&R é considerado a parte mais básica da linguagem cujo suporte deve ser assegurado por um compilador C. Durante muitos anos, mesmo após a introdução do padrão ANSI C, o padrão “C de K&R” era considerado o menor denominador comum em que os programadores se apoiavam quando era desejada a máxima portabilidade, uma vez que nem todos os compiladores eram atualizados para suportar na íntegra o padrão ANSI C.

Nos anos seguintes à publicação do C de K&R, algumas características “não oficiais” foram adicionadas à linguagem, suportadas por compiladores da AT&T e de outros vendedores. Estas alterações incluíam [10]:

- Funções *void* e tipos de dados *void\**.
- Funções que retornam tipos *struct* ou *union*.
- Campos de nome *struct* num espaço de nome separado para cada tipo *struct*.
- Atribuição a tipo de dados *struct*.
- Qualificadores *const* para criar um objeto só de leitura.
- Uma biblioteca padrão que incorpora grande parte das funcionalidades implementadas por vários vendedores.
- Enumerados
- O tipo de ponto flutuante de precisão simples.

Após o processo ANSI de padronização, as especificações da linguagem C permaneceram relativamente estáticas por algum tempo. Contudo o padrão foi submetido a uma revisão nos finais da década de 1990, levando à publicação da norma ISO 9899:1999, em 1999. Este padrão é conhecido como “C99” e as novas características incluem [10]:

- Funções *inline*.
- Remoção da restrição sobre a localização da declaração das variáveis.
- Adição do tipo `long long int` (para minimizar o esforço na transição de 32-bit para 64-bit).
- Adição do tipo *bool*.
- Adição do tipo *complex* que representa números complexos.
- Disposições de dados de comprimento variável
- Suporte para comentários de uma linha iniciados por `//`.

O padrão “C99” foi adotado como um padrão ANSI em março de 2000.

A linguagem C continua a ser usada nos dias de hoje, tendo uma forte presença no desenvolvimento de aplicações para microcontroladores: a grande maioria possui um compilador C. Além dos microcontroladores, C é uma linguagem utilizada no desenvolvimento de sistemas operativos, compiladores, analisadores léxicos, bases de dados, editores de texto, etc.

### 2.2.1. GCC

O projeto GNU Compiler Collection é um projeto que desenvolve um conjunto de compiladores para diversas linguagens de programação e para diversas arquiteturas de computadores. A primeira linguagem suportada foi a linguagem C, mas atualmente suporta diversas linguagens de programação como C, C++, Objective-C, Fortran, Java, Ada e Go. Este projeto foi criado com o objetivo de ser o compilador do sistema operativo da GNU [11]. Com o sucesso do GCC no sistema operativo GNU/Linux, rapidamente foi estendido a mais arquiteturas normalmente utilizadas em sistemas embebidos, como microcontroladores ARM e AVR, onde frequentemente o código gerado é *linkado* estaticamente para que corra diretamente no *hardware*.

### 2.2.2. Tiny C Compiler

O Tiny C Compiler (TCC) é um projeto de um compilador de linguagem C para as arquiteturas x86, x86-64 e ARM. Este projeto foi criado por Fabrice Bellard, com o intuito de funcionar em computadores com recursos reduzidos. Este compilador suporta a norma ISO C99 e é cerca de 9x mais rápido que o GCC a compilar o mesmo código fonte [12]. Como é um compilador em que a linguagem em que foi escrito e a linguagem que vai traduzir é a mesma é denominado de *self-hosting compiler*. Este compilador consegue após ser criado a primeira vez por outro compilador, compilar-se a si mesmo sendo de certa forma um teste ao correto funcionamento do compilador.

## 2.3. Sistemas operativos

Um sistema operativo é um programa que gere os recursos de *hardware* de uma máquina, funcionando como intermediário entre aplicações e a máquina. Existem diversos sistemas operativos para diversos fins. A finalidade do sistema operativo influencia diretamente os seus algoritmos internos e a sua complexidade [13].

Uma das principais funções do sistema operativo consiste em proporcionar um ambiente de execução multitarefa. Esta função introduz uma abstração ao fluxo de execução, denominado de tarefa. É da responsabilidade do sistema operativo fornecer serviços de criação e manipulação das tarefas, bem como possibilitar a execução paralela ou pseudo paralela das mesmas. Com este nível de abstração é mais fácil a manutenção do *software* de sistemas complexos, tornando o projeto mais organizado e escalável.

Num cenário onde o número de tarefas é superior ao número de processadores é necessário um mecanismo de seleção das tarefas. A este mecanismo de seleção chama-se escalonador. O escalonador é responsável por dividir o tempo de acesso ao processador entre as várias tarefas. Esta divisão deve ser feita de forma a obter a melhor utilização dos recursos do sistema, e em paralelo satisfazer os requisitos temporais do sistema. É importante referir que o próprio sistema operativo necessita de tempo de execução no CPU, introduzindo ele próprio latências no sistema.

Em sistemas embebidos o requisito mais comum para um sistema operativo é este ser de tempo real. Um sistema de tempo real tem restrições temporais sobre a conclusão de determinadas tarefas. Para satisfazer os requisitos de tempo real das aplicações, foram criados



sistemas operativos especificamente para responder a esta necessidade. Este tipo de sistemas operativos são denominados de sistemas operativos de tempo real ou RTOS (*real time operating system*). Os sistemas operativos de tempo real suportam a execução de aplicações de tempo real, que requerem respostas logicamente corretas nos intervalos de tempo corretos. Qualquer resposta fora do intervalo estipulado, mesmo que seja logicamente correta é uma resposta errada para a aplicação.

## 2.4. Portabilidade e Agnosticismo

Portabilidade é uma característica de um programa de computador se este correr numa arquitetura diferente para a qual foi criado sem necessitar de grandes mudanças no código fonte do programa. *Porting* é a ação necessária para fazer o programa correr no novo ambiente de execução [14]. Em geral programas que são escritos em linguagens de programação de alto nível como a linguagem C necessita apenas de recompilação para o novo ambiente de execução de modo a serem portados. Em sistemas embebidos muitas das vezes o código é compilado para uma arquitetura específica onde não existe um sistema operativo complexo o suficiente para abstrair a aplicação totalmente do *hardware*, nestas situações, é necessário a alteração do código que é específico ao *hardware*.

Agnosticismo de um dispositivo é a capacidade que um componente tem para trabalhar com vários sistemas sem a necessidade de quaisquer adaptações especiais. O termo pode ser aplicado a *hardware* e a *software*. Num sistema informático, agnosticismo refere-se a qualquer componente que é projetado para ser compatível entre os sistemas comuns [15]. Dispositivos agnósticos é uma questão importante nos tempos atuais. É importante conseguir sistemas agnósticos para eliminar o esforço necessário à realização do *porting* destes sistemas para novos dispositivos. Com esta estratégia reduz-se o esforço de engenharia necessário à realização do *porting* do sistema, tendo impacto direto na redução do *time to market*, tornando a empresa mais competitiva.

## 2.5. Arquitetura alvo

O processador alvo, para o qual é expectável que o compilador gere código, é um processador que está a ser desenvolvido no âmbito de outra tese de mestrado, integrado num SoC (*System-on-Chip*). Este SoC está a ser desenvolvido com o intuito de fornecer periféricos

capazes de suportar as aplicações desenvolvidas para a indústria automóvel. O SoC tem as seguintes características:

- Processador RISC
- Instruções de 32bits
- 32 registos de 32bits para uso genérico
- Arquitetura de memória Harvard (memória de código separada da memória de dados).
- *Pipeline* de cinco estágios
- Controlador de interrupções programável, aceitando interrupções de *software* e interrupções de *hardware*.
- Bloco de *timers* programáveis em termos de prioridades, contagem e frequência.
- Comunicação série via UART

O ISA do processador pode ser consultado no Anexo A, onde são apresentadas todas as instruções que compõe o processador, bem como o comportamento associado a cada instrução.

## 2.6. Conclusões

Neste capítulo foram apresentados os termos técnicos e os conceitos relacionados com a presente dissertação e necessários ao sucesso do trabalho. Foram apresentadas ferramentas para a construção de compiladores, bem como os conceitos inerentes a sistemas operativos embebidos e os conceitos de *porting* e agnosticismo. Foram também apresentadas as características principais da arquitetura alvo.

Para este trabalho de dissertação pretende-se desenvolver um compilador para a linguagem de programação C a partir de uma *framework*, de forma que seja fácil a alteração do *front end* para acomodar a ferramenta que tornará o compilador agnóstico ao sistema operativo. Esta ferramenta pretende remover o esforço de engenharia necessário para migrar aplicações entre sistemas operativos. Ainda no compilador é objetivo deste trabalho desenvolver o *back end* capaz de acomodar futuras arquiteturas a desenvolver. Pretende-se também com este trabalho dar suporte ao desenvolvimento de *software* para o novo processador.

## Capítulo 3

### ESPECIFICAÇÃO DO SISTEMA

---

No presente capítulo serão abordadas as fases de análise e *design* realizadas. Os resultados e conclusões deste capítulo servem de guia para melhor compreensão da fase de implementação e os resultados obtido. Numa fase inicial serão abordadas as funcionalidades e restrições do sistema, seguindo-se de uma explicação da divisão do trabalho em várias partes e por fim objetivos a atingir em cada parte. Depois de descritas as funcionalidades e restrições do sistema, será abordada a forma para obter os resultados práticos. O capítulo termina com uma breve descrição das ferramentas utilizadas para a elaboração do presente trabalho.

#### 3.1. Funcionalidades e Restrições

Como mencionado no capítulo introdutório, o trabalho proposto para esta dissertação passa por criar ferramentas de desenvolvimento para um novo microprocessador que está a ser desenvolvido em paralelo com o presente trabalho.

A primeira ferramenta a desenvolver é um compilador C. Este compilador deverá ser capaz de a partir de um ou mais ficheiros do tipo .c criar um único ficheiro com código objeto para o processador alvo. Este código objeto deve ser semanticamente correto e deve também explorar as vantagens oferecidas pelo *hardware* disponível. Durante o processo de compilação caso surja algum erro, este deve ser sinalizado ao programador. Em termos de estrutura do compilador, esta ferramenta deverá assentar numa *framework* que possua um conjunto de funcionalidades. A título de exemplo, a adaptação do *back end* para outras arquiteturas que poderão surgir, seja feito de forma simples, rápida e intuitiva.

Contudo é espectável que o compilador esteja estável o suficiente para que seja capaz de compilar um *microkernel* escrito em linguagem C para a arquitetura alvo. O *microkernel* deverá implementar as funcionalidades mínimas de *multithreading* e as APIs básicas para controlar as *threads*. Com a compilação deste *microkernel* espera-se validar o funcionamento do compilador. Por fim deverá ser criada uma ferramenta que assente na *framework* acima referida, e que permita a migração de aplicações de forma rápida e automática entre sistemas operativos. Esta ferramenta deve também ser feita de forma a facilitar futuras expansões, nomeadamente, seja possível adicionar e remover sistemas operativos. Um último requisito

para esta ferramenta é a sua integração no compilador para que esta atue durante o processo de compilação.

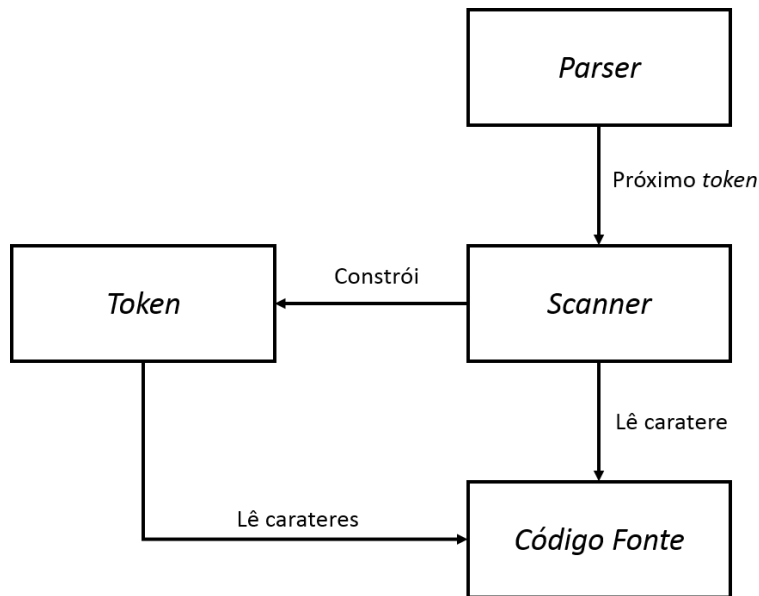
## 3.2. Compilador

O compilador pode ser dividido em três grandes módulos, *front end*, *middle end* e *back end*. O *front end* é o módulo responsável pela leitura e interpretação do código fonte. O *middle end* é o módulo responsável pela criação e atualização da informação acerca dos símbolos que são encontrados durante a interpretação do código fonte. E por fim o *back end* é o módulo responsável pela parte da geração de código para a arquitetura alvo, e portanto dependente do *hardware*. Nas próximas páginas será apresentado cada módulo em maior detalhe.

### 3.2.1. *Front end*

O *front end* é o módulo responsável pela leitura e interpretação do código fonte, composto por dois grandes componentes: o *parser* e o *scanner*. O *parser* é o ponto de entrada do compilador, e tem como função analisar os *tokens* criados pelo *scanner*, enquanto o *scanner* analisa, caracter a caracter o ficheiro fonte, criando vários tipos de *tokens*. Em primeiro lugar, o *parser* analisa o *token* capturado pelo *scanner*, e verifica se é necessário pedir mais *tokens* ou então estamos perante uma frase sintaticamente correta.

Relativamente ao *scanner*, este vai criar vários tipos de *tokens* conforme o tipo do primeiro caracter encontrado. Quando o *scanner* encontrar um caractere que não seja aceite pelo tipo de *token* a criação deste é finalizada, enviando o *token* ao *parser* para que seja analisado. No *parser* são analisados todos os *tokens* e é verificado se é uma frase válida, isto é se a ordem com que aparecem respeita as regras estabelecidas pela gramática da linguagem. A Figura 3.1 mostra o diagrama de blocos do *front end* e as interações entre os diversos componentes.



**Figura 3.1-Diagrama de blocos do front end**

Durante o processo de análise do código fonte o *scanner* cria tipos diferentes de *tokens*, cujos tipos são definidos pelo tipo do primeiro caractere que constitui o *token*. Para cada tipo de *token*, é necessário guardar informações diferentes, para isso foram criadas as classes listadas a seguir:

- *TWordToken*
- *TErrorToken*
- *TEOFToken*
- *TNumberToken*
- *TSpecialToken*
- *TStringToken*

A classe *TWordToken* cria *tokens* do tipo palavra, que podem ser palavras reservadas da linguagem que devem ter um tratamento especial posteriormente, ou simplesmente identificadores, como variáveis ou nomes de funções. A classe *TErrorToken* cria um *token* de erro, que surge quando o *scanner* deteta um caractere que não está definido para a linguagem. Quando um *token* deste tipo é detetado é enviada uma mensagem de erro ao programador. A classe *TEOFToken* cria um *token* que representa o fim do ficheiro fonte. Este *token* é criado quando o *scanner* chega ao fim do ficheiro do código fonte e normalmente sinaliza o *parser* para que este pare as suas ações, passando o compilador à próxima fase. A classe *TNumberToken* cria um *token* que representa um número, este número poderá ser um número inteiro ou um número real. Esta classe é responsável por fazer a conversão da representação

do número em forma de texto para o seu real valor numérico para que possa ser usado pelo compilador na resolução de expressões do programa fonte. A classe *TSpecialToken* cria um *token* que representa os *tokens* especiais da linguagem, como os operadores aritméticos e lógicos. A classe *TStringToken* cria um *token* quando é encontrado texto entre os caracteres “ ”. Normalmente são estes os caracteres usados, mas há linguagens que usam os caracteres ‘ ‘ para delimitar uma *string*. Cada uma das classes acima citadas herdam os métodos e os atributos da classe *TToken*, onde é definido o código do *token*, o tipo de dados que o *token* possa ter e o valor do dado. Para cada atributo existe métodos para ler estes valores da classe para que possam ser usados pelo *parser*.

O componente responsável por criar os diversos *tokens* é o *scanner*. Internamente é constituído pelas classes *TTextScanner* e *TTextInBuffer*. A classe *TTextScanner* é a classe responsável por remover os elementos decorativos do programa como os espaços, mudanças de linha e comentários. É também da responsabilidade desta classe a análise léxica necessária para a identificação do tipo de *token* a criar. A classe *TTextInBuffer* é a classe encarregue pela manipulação dos ficheiros de entrada que contém o código fonte. Esta classe implementa alguns métodos que abstraem a manipulação do ficheiro. No Anexo B pode ser consultado o diagrama de classes que implementa o *scanner*, bem como as suas ligações.

O componente do *front end* que consome e analisa os diversos *tokens* é o *parser*. O *parser* é o componente responsável pela análise sintática e semântica, que são feitas a partir da sequência de *tokens* fornecidos pelo *scanner*. O *parser* é implementado seguindo uma estratégia de análise *top-down*, pois é uma estratégia mais simples de implementar, o que se traduz numa redução do tempo de desenvolvimento que é uma restrição do projeto. O diagrama da classe *TParser* onde é possível identificar os diversos atributos e métodos que constituem a classe pode ser consultado no Anexo D.

A classe *TParser* possui quatro atributos, e um apontador para a classe *TTextScanner* à qual faz pedidos para obter novos *tokens*. Possui também um apontador para a classe *TToken*, encarregue de guardar o *token* lido pelo *scanner*. O último atributo é do tipo *TSyntabStack*, este atributo serve para manipular e gerir as várias tabelas de símbolos que a classe *TParser* vai criar e manter durante todo o processo de análise do código fonte.

A classe *TParser* é a classe responsável pela análise da sequência dos *tokens* e verificar se essa sequência respeita as regras da linguagem a traduzir. Na linguagem C existem quatro tipos de frases possíveis. Existem as diretivas do pré-processador que são iniciadas pelo caractere #. Existem as estruturas de controlo de fluxo do programa que são iniciadas sempre por uma palavra reservada da linguagem. Além disso, é possível ter expressões que são

iniciadas por uma variável ou por um *token* do tipo número. E por fim podemos ter declarações que começam também por uma palavra reservada.

### 3.2.1.1. Estruturas de controlo de fluxo

As estruturas de controlo de fluxo são os elementos mais comuns durante a escrita de aplicações em linguagem C, porque são elas que definem o comportamento da aplicação. Estas estruturas são sempre iniciadas por uma palavra reservada e segue uma sequência de *tokens* bem definida. Caso algum *token* da sequência falhe, a análise do programa é interrompida e é sinalizado o respetivo erro para o programador. Dentro do corpo de cada estrutura pode aparecer diversas estruturas, tanto de forma sequencial como de forma encadeada.

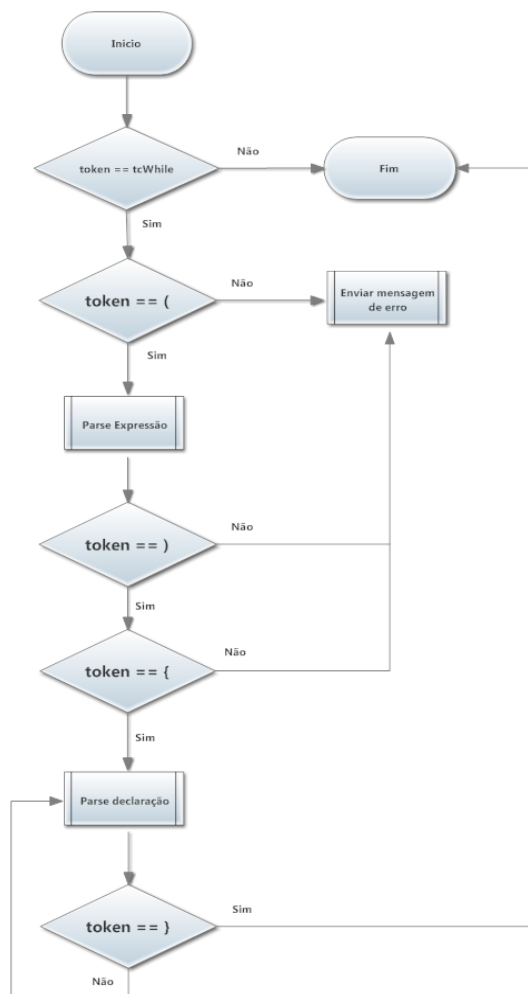


Figura 3.2 - Fluxograma de validação da estrutura de controlo de fluxo while

Na Figura 3.2 é apresentado o fluxograma responsável pela análise da sequência de *tokens* resultante da estrutura de controlo de fluxo *while*. Devido à semelhança sintática entre as várias estruturas de controlo de fluxo, apenas será apresentado o fluxograma para a estrutura do tipo *while*.

### 3.2.1.2. Declarações

Na linguagem C é possível fazer três tipos de declarações, nomeadamente declarações de variáveis, declarações de funções e declaração de novos tipos de dados. Na declaração de variáveis é possível inicializar a variável ou não. Este tipo de declaração começa por uma palavra reservada da linguagem que identifica o tipo de dados da variável, e deve ser seguida por uma palavra que não seja reservada de forma a identificar a variável. Depois do nome da variável é possível inicializar a variável utilizando o *token* "=", seguido do valor que se pretende atribuir, ou então finalizar a declaração. Para assinalar o final da declaração deve aparecer o *token* ";". Se nesta sequência de *tokens* aparecer um *token* diferente será enviada uma mensagem com o respetivo erro ao programador.

A declaração de funções à semelhança da declaração de variáveis começa sempre por uma palavra reservada da linguagem que identifica o tipo de dados de retorno da função, de seguida uma palavra que não seja reservada, normalmente com o nome que se pretende atribuir à função. Após o nome da função deve aparecer o *token* "(" . A seguir a este *token* aparecem os parâmetros da função. O número de parâmetros varia de função para função. Cada parâmetro começa por uma palavra reservada que identifica o tipo de dados de seguida aparece uma palavra identificadora do parâmetro e finalmente caso haja mais parâmetros aparece o *token* "," caso contrário aparece o *token* ")". Para finalizar a declaração da função aparece o *token* ";". Durante a análise dos *tokens* se a ordem e os tipos de *tokens* não corresponderem aos descritos é emitida uma mensagem de erro ao programador.

A declaração de novos tipos de dados é um processo semelhante à declaração de variáveis. A declaração de um novo tipo de dados começa sempre pela palavra reservada *struct* e de seguida deve aparecer uma palavra que identifique o novo tipo de dados. A seguir ao identificador aparece um *token* "{" e no interior das chavetas são declarados os vários campos do novo tipo de dados. A declaração dos campos é igual a uma declaração de variáveis descrito anteriormente. Finalizado a declaração dos campos deve aparecer o *token* "}" que marca o fim do corpo da declaração do novo tipo de dados e por fim aparece o *token*



“;” que marca o fim da declaração. No Anexo C podem ser consultados os fluxogramas que descrevem os passos tomados na verificação dos *tokens* das declarações supracitadas.

### 3.2.2. Representação intermédia

O *parser* à medida que vai adquirindo e analisando os *tokens* criados pelo *scanner*, vai criando uma representação intermédia para o código fonte e mantém a informação dos vários símbolos em diversas tabelas de símbolos. Basicamente o código intermédio é uma representação do código fonte, seguindo uma forma estruturada e livre de elementos decorativos. É a partir desta representação que o *back end* irá gerar o código objeto final. A Figura 3.3 mostra as ações que o *parser* executa sobre a tabela de símbolos e sobre o código intermédio, de forma a criar uma representação do programa fonte pré-tratada de forma a ser mais facilmente interpretada pelo *back end*.

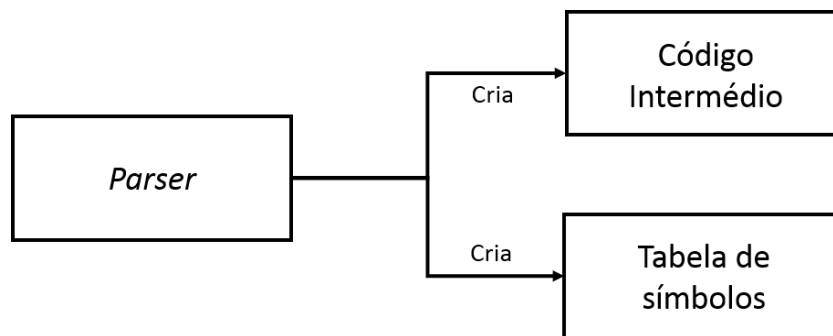


Figura 3.3-Diagrama de blocos da interação do *front end* com a representação intermédia

#### 3.2.2.1. Tabela de símbolos

O *parser* cria e mantém um conjunto de tabelas de símbolos durante todo o processo de tradução do programa fonte. Para analisar uma linguagem estruturada em blocos como a linguagem C, é necessário um conjunto de tabelas de símbolos, uma destinada aos símbolos globais e uma outra aos símbolos privados de cada função declarada no código fonte. Para gerir de forma simples as regras de *scope* da linguagem, é utilizada uma *stack* de tabelas de símbolos, assim sempre que entramos numa função fazemos *push* da respetiva tabela de símbolos para o topo da *stack* e quando saímos da função fazemos o respetivo *pop* da tabela

de símbolos, de modo a removê-la do topo da *stack*. O aspecto da *stack* de tabelas de símbolos é apresentado na Figura 3.4.

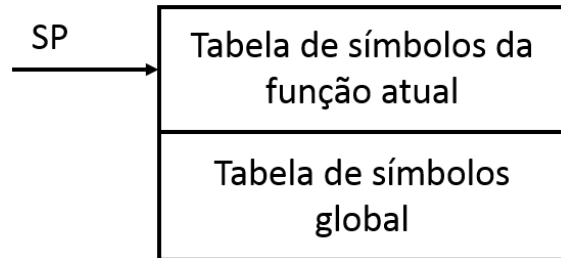


Figura 3.4-Representação da *stack* de tabelas de símbolos

Durante o processo de análise do código fonte, quando o *parser* tenta procurar um símbolo e este não se encontra em nenhuma das tabelas presentes na *stack*, significa que a função está a usar um símbolo que não pertence ao seu *scope* ou um símbolo inexistente. Quando isto se verifica deve ser enviada uma mensagem de erro ao programador, e a análise do código é abortada.

A tabela de símbolos é implementada sob a forma de uma árvore binária com ordenação alfabética com o objetivo de acelerar as operações de procura. A Figura 3.5 mostra um aspeto possível para uma tabela de símbolos.

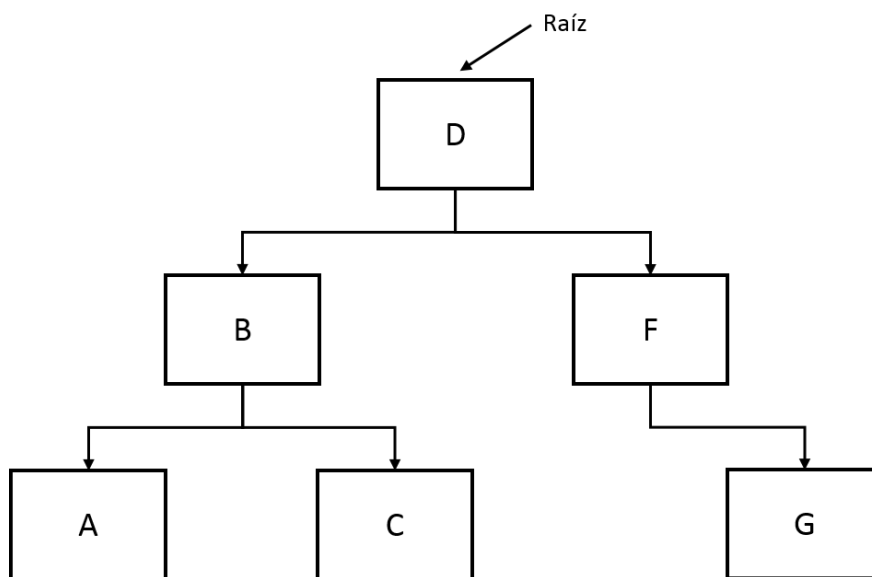


Figura 3.5-Aspeto de uma tabela de símbolos ordenada alfabeticamente

Cada nó da árvore binária representa a informação de cada símbolo, contendo a *string* com o nome do identificador, um atributo do tipo *TDefn* que especifica como o identificador foi definido (variável, apontador, função, etc.) e um atributo do tipo *TType*, que guarda o tipo de dados de cada identificador.

Na Figura 3.6 é apresentado o diagrama de classes da estrutura que guarda a informação de cada símbolo.

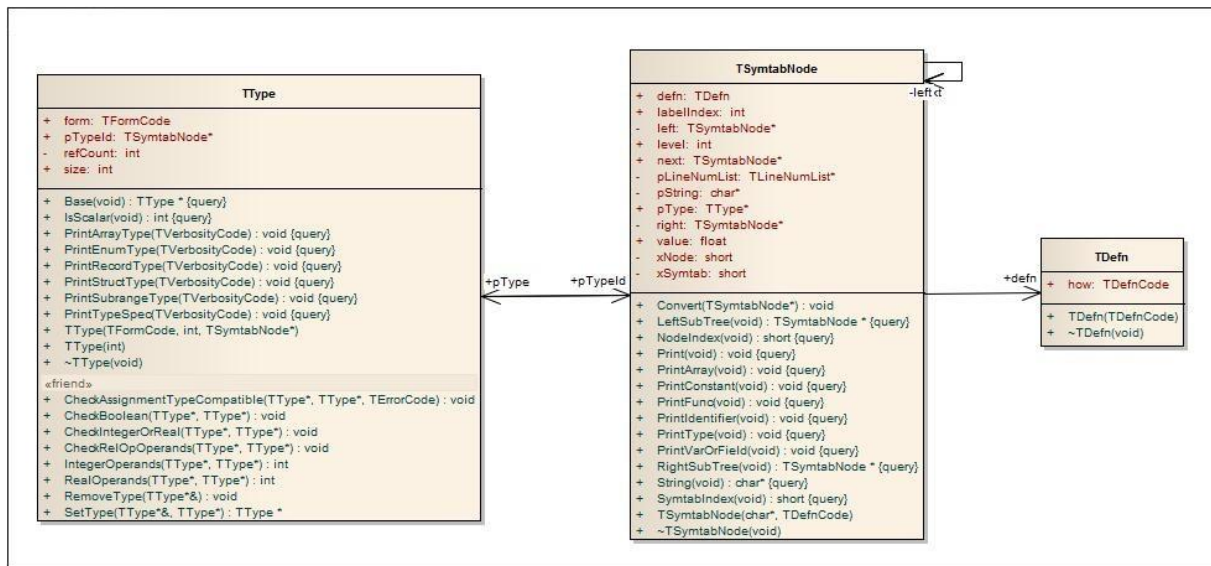


Figura 3.6-Estrutura e relações entre as classes que guarda a informação de cada símbolo

### 3.2.2.2. Código intermédio

A representação das instruções em código intermédio é bastante útil, pois simplifica o *back end* abstraíndo-o da estrutura de instruções da linguagem. A estrutura do código intermédio é constituído por um vetor, em que cada posição contém um *token* do código fonte. Para o *token* identificador além de guardar o código do próprio *token* é também guardado o *id* da tabela de símbolos e o *id* do nó do símbolo onde está guardada a informação do identificador. Através desta estratégia é possível aceder diretamente ao símbolo sem a necessidade de pesquisas na tabela de símbolos.

Na Figura 3.7 é apresentado a título de exemplo a estrutura do código intermédio para a expressão  $x=y+3$ ; escrita em linguagem C.

<i>Token</i> identificador
<i>Id</i> tabela de símbolos
<i>Id</i> do símbolo
<i>Token</i> igual
<i>Token</i> identificador
<i>Id</i> tabela de símbolos
<i>Id</i> do símbolo
<i>Token</i> soma
<i>Token</i> identificador
<i>Id</i> tabela de símbolos
<i>Id</i> do símbolo

**Figura 3.7-Exemplo da estrutura do código intermédio**

### 3.2.3. *Back end*

Depois do código ser analisado, e livre de quaisquer erros, e a representação intermédia estar criada, entra em ação o *back end*. Aqui são selecionadas as instruções a executar e a alocação de registos do processador. Para gerar o código objeto, o *back end* consulta o código intermédio criado na etapa anterior e a informação necessária das diversas tabelas de símbolos.

Na Figura 3.8 são apresentadas as ligações entre o *back end* e a representação intermédia. O gerador de código, tendo em conta o código intermédio e as informações

provenientes da tabela de símbolos cria o código objeto final. O código final é um único ficheiro com código executável pronto a correr no processador.

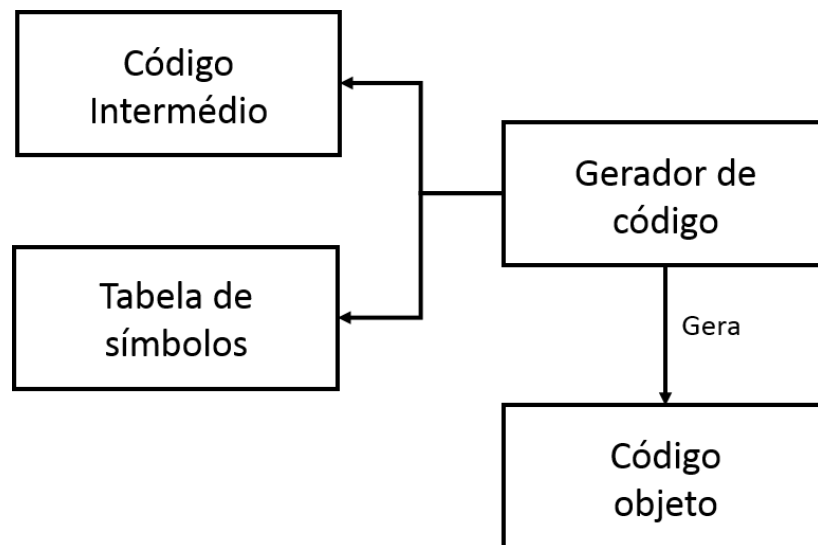


Figura 3.8-Diagrama de blocos *back end* em interação com a representação intermedia

Para auxiliar a geração de código foram criadas duas classes auxiliares, uma responsável por gerir a atribuição dos registos da máquina, e outra com o intuito de resolver as expressões matemáticas presentes no código fonte. De forma a gerir os registos da máquina foi criada a classe *TRegisterFile*, esta classe possui métodos para associar um registo a uma variável, procurar se um registo está associado a uma variável, e libertar registos associados a uma variável.

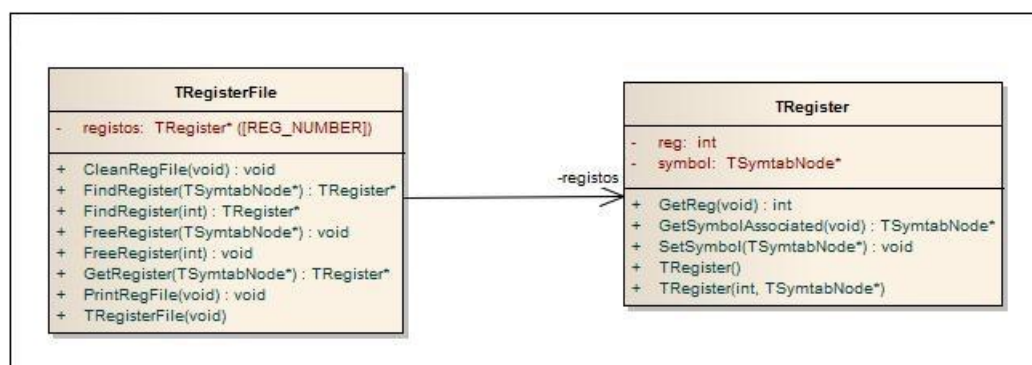


Figura 3.9 - Classe que gere a utilização dos registos da máquina

Para auxiliar na resolução das expressões matemáticas foi criada a classe *TExpressionStack* que é uma *stack* onde são colocados os operandos conforme a prioridade da

operação a resolver. Durante a resolução são retirados os dois valores do topo da *stack*, é resolvida a operação entre os dois valores retirados e o resultado é colocado de novo na *stack*. Na Figura 3.10 são apresentados os atributos e os métodos da classe *TExpressionStack*.

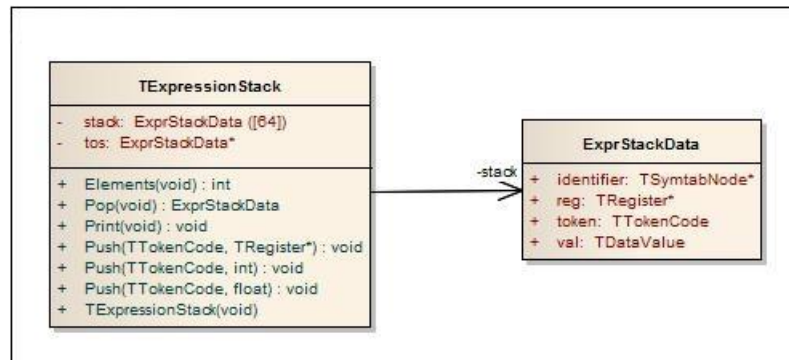


Figura 3.10 - *Stack* que auxilia a resolução de expressões matemáticas

O tipo de elementos da *stack* pode variar e para uniformizar o tipo do elemento foi criado um tipo de dados composto que alberga todos os tipos de dados possíveis de entrar na *stack*. Com esta abordagem durante a fase de implementação é possível evitar apontadores do tipo *void* e diversas conversões entre tipos de apontadores que prejudicam o desempenho do compilador.

### 3.2.4. ABI (*Application Binary Interface*)

O ABI define os aspetos do código máquina gerado através da utilização do compilador, como por exemplo, o tamanho e alinhamento dos tipos de dados nativos, a forma como os registos são usados, convenção de chamada das funções, estrutura e organização da *stack* durante a execução do programa, etc.

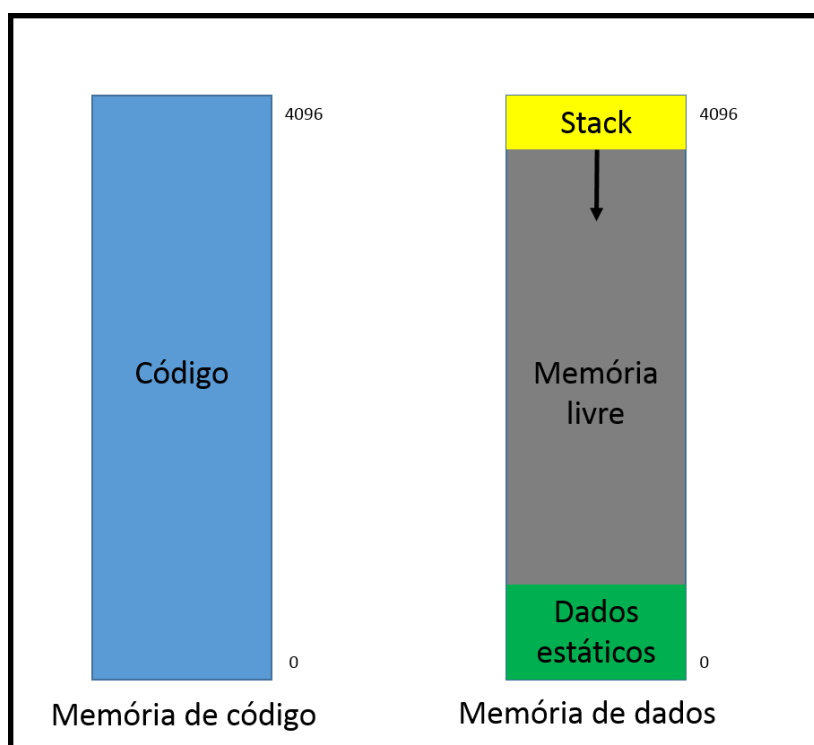
Nesta dissertação é definido um ABI para a arquitetura alvo com a linguagem de programação C.

Como o código gerado para a arquitetura alvo executa diretamente sob o *hardware* sem a presença de um sistema operativo entre a aplicação e o *hardware*, o compilador gera código *linkado* estaticamente. A Tabela 1 mostra os registos definidos no ABI e o seu propósito de utilização.

**Tabela 1 - Registos definidos no ABI**

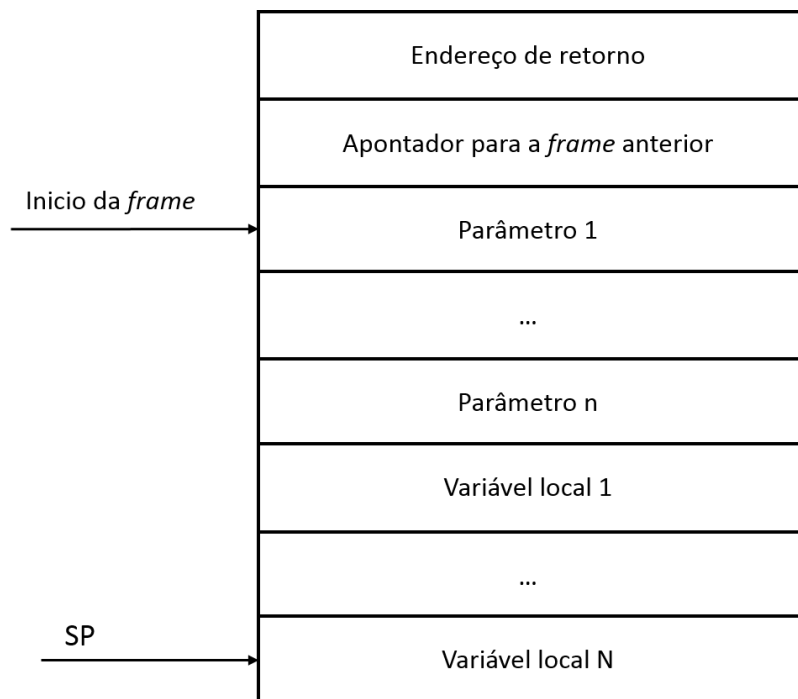
Registo	Função
R0	Contém sempre o valor 0
R1 - R13	Registos para uso geral
R14	Contém o endereço do início do segmento de dados
R15	Guarda sempre o valor de retorno da função
R16	Apontador para o início da <i>frame</i> da função a executar no momento
R17 – R30	Registos para uso geral
R31	Apontador para o topo da <i>stack</i> , este registo é gerido por <i>hardware</i>

Além de definir as funções de cada um dos registos do processador, também é necessário dividir a memória em regiões, com o objetivo de facilitar o uso por parte dos programas. A região de memória destinada à *stack* é a região alta da memória. Esta região é imposta pelo *hardware*, pois o *stack pointer* é inicializado com o endereço da última posição de memória e quando a *stack* cresce o valor do endereço decrementa. Com o intuito de maximizar a quantidade de memória disponível para a *stack*, a área de dados estáticos (variáveis globais e vetores) foi mapeada nos endereços mais baixos da memória. Na Figura 3.11 é apresentado o esquema com a distribuição das áreas de dados pela memória.



**Figura 3.11 - Áreas de memória usadas durante a execução de um programa**

A linguagem C permite a uma função invocar a si própria, implicando que os valores dos parâmetros e as variáveis locais sejam inicializados aquando da nova invocação, mas é desejável que quando a função retornar os valores antigos sejam restaurados. Para conseguir esta funcionalidade foi definida uma estrutura de *frame* para a *stack*. A cada invocação é criada uma nova *frame* para alocar os parâmetros e as variáveis locais da função. Desta forma, nunca vão surgir incoerências sobre os dados que o processador está a executar. Na Figura 3.12 é apresentada a estrutura adotada para a *frame* da *stack*.



**Figura 3.12 - Estrutura de uma *frame* na *stack***

A *stack* cresce de forma descendente e o *stack pointer* contém sempre o valor do item mais baixo da *stack*. Relativamente à *frame* na sua base está colocado o endereço de retorno da função, e na posição imediatamente abaixo é colocado o apontador para o início da *frame* anterior. A seguir ao apontador para a *frame* anterior são colocados todos os parâmetros que a função recebe e todas as variáveis locais à função.

Para implementar o tipo de dados *float* foi decidido usar o *standard* IEEE 754. Este *standard* define regras de normalização a serem seguidas nas operações e define o formato de representação em memória de números com vírgula flutuante [16]. Como a arquitetura alvo não suporta operações com vírgula flutuante diretamente, todos os algoritmos serão implementados em *software* para dar suporte às quatro operações aritméticas base (i) soma;



(ii) subtração; (iii) multiplicação; (iv) divisão. Além das quatro operações será dado suporte para a conversão de vírgula flutuante em inteiro e vice-versa.

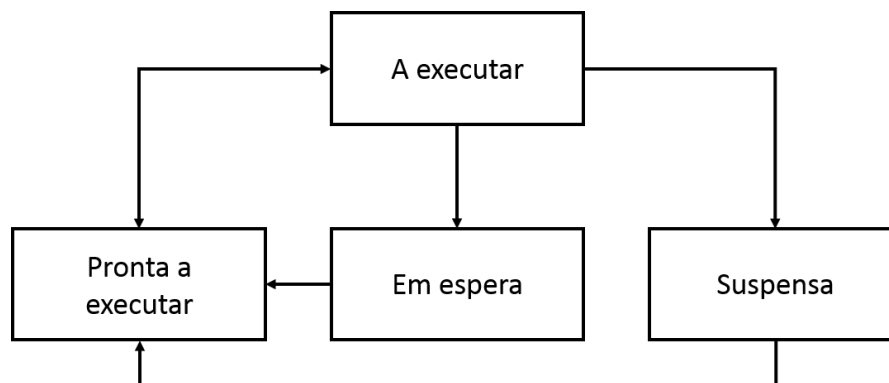
Na Tabela 2 são expostos os tipos de dados nativos e os respectivos tamanhos.

**Tabela 2 - Tamanho dos tipos de dados primitivos**

Tipo dados	Tamanho ( <i>bytes</i> )
<i>char</i>	1
<i>int</i>	4
<i>unsigned int</i>	4
<i>float</i>	4
Apontador qualquer tipo	4

### 3.3. *Microkernel*

A segunda parte da presente dissertação passa por desenvolver um *microkernel* para o novo microprocessador, com o objetivo de aumentar as ferramentas disponíveis para o auxílio na criação de aplicações para o novo microprocessador. Assim foi decidido que o *microkernel* deve ser escrito sob a forma de bibliotecas em linguagem de programação C, e deve oferecer os serviços mínimos de *multithreading*. Devido à natureza da arquitetura alvo apenas uma tarefa pode estar em execução num determinado instante de tempo, e as restantes devem estar em algum outro estado. Quando uma tarefa é criada é colocada automaticamente no estado de “pronta a executar”. Na Figura 3.13 está representado a máquina de estados das tarefas e possíveis transições.



**Figura 3.13 - Diagrama de estados das tarefas do *microkernel***

O escalonador é o componente do *microkernel* responsável por decidir que tarefa deve executar num determinado momento. Uma tarefa pode ser retirada de execução e recolocada posteriormente várias vezes durante o seu tempo de vida. Adicionalmente, uma tarefa pode retirar-se de execução de forma involuntária, cedendo o processador, por precisar de esperar por algum evento externo à sua própria execução. Quando uma tarefa sai de execução é necessário colocar outra. O processo de escolha da nova tarefa para execução é baseado no seu estado e na sua prioridade. Assim a próxima tarefa a receber tempo de execução é a tarefa que tiver maior prioridade e que esteja no estado “pronta a executar”. Neste caso o algoritmo do escalonador é denominado de *fixed priority preemptive scheduling*.

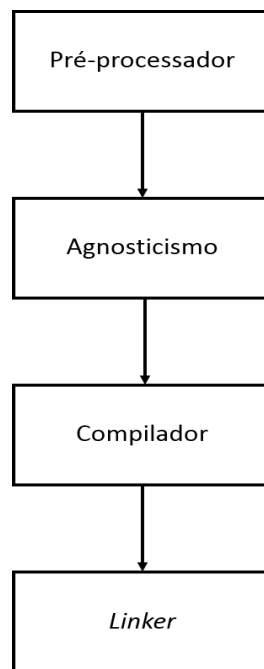
Enquanto uma tarefa utiliza o processador esta acede aos registos e às memórias de código e de dados, tal como qualquer outro programa. Ao estado do conjunto de todos estes recursos são denominados de contexto da tarefa. Uma tarefa é um bloco de código sequencial, pode ver a sua execução ser interrompida a qualquer instante. Considerando como exemplo uma tarefa que é suspensa imediatamente a seguir ao realizar uma operação de soma, outras tarefas poderão tomar conta do processador e executar e modificar os valores contidos nos registos. Quando a tarefa original retornar à execução o valor resultante da soma realizada na instrução anterior poderá estar incorreto, comprometendo a integridade da aplicação. Para prevenir este problema é vital que quando a execução de uma tarefa seja resumida o estado do processador seja exatamente o mesmo aquando da sua suspensão. Ao processo de salvar o contexto de uma tarefa a ser suspensa e o restauro do contexto de uma tarefa a ser resumida é denominado de comutação de contexto.

Na arquitetura alvo o *stack pointer* é o registo 31 do banco de registos, assim o contexto de uma tarefa no processador alvo é dado pelo *program counter* juntamente com os 32 registos do processador.

### **3.4. Agnosticismo**

A terceira e última parte deste projeto, passa por introduzir no compilador uma nova funcionalidade. Pretende-se construir uma ferramenta que possibilite o desenvolvimento de aplicações com chamadas ao sistema operativo genéricas. A aplicação é desenvolvida independentemente do sistema operativo, posteriormente, e de uma forma automática são alteradas para as chamadas específicas do sistema operativo pretendido. Esta ferramenta deve permitir a adição de novos sistemas operativos sem a necessidade de alterar o código fonte da

ferramenta, ou seja o compilador deve ser agnóstico do sistema operativo alvo sobre o qual a aplicação irá correr.



**Figura 3.14 - Fases de compilação depois de introduzida componente agnóstica do sistema operativo**

Para tornar o processo do *porting* da aplicação para o sistema operativo pretendido, transparente ao programador, o processo foi introduzido no compilador como uma fase de compilação. O objetivo principal desta ferramenta é eliminar o esforço de engenharia necessário ao *porting* de aplicações entre sistemas operativos. Na Figura 3.14 são apresentadas as fases de compilação depois de introduzida a ferramenta que torna o compilador agnóstico do sistema operativo.



## Capítulo 4

### IMPLEMENTAÇÃO

---

Neste capítulo são explicadas as três ferramentas desenvolvidas no âmbito desta dissertação. O capítulo começa por explicar a ferramenta que serve de base a todo o trabalho, o compilador. É apresentada a implementação das classes e dos métodos mais importantes nesta ferramenta. Após a descrição do compilador é apresentado o *microkernel*. Aqui são mostrados os pontos chave do *microkernel* bem como as API's presentes para auxiliar no desenvolvimento de aplicações *multithread*. Por fim é apresentada a ferramenta integrada no compilador que servirá de suporte ao desenvolvimento de aplicações genéricas independentes do sistema operativo utilizado. Aqui será mostrado o fluxo de compilação e onde esta ferramenta se encaixa e a forma como foi implementada de forma a tornar o compilador agnóstico do sistema operativo alvo.

#### 4.1. Compilador

O processo de implementação do compilador foi dividido em três partes: (i) *front end* é a parte responsável por ler e interpretar o(s) ficheiro(s) com o código fonte; (ii) *middle end* é parte responsável por criar e atualizar as informações sobre os diversos símbolos encontrados ao longo da análise do código fonte; (iii) *back end* é a parte responsável pela geração de código objeto para a arquitetura alvo. De seguida serão apresentadas em pormenor as partes acima referidas.

##### 4.1.1. *Front end*

O *front end* é a parte do compilador responsável por ler e interpretar o(s) ficheiro(s) com o código fonte. O *front end* é composto por dois componentes chave, o *scanner* e o *parser*. O *scanner* é o componente responsável por ler o código fonte, remover os elementos decorativos (espaços, comentários e mudanças de linha) e criar os *tokens* necessários à ação do *parser*. O *parser* é responsável por analisar e interpretar a ordem com que os *tokens* são fornecidos e verificar se esta ordem respeita a gramática da linguagem a traduzir.

#### 4.1.1.1. Scanner

O *scanner* analisa o fluxo de caracteres do código fonte em busca de expressões regulares e de caracteres especiais, agrupando-os em conjuntos de caracteres com determinados tipos por forma a criar *tokens*. Para ser possível agrupar os caracteres é necessário “classificar” os mesmos, para posteriormente ser possível a identificação do *token* a criar. O tipo do *token* a criar é definido pelo tipo do primeiro caractere pelo qual é formado o *token*. A Figura 4.1 mostra o construtor da classe *TTextScanner*, onde é inicializada uma estrutura de dados, onde é guardado o tipo de cada caractere.

```
TTextScanner::TTextScanner (TTextInBuffer
*pBuffer) : pTextInBuffer (pBuffer)
{
    int i;

    for (i=0; i<127; i++) {charCodeMap[i]=ccError;}
    for (i='a'; i<='z'; i++) {charCodeMap[i]=ccLetter;}
    for (i='A'; i<='Z'; i++) {charCodeMap[i]=ccLetter;}
    for (i='0'; i<='9'; i++) {charCodeMap[i]=ccDigit;}
    charCodeMap['&']=charCodeMap['|']=ccSpecial;
    charCodeMap['+']=charCodeMap['-']=ccSpecial;
    charCodeMap['*']=charCodeMap['/']=ccSpecial;
    charCodeMap['=']=charCodeMap['^']=ccSpecial;
    charCodeMap['.']=charCodeMap[',']=ccSpecial;
    charCodeMap['<']=charCodeMap['>']=ccSpecial;
    charCodeMap['(']=charCodeMap[')']=ccSpecial;
    charCodeMap['[']=charCodeMap[']']=ccSpecial;
    charCodeMap['{']=charCodeMap['}']=ccSpecial;
    charCodeMap[':']=charCodeMap[';']=ccSpecial;
    charCodeMap['!']=charCodeMap['%']=ccSpecial;
    charCodeMap['~']=ccSpecial;
    charCodeMap[' ']=charCodeMap['\t']=ccWhiteSpace;
    charCodeMap['\n']=charCodeMap['\0']=ccWhiteSpace;
    charCodeMap['\r']=ccWhiteSpace;
    charCodeMap['"']=charCodeMap['\'']=ccQuote;
    charCodeMap[eofChar]=ccEndOfFile;
}
```

Figura 4.1 - Atribuição de tipo a cada caractere da tabela ASCII

Durante a interpretação do código fonte, o *scanner* ignora os caracteres decorativos do programa, por caracteres decorativos entende-se espaços, tabulações, mudanças de linha e comentários. A Figura 4.2 mostra o método de construção de um *token*. Neste método podemos ver a chamada da função *SkipWhiteSpaces()* que como o nome sugere ignora espaços em branco, ignora também mudanças de linha e comentários. Depois de ignorar os espaços em branco procede-se à análise do tipo do primeiro caracter. Este caractere define o tipo de *token* a construir. Identificado o *token*, é chamado o método *Get()* específico de cada tipo de *token* para que toda a *string* seja construída.

```

TToken *TTextScanner::Get (void)
{
    TToken *pToken;

    SkipWhiteSpace ();

    switch (charCodeMap[pTextInBuffer->Char ()])
    {
        case ccLetter:      pToken=&wordToken;      break;
        case ccDigit:      pToken=&numberToken;    break;
        case ccQuote:      pToken=&stringToken;    break;
        case ccSpecial:    pToken=&specialToken;    break;
        case ccEndOfFile:  pToken=&eofToken;       break;
        default:           pToken=&errorToken;     break;
    }

    pToken->Get (*pTextInBuffer);
    return pToken;
}

```

**Figura 4.2 - Método para construção do *token***

Depois do *token* ter sido criado, este é passado ao *parser* que vai verificar o seu tipo e significado relativamente à posição em que aparece no código fonte, isto é, segundo a gramática de contexto livre da linguagem.

#### 4.1.1.2. *Parser*

O *parser* é o componente responsável por analisar a sequência de *tokens* fornecidos pelo *scanner*. Com estes *tokens* o *parser* analisa a sequência e guarda as informações relevantes à cerca dos símbolos (variáveis, funções, constantes, etc) em tabelas de símbolos, e agrupa os *tokens* de forma a criar uma representação intermédia facilmente interpretada posteriormente pelo *backend*. O *parser* implementado segue uma abordagem *top-down*, em que o *parser* identifica o primeiro *token* de um *statement* e recursivamente chama as funções necessárias à resolução do *statement* específico. Este tipo de *parsers* são denominados de *predictive parsers*. Para o correto funcionamento deste tipo de *parsers* a gramática não pode ser ambígua, ou seja para uma sequência de *tokens* apenas pode existir uma *parse tree*. Durante a análise das *parse trees* sempre que apareça um *token* inesperado ou desconhecido é da responsabilidade do *parser* emitir um erro e abortar a análise do código fonte. Na Figura 4.3 é apresentado o método que diferencia os vários *statements* e chama o método que resolve cada um.

```

void TParser::ParseStatement(void)
{
    switch (token)
    {
        case tcStar :
            while(token == tcStar)
            {
                icode.Put(token);
                GetToken();
            }
        case tcIdentifier:
            ParseAssignment();
            CondGetToken(tcSemicolon, errMissingSemicolon);
            break;
        case tcAsm:      ParseAsmStatement();break;
        case tcWhile:    ParseWHILE();      break;
        case tcIf:       ParseIF();          break;
        case tcFor:      ParseFOR();         break;
        case tcSwitch:   ParseSWITCH();      break;
        case tcReturn:   ParseRETURN();     break;
        case tcWriteBus: ParseWriteBus();    break;
        case tcReadBus:  ParseReadBus();    break;
    }
}

```

**Figura 4.3 - Método que diferencia o tipo de *statement***

Depois de ser determinado o tipo de *statement* a ser resolvido é necessário consumir mais *tokens* por forma a verificar a validade do mesmo. Durante a análise dos *tokens* o *parser* cria e mantém atualizadas diversas tabelas de símbolos. Além das tabelas de símbolos o *parser* cria blocos de representação intermédia do código para cada função presente no código fonte. Estes blocos de código intermédio são uma forma de código “pré digerida” a partir do código fonte. Esta forma de código pode ser assim tratada mais eficientemente pelo *back end*.

#### 4.1.2. Representação intermédia

A representação intermédia do código fonte é guardada em duas estruturas importantes, na tabela de símbolos que guarda todas as informações relevantes acerca de variáveis, funções e constantes que vão aparecendo ao longo do código, e o código intermédio que guarda o código fonte de uma forma estruturada mais facilmente acedida e interpretada pelo *back end*. O código intermedio é guardado numa estrutura de dados reservada em memória para que a sua leitura e escrita ocorram de forma mais rápida, pois estes dados serão acedidos inúmeras vezes durante o processo de geração de código.



#### 4.1.2.1. Tabela de símbolos

Durante a tradução do programa fonte para código máquina da arquitetura alvo são criadas e mantidas diversas tabelas de símbolos. Existe uma tabela de símbolos global que guarda as informações de variáveis globais, funções declaradas, constantes, tipos de dados básicos e tipos de dados criados através da palavra reservada *struct*. Para além da tabela de símbolos global existe também uma tabela de símbolos para cada função utilizada no programa fonte. Estas tabelas de símbolos locais guardam informação sobre as variáveis locais e os parâmetros que têm de ser passados à função.

```
private:
    TSymtabNode *left, *right;
    char        *pString;
    short       xSymtab;
    short       xNode;

public:
    TSymtabNode *next;
    TType       *pType;
    TDefn       defn;
```

**Figura 4.4 - Atributos da classe *TSymtabNode***

Na Figura 4.4 é apresentado o código que implementa a classe do nó da tabela de símbolos que guarda a informação de um símbolo. Além dos atributos inerentes à implementação da estrutura de dados baseada em árvores binárias, esta classe possui atributos para guardar informações relevantes acerca do símbolo. Estas informações são guardadas nos atributos *pString* onde é guardado o nome, *pType* onde é guardada a informação acerca do tipo de dados do símbolo e no atributo *defn* onde são guardadas informações como valores de inicialização das variáveis, valores de constantes, tamanho e localização de vetores em memória, etc.

Os nós são arrançados na forma de árvores binárias, onde as árvores posteriormente são mantidas e organizadas numa *stack* de árvores binárias para uma fácil implementação das regras de *scope* da linguagem. As tabelas de símbolos ao passar do *front end* para o *back end* sofrem uma alteração na sua morfologia. No *front end* têm a forma de árvore binária, ao passar para o *back end* são arrançadas em vetores de dados para que se consiga aceder ao nó sem a necessidade de pesquisa. Aceder ao nó diretamente é possível pois o *back end* terá a informação do acesso ao *id* do nó pretendido.

#### 4.1.2.2. Código intermédio

O código intermédio é uma forma tratada do código fonte. Esta representação é uma forma “pré-digerida” do código fonte, pois só guarda os *tokens* necessários à geração de código. Do código intermédio são removidas todas as declarações, ficando apenas os *statements* e os seus *tokens* terminadores. A estrutura que alberga o código intermédio é um vetor que em cada entrada guarda o código do *token*. Quando o *token* é um identificador, são colocados o *id* da tabela de símbolos e o *id* do nó da tabela de símbolos onde se encontra a informação relativa ao identificador nas duas posições adjacentes. Com esta estratégia é possível ao *back end* aceder às informações de um identificador com apenas uma iteração em vez da tradicional pesquisa de árvore binária.

```
void Tlcode::Put(TTokenCode tc)
{
    if(errorCount > 0) return;
    char code = tc;
    CheckBounds(sizeof(char));
    memcpy((void*)cursor, (const void*)&code, sizeof(char));
    cursor += sizeof(char);
}

void Tlcode::Put(const TSymtabNode *pNode)
{
    if(errorCount > 0) return;

    short xSymtab = pNode->SymtabIndex();
    short xNode= pNode->NodeIndex();

    CheckBounds(2*sizeof(short));
    memcpy((void*)cursor, (const void*)&xSymtab, sizeof(short));
    memcpy((void*)(cursor + sizeof(short)), \
(const void*)&xNode, sizeof(short));

    cursor += 2*sizeof(short);
}
```

Figura 4.5 - Métodos de introdução de *tokens* na estrutura do código intermédio

Na Figura 4.5 são apresentados os dois métodos de introdução de *tokens* no código intermédio. O primeiro método *Put* que recebe como parâmetro um *TTokenCode*, verifica se no segmento de código intermédio é possível introduzir o código do *token*. Se for possível copia o mesmo para o vetor e atualiza o cursor para a próxima posição. O segundo método *Put* é chamado quando o *token* a introduzir é um identificador. Neste método é admitido que o

código do *token* já foi introduzido e apenas introduz o *id* da tabela de símbolos e o *id* do nó no qual se encontra a informação do símbolo.

```
TToken *Tlcode::Get (void)
{
    ...

    switch(token)
    {
    case tcIdentifier:
    case tcNumber:
    case tcString:
        pNode=GetSymtabNode ();
        strcpy (pToken->string,pNode->String ());
        break;

    default:
        pNode=NULL;
        if(token != tcEndOfFile)
        {
            strcpy (pToken->string,symbolStrings[code]);
        }
        break;
    }
    return pToken;
}

TSymtabNode *Tlcode::GetSymtabNode (void)
{
    extern TSymtab **vpSymtabs;
    short xSymtab, xNode;

    memcpy((void*)&xSymtab,(const void*)cursor,sizeof(short));
    memcpy((void*)&xNode,(const void*) \
(cursor + sizeof(short)),sizeof(short));

    cursor += 2*sizeof(short);

    return vpSymtabs[xSymtab]->Get(xNode);
}
```

**Figura 4.6 - Métodos de obtenção de *tokens* a partir do código intermédio**

Na Figura 4.6 são apresentados os métodos para realizar a operação inversa dos acima citados. O método de *Get* reconstrói o *token* a partir do código intermédio. Caso seja um *token* com informação na tabela de símbolos é chamado o método *GetSymtabNode* para obter essa informação. A classe *Tlcode* é a classe que faz a interface entre o *front end* e o *back end*. Os métodos de *Put* são utilizados pelo *parser* para escrever o código fonte tratado por este. O método de *Get* é utilizado pela classe *TCodeGenerator* para consumir os *tokens* do código intermédio, e a partir destes gerar o código objeto final.

### 4.1.3. Back End

O *back end* é a parte do compilador responsável por gerar o código objeto final para a arquitetura alvo. Para gerar o código o *back end* vai ler o código intermédio contido na classe *ICode* apresentada anteriormente e usar as informações das diversas variáveis e funções presentes nas tabelas de símbolos.

#### 4.1.3.1. Gestão dos registos da máquina

Para gerar código máquina funcional é necessário um uso eficiente dos registos. Para uma fácil gestão dos registos foram criadas duas classes, a classe *TRegister* e a classe *TRegisterFile*. A classe *TRegister* guarda a informação relativa ao registo. Nesta classe é guardado o número do registo e o apontador para o símbolo ao qual o registo está associado. Na Figura 4.7 é mostrado o código da declaração da classe *TRegister*, onde é possível visualizar os atributos e os métodos de interface da classe.

```
class TRegister
{
    int reg;
    TSymtabNode *symbol;

public:
    TRegister() {}
    TRegister(int Reg, TSymtabNode* Symbol);
    int GetReg(void);
    TSymtabNode* GetSymbolAssociated(void);
    void SetSymbol(TSymtabNode* Symbol);
};
```

Figura 4.7 - Declaração da classe TRegister

Para gerir todos os registos foi implementada a classe *TRegisterFile*. Esta classe contém métodos para associar uma variável a um registo, procurar se uma variável está associada a um registo, procurar se um registo tem variável associada e libertar um registo. Na Figura 4.8 é apresentada a classe *TRegisterFile* sendo possível identificar o atributo onde são guardadas todas as informações acerca dos registos da máquina e os métodos da classe.

```

class TRegisterFile
{
    TRegister *registos[REG_NUMBER];

public:
    TRegisterFile(void);
    TRegister* GetRegister(TSyntabNode* var);
    void FreeRegister(TSyntabNode* var);
    void FreeRegister(int reg);
    void CleanRegFile(void);
    TRegister* FindRegister(TSyntabNode* var);
    TRegister* FindRegister(int reg);

    void PrintRegFile(void);
};

```

**Figura 4.8 - Declaração da classe *TRegisterFile***

#### 4.1.3.2. Método de entrada no *back end*

O primeiro passo para gerar código de programas escritos em linguagem C é procurar pela função de entrada do programa, que por convenção é chamada de *main*. Se esta função não existir é emitido um erro ao programador. Caso a função exista é necessário preparar a instância da classe *ICode* que contém o código intermedio da função *main*, e introduzir na *stack* de tabelas de símbolos a tabela de símbolos desta função. A Figura 4.9 mostra a procura pelo ponto de entrada do programa e a preparação do código intermédio e da tabela de símbolos.

```

TSyntabNode *func_main=globalSyntab.Search("main");
if(!func_main){Error(errMissingEntryPoint);}

syntabStack.PushSyntab(func_main->defn.routine.pSyntab);
icode=*(func_main->defn.routine.pIcode);
icode.Reset();

```

**Figura 4.9 - Ponto de entrada para geração do programa**

Depois das informações estarem preparadas começamos por emitir o prólogo do programa. No prólogo são reservadas e inicializadas as variáveis globais e é feita a chamada da função *main*.

Na Figura 4.10 é mostrada a ordem de geração do código, depois do prólogo do programa ser gerado. O passo seguinte é a geração do código que constitui a função *main*, de seguida o código de todas as outras funções, pela ordem que foram encontradas e por fim é gerado o epílogo do programa, terminando assim a geração de código. No fim do código estar

gerado podem existir saltos que não estejam resolvidos; este é o trabalho do *linker*: resolver os saltos para que o programa fique completamente funcional.

```
EmitProgramPrologue ();

EmitFunction (func_main);

for (int i=0; i<globalSymtab.NodeCount()-1; i++)
{
    TSymtabNode *func=globalSymtab.Get (i);
    if (func->defn.how == dcFunction && func != func_main)
    {
        symtabStack.PopSymtab ();
        symtabStack.PushSymtab (func->defn.routine.pSymtab);
        icode=* (func->defn.routine.pIcode);
        icode.Reset ();
        EmitFunction (func);
    }
}

EmitProgramEpilogue ();
```

**Figura 4.10 - Geração do código do programa fonte**

Os saltos que são necessários resolver são: os CALL de funções e os BRAI que são usados para saltar para as interrupções. Todos os outros saltos são resolvidos durante o processo de geração de código graças às instruções de saltos relativos ao PC presentes no ISA da arquitetura alvo.

Finalizado a resolução dos saltos, o código executável está pronto a ser escrito num ficheiro final que será posteriormente carregado para a memória de código do microprocessador alvo.

```
char buffer[20];
for (int i=0; i<MEM_CODE_SIZE; i++)
{
    sprintf (buffer, "%.8x", CodeMem[i]);
    PutLine (buffer);
    PutLine ();
}
PutLine ("0;");
```

**Figura 4.11 - Bloco de código que cria o ficheiro final com o código máquina**

Na Figura 4.11 é mostrado a bloco de código que cria o ficheiro final para a arquitetura alvo. Durante todo o processo de geração de código, este é guardado num vetor para mais facilmente ser manipulado. Apenas quando o processo de geração e resolução de salto é

finalizado é que o código é escrito num ficheiro para que possa ser colocado na memória da arquitetura alvo.

#### 4.1.3.3. Diferenciação dos *statements*

Na Figura 4.10 o método *EmitFunction* é o responsável por gerar o código que compõe a função. Neste código é necessário identificar o tipo de *statement* para gerar o código apropriado. Na linguagem C os *statements* podem começar pelos seguintes *tokens*:

- \*
- Identificador
- asm
- if
- while
- for
- switch
- return

No caso de ser um identificador é necessário distinguir o seu tipo, se é uma variável, apontador, função ou vetor. Dependendo do tipo do identificador, são chamados diferentes métodos para gerar o código correspondente ao identificador. Nos restantes *tokens* onde o *token* é uma palavra reservada da linguagem só há um método possível de ser chamado. Na Figura 4.12 é apresentado o excerto de código que distingue o tipo de *token* e chama o método correspondente dependendo do *token*.

```

void TCodeGenerator::EmitStatement(void)
{
    switch (token)
    {
        case tcStar:          EmitAssignPtr();          break;
        case tcIdentifier: {
            TSymtabNode *var=symtabStack.SearchAll(pToken->String());
            if(var->defn.how == dcVariable ||
                var->defn.how == dcPointer)
            {
                EmitAssignment(var);
            }
            else if(var->defn.how == dcArray) {EmitAssignArray(var);}
            else { EmitFunctionCall(var); }

            regFile.CleanRegFile();
        }break;
        case tcAsm:          EmitAsmStatement();        break;
        case tcIf:           EmitIF();                  break;
        case tcWhile:        EmitWHILE();               break;
        case tcFor:          EmitFOR();                 break;
        case tcSwitch:       EmitSWITCH();              break;
        case tcReturn:       EmitReturn();              break;
    }
}

```

**Figura 4.12 - Método que identifica o tipo de *statement* a gerar**

Os *statements* da linguagem podem ser constituídos por mais *statements* e por expressões aritméticas e lógicas. Uma expressão aritmética ou lógica não pode existir de forma isolada, tem de aparecer sempre associada a uma atribuição ou no campo da condição das estruturas de controlo de fluxo do programa.

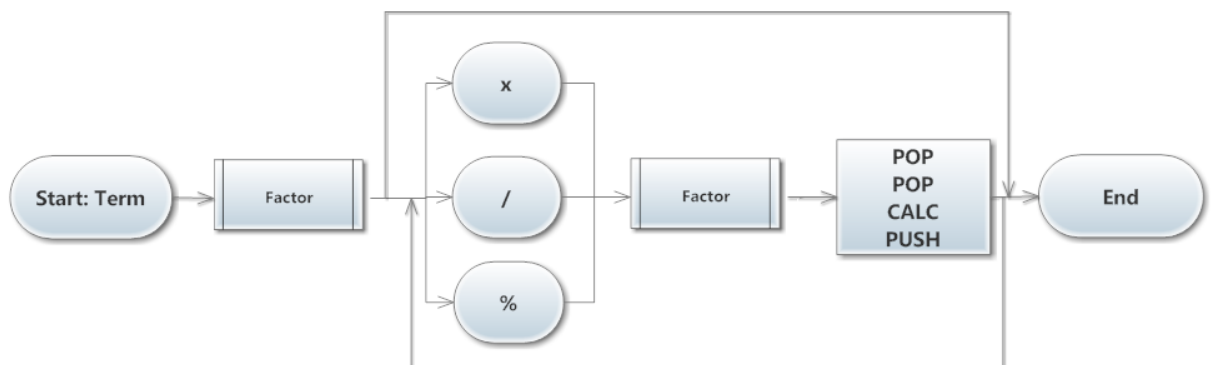
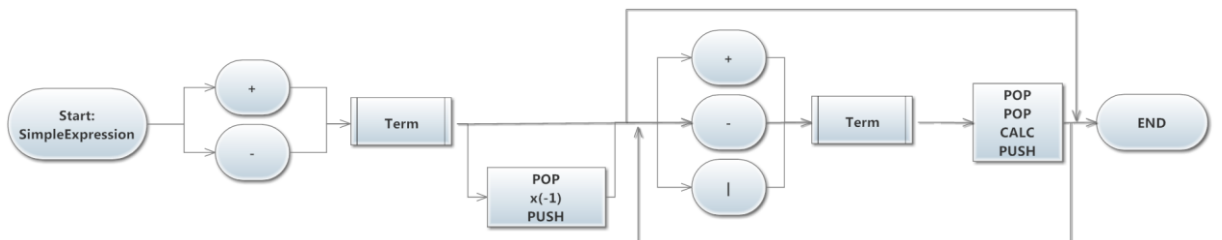
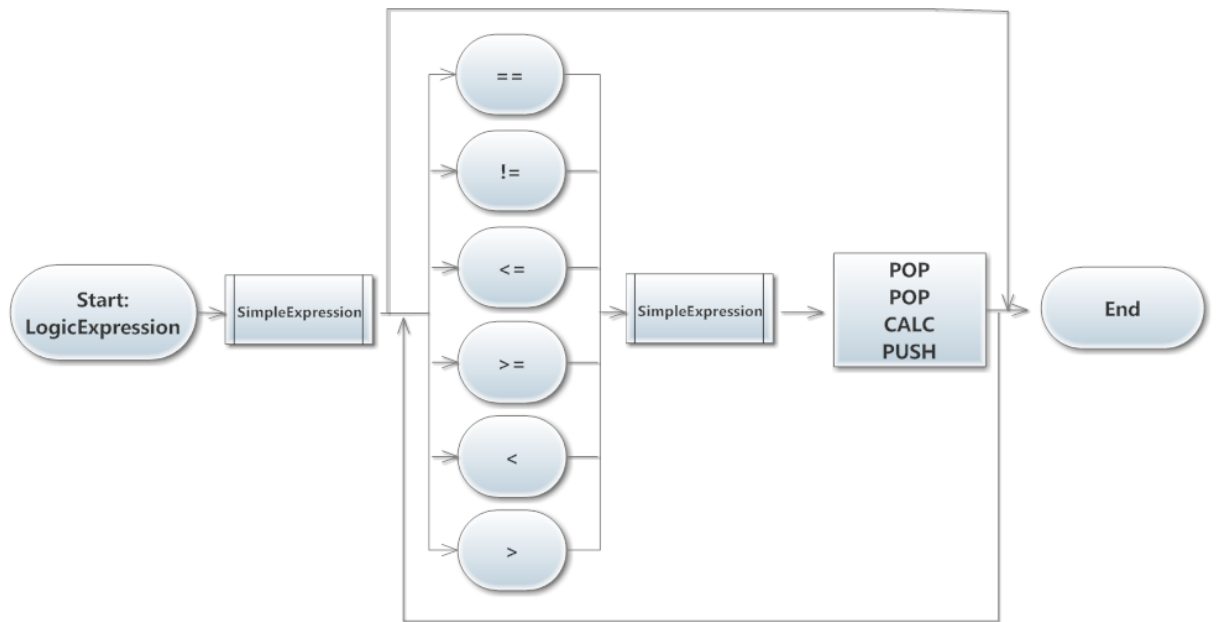
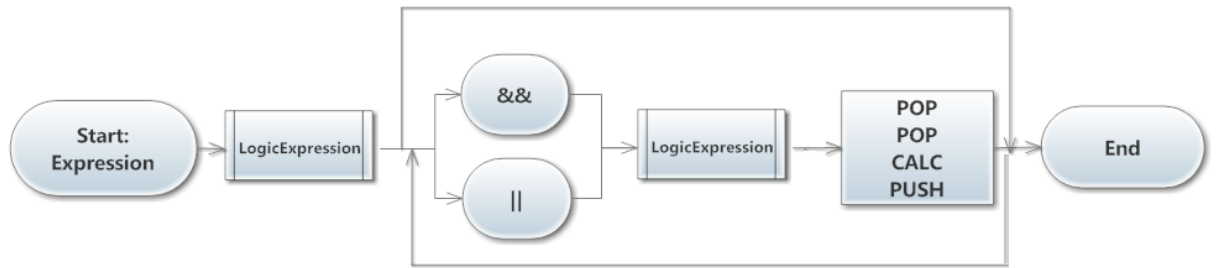
#### 4.1.3.4. Resolução de expressões

Para resolver as expressões aritméticas e lógicas presentes no código fonte, foi criada uma classe para esse efeito. Esta classe é baseada numa *stack* e em cinco métodos.

Os métodos vão sendo chamados recursivamente consoante a prioridade do operador que aparece na expressão, e os valores ou variáveis vão sendo colocados na *stack*.

As funções são chamadas de forma a resolver as operações de maior prioridade primeiro e o resultado de cada operação é colocado no topo da *stack* de forma a ser usado pela operação seguinte. Na Figura 4.13 são apresentados os diagramas sintáticos das cinco funções.





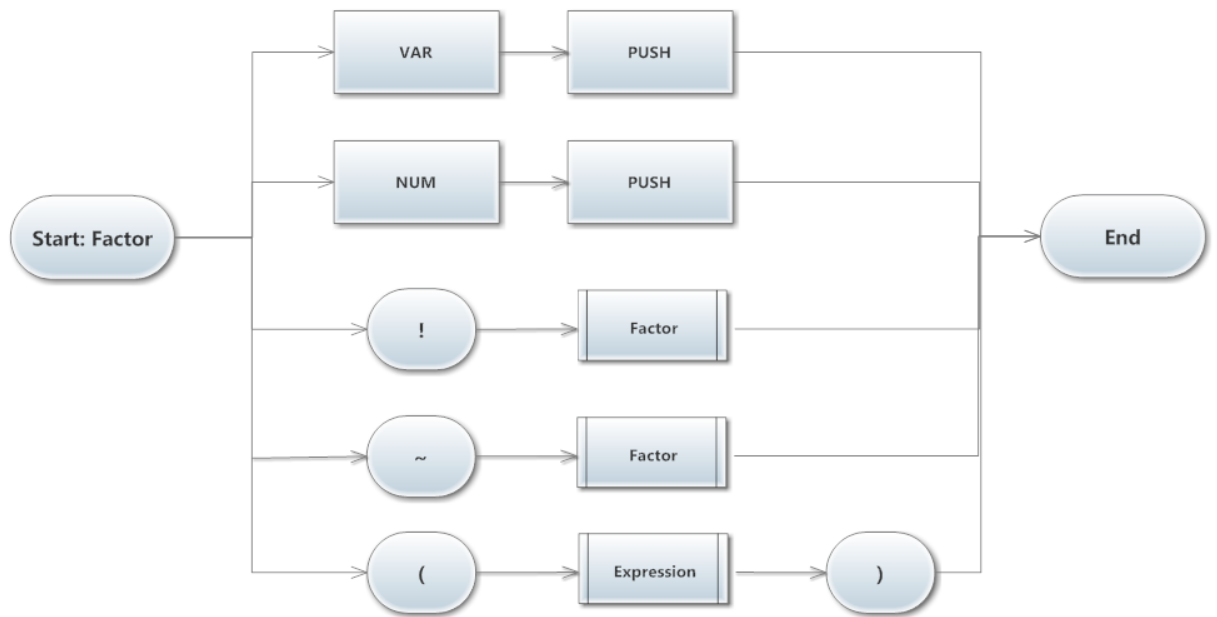


Figura 4.13 – Diagramas sintáticos que resolvem as expressões lógicas e aritméticas no compilador.

Por forma a elucidar o comportamento ilustrado na Figura 4.13, na Figura 4.14 é apresentado um exemplo onde é visível a evolução da *stack* durante a resolução de uma expressão aritmética simples.

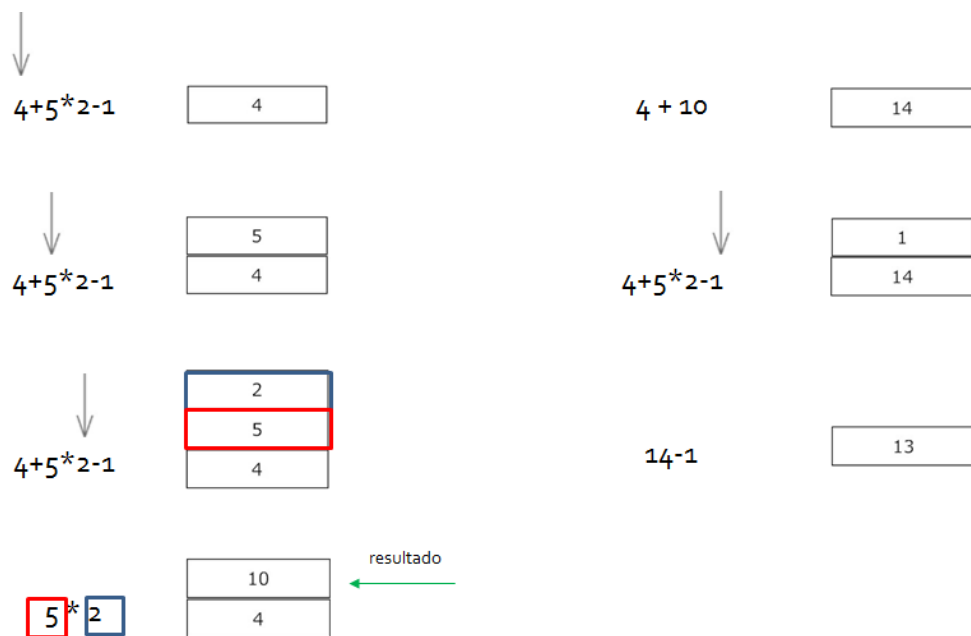


Figura 4.14 - Exemplo da resolução de uma expressão aritmética

A expressão apresentada na Figura 4.14 é a seguinte:  $4+5*2-1$ . Para resolver esta expressão segundo as regras aritméticas é necessário resolver a multiplicação à priori. De

seguida são resolvidas da esquerda para a direita as restantes operações. Seguindo o fluxo dos diagramas sintáticos apresentados na Figura 4.13, e tendo como ponto de entrada o diagrama *Expression*, a primeira ação a realizar é a colocação do número 4 na *stack*. Após isso ficam concluídas as ações no *Factor* regressando ao *Term*. No *Term* como o sinal encontrado é o sinal de soma, não há nada a fazer finalizando-se as ações no *Term*, voltando-se ao *SimpleExpression*. No *SimpleExpression* é encontrado o sinal de soma e então prossegue o fluxo. Aqui é chamado novamente o *Term* que por sua vez chama o *Factor* e é colocado o número 5 na *stack*. Ao regressar ao *Term*, desta vez é encontrado o sinal de multiplicação. É chamado o *Factor* novamente para colocar o próximo operando na *stack* e de seguida é realizada a operação de multiplicação e o resultado é colocado na *stack*. Finalizado o *Term* e como já existem dois elementos na *stack* é realizada a operação de soma e o resultado é colocado também na *stack*. A execução termina no *SimpleExpression* e retorna aos níveis anteriores onde não há nada a fazer. Como ainda existem elementos na expressão que não são terminadores, a execução volta ao ponto de partida em *Expression*, voltando a percorrer todo o fluxo de forma a resolver a subtração que falta, colocando o resultado final no topo da *stack*. Na Tabela 3 são apresentadas as prioridades dos operadores utilizando o conjunto de fluxogramas apresentados

**Tabela 3 - Prioridades dos operadores**

Nível	Operadores
1 (maior prioridade)	Operador de negação
2	Operadores de multiplicação, divisão e resto da divisão
3	Operadores de soma, subtração, OU lógico e E lógico
4	Operadores relacionais
5 (menor prioridade)	Operadores lógicos

Durante a resolução de expressões apenas podem ser resolvidos dois operandos de cada vez. Consultando o ISA da arquitetura alvo, é possível ver que apenas existe suporte em *hardware* para as operações lógicas ao bit e para as operações de soma e de subtração. Para conseguir dar suporte às operações de multiplicação, divisão e módulo da divisão presentes na linguagem C é necessário agrupar um conjunto de instruções de forma a conseguir o comportamento das operações pretendidas. A operação de multiplicação pode ser obtida à custa de várias somas.

Quando é encontrada uma operação com dois operandos é necessário proceder à sua resolução. Para determinadas operações como soma e subtração existe suporte em *hardware* (a lista de operações suportadas em *hardware* pode ser consultada no Anexo A onde é mostrado o *instruction set* da arquitetura alvo).

Quando uma operação não tem uma instrução em *hardware* para executá-la é necessário agrupar um conjunto de instruções em *software* para que a operação seja possível.

A arquitetura alvo deste trabalho não possui instruções em *hardware* para resolver multiplicações, divisões e módulo da divisão, que são operações suportadas pela linguagem C. Para contornar a falta destas instruções em *hardware* foi implementado estas operações à custa de operações presentes no *instruction set*. A multiplicação foi implementada à custa de várias operações de soma e a divisão e o módulo da divisão foram implementados à custa de várias subtrações. Na Figura 4.15 é apresentado o fluxograma para obter a operação de multiplicação à custa de diversas somas.

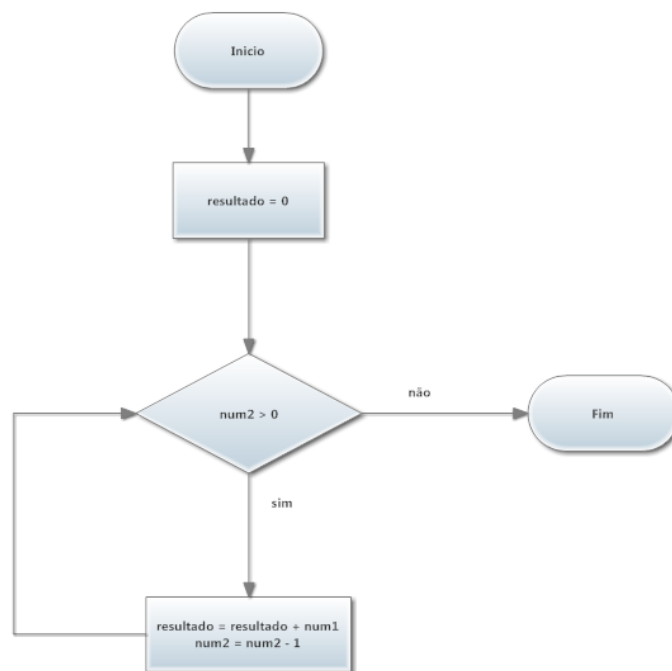


Figura 4.15 - Algoritmo para resolver uma multiplicação à custa de várias somas

Assumindo a forma  $resultado = num1 \text{ op } num2$ , onde *resultado* é o resultado da operação, *num1* e *num2* são os operandos e *op* é a operação a realizar.

Para implementar a multiplicação soma-se a variável *num1* tantas vezes quanto o valor presente na variável *num2*. Este algoritmo trás alguns problemas associados, o primeiro problema tem a ver com a *performance*, pois quanto maior for o valor na variável *num2* maior

será o número de iterações que o processador terá de executar, ou seja para multiplicações de números grandes o tempo de execução será muito grande. O segundo problema prende-se com o determinismo, pois o tempo de execução desta operação depende do valor das variáveis, não sendo assim fixo e previsível.

Para implementar a divisão é subtraída a variável *num2* à variável *num1* e conta-se o número de subtrações possíveis de fazer sem que a variável *num1* atinja um número negativo. O número de vezes que for possível efetuar esta operação é o resultado da divisão. Este algoritmo também apresenta os problemas de performance e de determinismo como o algoritmo da multiplicação: estes problemas agravam-se quando a diferença de valores entre o dividendo e o divisor aumenta.

Para a operação de resto da divisão o algoritmo é semelhante ao da divisão. A diferença prende-se na variável de resultado que em vez de ser a contagem do número de vezes que é possível subtrair, o resultado é o valor presente em *num1* quando este for inferior ao valor contido em *num2*. Os problemas de *performance* e determinismo, e a forma como se agravam são os mesmos do algoritmo da divisão. Os fluxogramas das operações de divisão e resto da divisão podem ser consultados no Anexo E.

#### 4.1.3.5. Números reais

O suporte para números reais é dado através da representação dos números de vírgula flutuante seguindo o *standard* IEEE 754. Na presente dissertação apenas é suportado o formato de precisão simples. Este formato utiliza 32-bits para representar o número, onde o *bit* 31 representa o sinal, o intervalo entre o *bit* 30 e o *bit* 23 representa o expoente e os restantes *bits* representam a mantissa. A mantissa é a representação da parte fracionária do número.

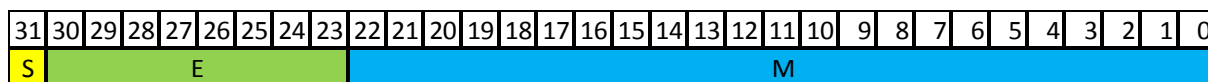


Figura 4.16 – Representação dos campos que compõem o número.

A construção do número é apresentada na Figura 4.17 e a sua representação em memória pode ser vista na Figura 4.16

$$S * M * 2^E$$

Figura 4.17 - Configuração do número seguindo a norma do *standard* IEEE 754

A arquitetura alvo não possui instruções para realizar diretamente operações com números em vírgula flutuante. Para dar suporte a estas operações é necessário implementar estas operações em *software*.

Para realizar qualquer operação aritmética é necessário dividir a *word* nos respetivos campos e tratá-los individualmente. Na Figura 4.18 são apresentadas as formas como os campos são tratados, nas quatro operações aritméticas básicas.

$$S1 * M1 * 2^{E1} \pm S2 * M2 * 2^{E2}$$

$$S1 * (M1 \pm M2) * 2^{E1} \quad \text{Se } E1 > E2$$

$$S1 * M1 * 2^{E1} * S2 * M2 * 2^{E2}$$

$$(S1 * S2) * (M1 * M2) * 2^{E1+E2}$$

$$S1 * M1 * 2^{E1} / S2 * M2 * 2^{E2}$$

$$(S1 * S2) * (M1/M2) * 2^{E1-E2}$$

Figura 4.18 – Fórmula matemática para resolução de expressões aritméticas envolvendo números com vírgula flutuante.

Para implementar estas operações é necessário um grande número de instruções. Para minimizar a quantidade de código estas operações foram implementadas em código *assembly* e introduzidas no compilador sob a forma de bibliotecas que são chamadas quando uma operação deste tipo necessita de resolução. As bibliotecas *assembly* para as quatro operações podem ser consultadas no Anexo F.

#### 4.1.3.6. Suporte para manipulação de periféricos

Nesta secção serão apresentadas as APIs para manipular os periféricos do microprocessador. Os periféricos do microprocessador estão interligados por um barramento, e cada periférico tem um *id* associado. Para enviar e receber dados dos periféricos, foram

criadas duas APIs para o efeito. Na Figura 4.19 são apresentadas as APIs de manipulação dos periféricos e os seus parâmetros.

```
writeBus (val, dvicID, IRQ0_prio);  
readBus (var, dvicID, IRQ0_prio);
```

**Figura 4.19 - API's de manipulação dos periféricos**

Os parâmetros que as APIs recebem são idênticos. O primeiro parâmetro é a variável com o valor a escrever no periférico no caso da API de escrita, ou caso seja a API de leitura, o primeiro parâmetro é a variável que vai receber o valor lido do periférico. O segundo parâmetro é uma constante que indica o periférico sobre o qual se pretende realizar a ação. O terceiro e último parâmetro indica o registo interno do periférico sobre o qual se pretende ler ou escrever.

#### 4.1.3.7. Suporte a rotinas de serviço à interrupção

O suporte a rotinas de serviço à interrupção é garantido por funções de código C com nomes identificativos da interrupção. Estes nomes são inalteráveis, pois estas funções necessitam de um tratamento especial por parte do compilador e do *linker*. No caso do compilador o prólogo e o epílogo da função de atendimento à interrupção é gerado de forma diferente das demais. Nas funções convencionais são salvos todos os registos utilizados até ao momento da chamada da função, no caso da interrupção como não é possível saber o seu momento de chamada é necessário guardar todos os registos que são utilizados na função de atendimento à interrupção. Por parte do *linker* é necessário resolver o valor de salto a colocar na *vector table*. Este valor deve corresponder ao endereço onde a interrupção ficará alojada na memória, pois a localização é atribuída como uma função convencional.

```
void SYSTICK_ISR_vect(void)           0x0034    PUSH    r1  
{                                     0x0035    LWI    r1,r14,0  
    tempo++;                           0x0036    ADDI   r1,r1,1  
}                                       0x0037    SWI    r1,r14,0  
                                       0x0038    POP    r1  
                                       0x0039    RETI
```

**Figura 4.20 - Código gerado a partir de uma função de serviço à interrupção**

Na Figura 4.20 é possível ver no código *assembly* que apenas o registo r1 é que sofre uma alteração de valor durante a execução da rotina de serviço à interrupção. Assim apenas foi salvo o valor deste registo antes da execução deste bloco de código. À saída foi restaurado o valor do registo com a instrução POP. Na Tabela 4 são apresentadas todas as fontes externas ao processador capazes de gerar uma interrupção ao seu normal fluxo de execução.

**Tabela 4 - Lista de interrupções suportadas pelo periférico DVIC**

Posição	Nome do vetor de interrupção	Descrição
0x01	SYSTICK_ISR_vect	Interrupção gerada a pelo periférico <i>Systick</i>
0x02	INT4_ISR_vect	Interrupção gerada por um pino do periférico de <i>input/output</i>
0x03	TIMER0_ISR_vect	Interrupção gerada pelo <i>Timer 0</i> do vetor de <i>timers</i>
0x04	TIMER1_ISR_vect	Interrupção gerada pelo <i>Timer 1</i> do vetor de <i>timers</i>
0x05	TIMER2_ISR_vect	Interrupção gerada pelo <i>Timer 2</i> do vetor de <i>timers</i>
0x06	TIMER3_ISR_vect	Interrupção gerada pelo <i>Timer 3</i> do vetor de <i>timers</i>
0x07	INT0_ISR_vect	Interrupção gerada por um pino do periférico de <i>input/output</i>
0x08	INT1_ISR_vect	Interrupção gerada por um pino do periférico de <i>input/output</i>
0x09	INT2_ISR_vect	Interrupção gerada por um pino do periférico de <i>input/output</i>
0x0A	INT3_ISR_vect	Interrupção gerada por um pino do periférico de <i>input/output</i>
0x0B	USART_TX_ISR_vect	Interrupção gerada pelo periférico UART quando termina o envio de um caractere
0x0C	USART_RX_ISR_vect	Interrupção gerada pelo periférico UART quando recebe um caractere



Uma característica importante do controlador de interrupções da arquitetura alvo, é o facto de este permitir o desencadeamento de interrupções por *software*, para isso basta habilitar um registo específico da interrupção.

## 4.2. *Microkernel*

O *microkernel* foi implementado sob a forma de bibliotecas que têm de ser incorporadas e compiladas em conjunto com a aplicação. Para implementar este *microkernel* foram apenas criados dois ficheiros de interface do tipo .h e dois ficheiros de código do tipo .c. Nos ficheiros task.h e task.c que são os dois ficheiros onde o código é independente da máquina, estão declaradas e implementadas as APIs de interface com a aplicação para gerir as tarefas e estão declaradas as estruturas que guardam as informações das tarefas.

### 4.2.1. Tarefas

A execução intercalada de diversas tarefas implica a virtualização dos recursos do sistema. O *microkernel* deve salvar o estado do processador em que cada tarefa deixa a execução e restaurá-lo adequadamente quando a tarefa voltar à execução. A informação guardada quando uma tarefa é retirada de execução é denominada de contexto da tarefa. O contexto da tarefa consiste na informação de todos os registos do processador em qualquer instante de execução. O contexto da tarefa é guardado imediatamente antes de a tarefa ser retirada de execução, para que quando esta voltar a executar, continue do mesmo ponto onde se encontrava. Para além do contexto, o *microkernel* necessita de informação adicional para gestão de tarefas.

A estrutura que guarda a informação adicional requerida pelo *microkernel* para a gestão das tarefas denomina-se TCB (*Task Control Block*), e a sua declaração pode ser vista na Figura 4.21.

```
struct tcb
{
    int id;
    int prio;
    int stack;
    int sleepTicks;
    int estado;
};
```

**Figura 4.21 - TCB do *microkernel***

Para cada tarefa é guardado um identificador único, que é dado à tarefa pela API de criação de tarefas. É também guardado o valor de prioridade (quanto maior, mais tempo de CPU a tarefa obtém se precisar), que é atribuído à tarefa pelo programador e é utilizado pelo escalonador para gerir quem obtém acesso ao processador. É guardado também um endereço para a sua própria *stack*, onde são guardadas variáveis locais, endereços de retorno de funções e o contexto da tarefa quando esta é retirada de execução. Caso a tarefa necessite de um *delay*, este é guardado no campo *sleepTicks*, que guarda o número de *systicks* que a tarefa deve permanecer inativa: se o valor for zero significa que a tarefa está pronta para execução. Por fim é guardado o estado da tarefa. Existem quatro estados possíveis para as tarefas: *running*, *ready*, *suspended* e *delayed*. No estado de *running* significa que a tarefa está no momento a usufruir do processador. O estado de *ready* significa que a tarefa está pronta a assumir o controlo do processador a partir do momento que o escalonador assim o indicar. Quando a tarefa está no estado de *suspended*, significa que a mesma está a aguardar que um evento interno ou externo ocorra. E por fim no estado de *delayed*, a tarefa aguarda que um determinado número de *systicks* ocorra para que esta possa voltar a executar.

#### **4.2.2. Escalonador**

O *core* do *microkernel* é o seu escalonador. É o escalonador que gere a informação das tarefas que constituem o ambiente de execução e gere o acesso destas ao processador, de acordo com a estratégia de escalonamento implementada. O escalonador do presente *microkernel* usa uma estratégia denominada de *fixed priority preemptive*. Com esta estratégia o escalonador garante que em cada ponto de escalonamento a tarefa escolhida para executar é a tarefa com maior prioridade, desde que esta esteja pronta para executar. Na Figura 4.22 é apresentado o excerto de código que implementa o escalonador do *microkernel*.

```

int getNextTask(void)
{
    int i, maior=255;

    for(i=0;i<maxTasks;i++)
    {
        if(List[i].estado == ready)
        {
            if(maior == 255){maior=i;}

            if(List[i].prio > List[maior].prio)
            {
                maior = i;
            }
        }
    }

    return maior;
}

```

Figura 4.22 - Código que implementa o algoritmo de escalonamento baseado em prioridades

### 4.2.3. Comutação de contexto

Para suportar a execução intercalada de tarefas é necessário parar a tarefa que está a utilizar o processador no momento salvar o seu contexto, escolher uma nova tarefa para utilizar o processador e restaurar o contexto da nova tarefa. Este processo é denominado de comutação de contexto. Na arquitetura alvo o contexto do processador é dado pelo *Program Counter* e pelos trinta e dois registos de uso genérico. Uma comutação de contexto ocorre sempre que é retirada a tarefa em execução e é colocada uma nova tarefa, que pode acontecer sempre que existe uma interrupção do *systick* ou quando uma tarefa entra em estado de *delay* ou de *suspended*. Na Figura 4.23 é apresentado o pseudo código da interrupção do *systick* que é um ponto onde ocorre comutação de contexto.

```

TickISR()
{
    TaskContextSave()

    UpdateDelayedTaskTickCount()
    ChooseNewTask()

    TaskContextRestore()
    ReturnFromISR()
}

```

Figura 4.23 - Pseudo código da ISR do *systick* onde ocorre comutação de contexto

Para além da interrupção do *systick* também pode acontecer comutação de contexto quando uma tarefa entra em estado de *delayed* ou de *suspended*. Por forma a facilitar a

implementação das APIs de *delay* e de *suspend* a estratégia adotada passa por forçar uma interrupção por *software* e na ISR do *systick* no ponto *UpdateDelayedTaskTickCount()* verificar se a interrupção foi causada por *software* ou *hardware*. Se foi por *hardware* então atualizamos o *tickcount*, caso contrário ignoramos este *update*.

### 4.3. Agnosticismo

Na terceira e última parte da presente dissertação, foi introduzida uma nova fase de compilação com o objetivo de tornar o compilador independente do sistema operativo alvo. A aplicação deve ser escrita utilizando APIs genéricas para que o compilador possa interpretar estas APIs e substituí-las pelas correspondentes do sistema operativo alvo pretendido.

Como foi referido na secção 3.4, esta ferramenta deve ser flexível ao ponto de permitir adicionar ou alterar a lista de sistemas operativos alvo. Para conseguir isso a informação necessária deve estar disponível em ficheiros externos à ferramenta. Com o objetivo de tornar a ferramenta flexível foi criado um repositório onde está toda a informação relativa às APIs genéricas e a informação necessária de cada sistema operativo alvo. Na Figura 4.24 é apresentado o esquema do repositório e o seu conteúdo.

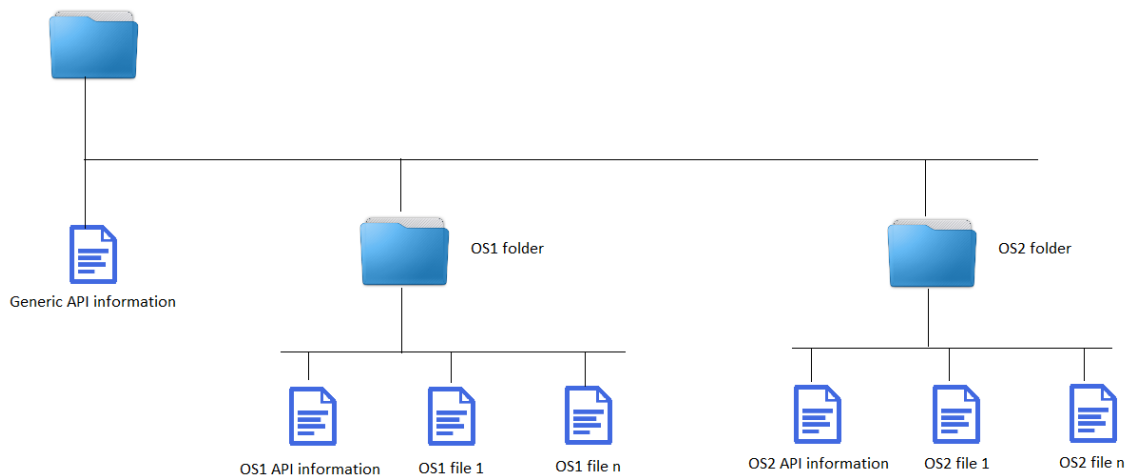


Figura 4.24 - Esquema do repositório do agnóstico

O repositório tem uma pasta principal onde contém o ficheiro denominado *genericAPI.data*, este ficheiro contém as informações sobre as APIs genéricas. As informações contidas são, nome da API genérica, os parâmetros genéricos e o valor dos parâmetros por defeito, caso não seja providenciado um valor durante o desenvolvimento do

código fonte da aplicação. Na Figura 4.25 é mostrado o aspeto do ficheiro *genericAPI.data* que contém as informações das APIs genéricas.

```
genericTaskCreate(priority=1,stackSize=150)
genericTaskDelay(id=0,ticks=100)
genericTaskSuspend(id=0)
```

**Figura 4.25 - Aspeto do ficheiro genericAPI.data**

Para além do ficheiro com as informações das APIs genéricas, também na pasta principal são criadas diversas subpastas uma para cada sistema operativo alvo. No interior da subpasta do sistema operativo deve existir um ficheiro que define as informações do sistema operativo alvo. Este ficheiro tem de ser nomeado com o mesmo nome da pasta acrescido de *.interface*. A título de exemplo se o sistema operativo se chamar MYRTOS a pasta deverá ser nomeada de MYRTOS e o ficheiro de interface no interior desta pasta deverá ser chamado de MYRTOS.interface. Além do ficheiro de interface estão também contidos na pasta do sistema operativo os ficheiros que o compõem. Na Figura 4.26 é apresentado o aspeto do ficheiro que contém a informação das APIs de um sistema operativo alvo.

```
genericTaskCreate::vTaskCreate(priority,stackSize)
genericTaskDelay::vTaskDelay(ticks)
genericTaskSuspend::vTaskSuspend()

files
task.h
port.h
usart.h
task.c
port.c
usart.c
```

**Figura 4.26 - Aspeto do ficheiro .interface de um sistema operativo alvo**

Neste ficheiro é feito o *mapping* entre as APIs genéricas e as APIs do sistema operativo alvo. É explicitado qual a API genérica que corresponde à API específica e que parâmetros da API genérica são passados à API do sistema operativo alvo. Além das informações das APIs também é especificado neste ficheiro a lista de ficheiros que compõem o sistema operativo alvo, e que serão incluídos automaticamente durante a compilação da aplicação.



## Capítulo 5

### RESULTADOS

---

Neste capítulo serão apresentados os resultados desta dissertação. Numa fase inicial são enumeradas e explicadas as várias plataformas de teste. Depois da apresentação das plataformas são efetuados testes individuais a cada parte do trabalho (compilador, *microkernel* e ao agnosticismo do compilador). Por fim são feitos testes de integração onde se pode ver o funcionamento conjunto de todas as ferramentas no apoio ao desenvolvimento de aplicações.

#### 5.1. *Disassembler*

O *disassembler* é um pequeno *software* que converte um código objeto em código *assembly*. Este utilitário é muito útil nas fases iniciais do compilador, uma vez que permite verificar se o código gerado corresponde ao pretendido, em termos de instruções e de registos.

Para obter o código *assembly* é necessário compilar a aplicação com o compilador C desenvolvido nesta dissertação, executando-se seguidamente o utilitário *disassembler* com o ficheiro de output do compilador a ser o *input* do *disassembler*. Assim será produzido o código *assembly* da aplicação compilada. O output do *disassembler* é um ficheiro de texto com o código *assembly* estruturado e com uma instrução por linha para facilitar a leitura.

Na Figura 5.1 mostra-se o processo de obtenção de um código *assembly* a partir de uma aplicação escrita em C.



Figura 5.1 – Processo de obtenção do código *assembly* a partir do código C

A apresentação do código gerado pelo compilador ao longo do presente capítulo será sempre realizada recorrendo ao código *assembly*, e nunca ao código máquina gerado diretamente a partir do compilador. A apresentação de código será apenas feita para aplicações muito simples onde se pretenda mostrar operações aritméticas simples ou acessos à memória de dados. No subcapítulo seguinte são apresentadas as plataformas de teste utilizadas para a validação do código compilado.

## 5.2. Plataformas de teste

Nesta secção serão apresentadas as duas plataformas de teste utilizadas para obter e validar os resultados experimentais do trabalho desenvolvido.

### 5.2.1. Simulador

O simulador é um *software* que foi desenvolvido propositadamente para auxiliar na depuração de erros durante o desenvolvimento do compilador. Este simulador foi



desenvolvido para o sistema operativo Windows, com o objetivo de recriar o comportamento do microprocessador alvo. Assim com este simulador é possível verificar o estado do processador a cada instrução executada. Permite também parar a execução ao fim de um determinado número de instruções executadas e é possível ver os valores contidos em cada registo em qualquer instante da simulação e os valores de todas as posições da memória RAM. Pelo facto de o microprocessador alvo estar implementado sobre tecnologia FPGA e a memória de código estar *on-chip*, o processo de síntese de *hardware* e atualização das memórias com o novo código da aplicação é uma tarefa morosa. Assim antes de executar o *software* gerado pelo compilador diretamente no microprocessador, são feitos testes no simulador para despistar possíveis erros, quer de comportamento quer de compilação.

```

PC = 34
MOUI      r14,32

r0 = 0      r1 = 0      r2 = 0      r3 = 0
r4 = 0      r5 = 0      r6 = 0      r7 = 0
r8 = 0      r9 = 0      r10 = 0     r11 = 0
r12 = 0     r13 = 0     r14 = 32    r15 = 0
r16 = 0     r17 = 0     r18 = 0     r19 = 0
r20 = 0     r21 = 0     r22 = 0     r23 = 0
r24 = 0     r25 = 0     r26 = 0     r27 = 0
r28 = 0     r29 = 0     r30 = 0     r31 = 4095

ram[4078] = 0
ram[4079] = 0
ram[4080] = 0
ram[4081] = 0
ram[4082] = 0
ram[4083] = 0
ram[4084] = 0
ram[4085] = 0
ram[4086] = 0
ram[4087] = 0
ram[4088] = 0
ram[4089] = 0
ram[4090] = 0
ram[4091] = 0
ram[4092] = 0
ram[4093] = 0
ram[4094] = 0
ram[4095] = 0

```

Figura 5.2 - *Layout* inicial do simulador

Na Figura 5.2 é mostrado o *layout* do simulador durante uma depuração passo a passo. É possível visualizar-se o endereço da posição de memória da instrução a executar, a instrução *assembly* a executar no momento, os valores de todos os registos do processador e as posições finais da memória RAM, que no caso da arquitetura alvo representa a *stack* do processador. Numa fase mais avançada do trabalho a depuração passo a passo deixou de ser viável devido ao tamanho do *software*. Assim a interface do simulador foi alterada para

mostrar apenas o *output* dado pela UART do microprocessador. A Figura 5.3 ilustra um exemplo do *layout* do simulador a correr uma aplicação com o *microkernel* onde cada tarefa imprime o seu número através da USART.

```
Start scheduler...
Task1
Task2
Task3
Task1
Task2
Task3
Task1
Task2
Task3
Task1
Task2
Task3
```

Figura 5.3 - *Layout* final do simulador

### 5.2.2. Plataforma alvo

A arquitetura alvo é um *softcore* desenvolvido para a plataforma FPGA XC5VLX110T do fabricante Xilinx. Os testes finais ao trabalho desenvolvido foram feitos nesta plataforma sendo esta responsável por validar o correto funcionamento do trabalho. Na Figura 5.4 é apresentada a plataforma XUPV5 onde será sintetizado o novo *softcore*.

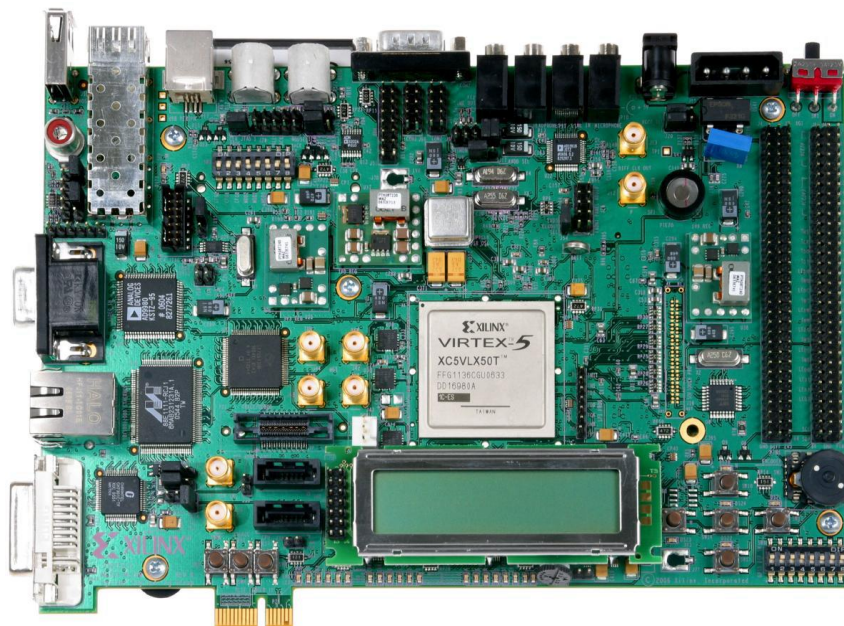


Figura 5.4 - Plataforma onde será alojado o *softcore*

A plataforma de desenvolvimento da Xilinx possui uma FPGA de alto desempenho denominada XC5VLX110T da família Virtex-5. Esta placa encontra-se abrangida pelo Xilinx University Program, e é suficientemente poderosa para ser utilizada em projetos de investigação. As capacidades desta placa conseguem cobrir uma vasta gama de aplicações.

A placa possui bastantes recursos, tem um amplo conjunto de periféricos que podem ser usados na construção de diversos sistemas. A respetiva configuração e programação pode ser efetuada de duas formas, através da *compact flash card*, ou então através do cabo de programação USB/JTAG. Quanto às memórias, a placa oferece variados recursos, como uma SDRAM, ZBT SRAM, *compact flash card*, *strataflash* e uma *flash* não volátil.

Um dos problemas da utilização de plataformas FPGA é o elevado tempo de síntese do sistema. No desenvolvimento do *softcore* o tempo de sintetização era superior a 20 minutos o que não tornava a plataforma atrativa para fazer testes constantemente, e motivou o desenvolvimento do simulador acima descrito.

### **5.3. Testes**

Os testes elaborados estão divididos em três grupos, nomeadamente: (i) testes ao compilador; (ii) testes ao *microkernel*; (iii) testes ao agnosticismo do compilador que pela natureza desta ferramenta também representa os testes de integração finais. Sempre que se pretenda visualizar o código gerado pelo compilador, o teste realizado apresentará os resultados sob a forma de código *assembly*, gerado através do *disassembler* apresentado no subcapítulo 5.1. Para os testes onde o objetivo é visualizar os valores contidos nos registos ou em posições de memória, o resultado é apresentado a partir do *output* do simulador descrito no ponto 5.2.1. Por fim, nos testes onde se pretenda perceber o comportamento da aplicação, o resultado é apresentado através do *output* da UART do microprocessador, ou através dos led's conectados ao porto de *output* do microprocessador.

#### **5.3.1. Validação das funcionalidades do compilador**

Os testes descritos nesta secção consistem na elaboração de diversos casos de teste de aplicações simples, em que os resultados esperados sejam facilmente perceptíveis. Estes testes foram realizados para validar o correto funcionamento do compilador. O significado e o comportamento de cada instrução *assembly* podem ser consultados no Anexo A.

### 5.3.1.1. Entrada na função *main* e expressões simples

Este teste permite demonstrar o código gerado pelo compilador para a preparação e entrada da função *main* que é o ponto de entrada em programas escritos em linguagem de programação C. Na Figura 5.5 é apresentado no canto superior esquerdo uma pequena aplicação escrita em linguagem C que realiza duas operações aritméticas. À direita é apresentado o código *assembly* obtido a partir do ficheiro com código objeto produzido pelo compilador. No canto inferior esquerdo é apresentado os valores da memória de dados produzidos pelo simulador.

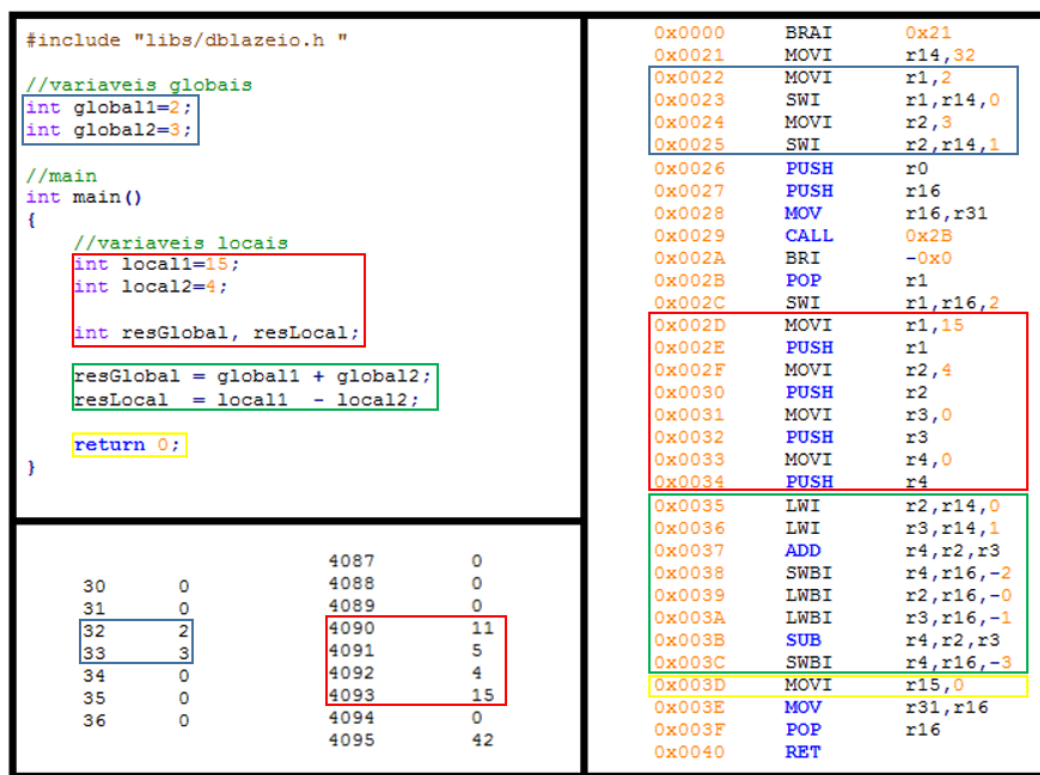


Figura 5.5 - Teste de entrada na função *main* e expressões simples

Analisando o código *assembly* presente na Figura 5.5 é possível observar que a aplicação não começa a executar imediatamente na função *main*. Antes de executar o código presente na função *main* é necessário preparar um ambiente de execução com os valores corretos das variáveis a utilizar. No retângulo azul é possível observar a declaração e inicialização das duas variáveis globais. No retângulo encarnado é possível ver a declaração e inicialização das variáveis locais da função *main*. No canto inferior esquerdo onde é possível ver as posições de memória associadas às variáveis os valores apresentados são os valores

após a execução total do programa, ou seja as expressões já foram resolvidas e os valores associados às respectivas variáveis. No retângulo verde é possível ver a correspondência entre as expressões escritas em código C e o código *assembly* que resolve essas mesmas expressões.

### 5.3.1.2. Chamadas de funções

Este teste permite validar o funcionamento de chamadas de funções escritas em linguagem C. Antes de chamar a função é necessário preparar a uma nova *frame* na *stack* para a função a invocar. Este teste demonstra a implementação da estrutura de uma *frame* da *stack* descrito na secção 3.2.4. Analisando o código *assembly* apresentado na Figura 5.6, a chamada da função começa na instrução com o endereço 0x2E. Aqui é criado um *placeholder* para o endereço de retorno, esta ação é necessária pois só se obtém o endereço de retorno quando a instrução CALL for executada e o *hardware* colocar o endereço no topo da *stack*. Depois de se ter o *placeholder*, guarda-se o apontador para a *frame* anterior da *stack*. De seguida procede-se à salvaguarda dos parâmetros da função na *stack*. Antes da chamada da função falta apenas mudar o apontador para a *frame* atual e por fim procede-se à chamada da função. Quando se entra na função (endereço 0x3F) a primeira ação a realizar é colocar o valor de retorno da função no local apropriado da *stack*, para isso retira-se da *stack* o último valor colocado que é o endereço de retorno introduzido na *stack* pela instrução CALL e guarda-se o mesmo no *placeholder* criado anteriormente. Na Figura 5.6 é mostrado o código C da aplicação, o código *assembly* obtido a partir do código gerado pelo compilador e os valores da memória após a execução da aplicação.

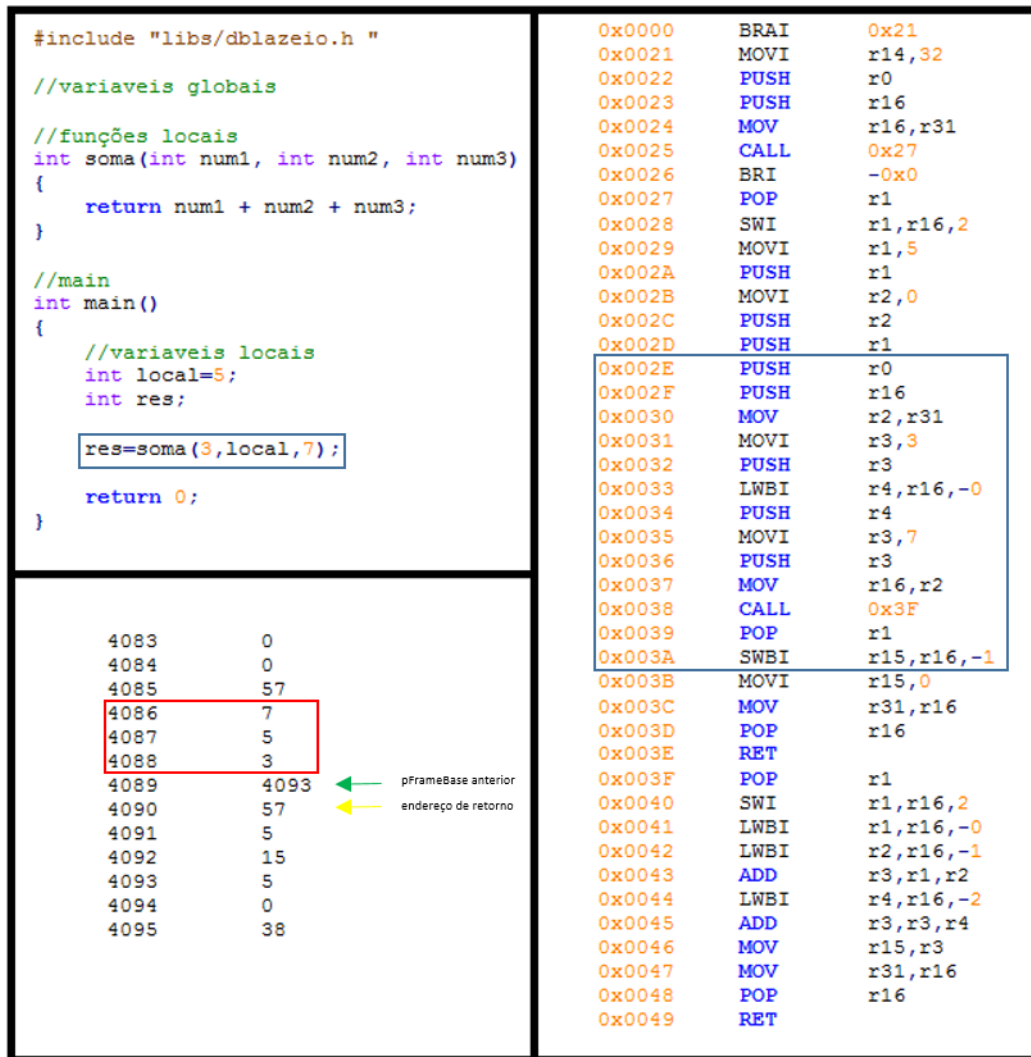


Figura 5.6 - Teste de chamadas de funções

### 5.3.1.3. Estruturas de controlo de fluxo do programa

Para validar as estruturas de controlo de fluxo foi escolhido o algoritmo de ordenação de vetores *bubble sort*. Neste teste é possível demonstrar o correto funcionamento de três estruturas de controlo de fluxo do programa (*for*, *while* e *if*). Também é possível verificar o correto funcionamento de vetores como estruturas de dados e o encadeamento de diversas estruturas de controlo de fluxo do programa. Neste exemplo é criado um vetor de dez posições que é inicializado com um conjunto de números desordenados. De seguida é aplicado o algoritmo *bubble sort* ao vetor e é esperado que os números no vetor fiquem ordenados de forma crescente na estrutura de dados.

<pre> #include "libs/dblazeio.h "  //constantes const int arraySize=10;  //main int main() {     //variaveis locais     int lista[arraySize];     int i,j,aux;      lista[0]=7;     lista[1]=5;     lista[2]=9;     lista[3]=1;     lista[4]=11;     lista[5]=15;     lista[6]=66;     lista[7]=33;     lista[8]=27;     lista[9]=47;      i=0;     while(i&lt;arraySize-1)     {         for(j=i+1;j&lt;arraySize;j++)         {             if(lista[i] &gt; lista[j])             {                 aux=lista[i];                 lista[i]=lista[j];                 lista[j]=aux;             }         }         i++;     }      return 0; } </pre>	<pre> 30 0 31 0 32 1 33 5 34 7 35 9 36 11 37 15 38 27 39 33 40 47 41 66 42 0 43 0 44 0 45 0 </pre>
--	--

Figura 5.7 - Implementação do algoritmo *bubble sort* e resultado após execução

Na Figura 5.7 é apresentado o código C da inicialização do vetor e posterior aplicação do algoritmo *bubble sort*. À direita na imagem é apresentado os valores em memória do vetor após a execução do programa, onde é possível ver os valores contidos no vetor, ordenados de forma crescente.

#### 5.3.1.4. Periféricos

Para validar o suporte aos periféricos, foi desenvolvida uma pequena aplicação onde é configurada uma interrupção externa. Quando esta é desencadeada é escrito um valor no periférico de *output* do microprocessador. O código C da aplicação pode ser visualizado na Figura 5.8.

```

#include "libs/dblazeio.h "

//variaveis globais
int output=0xAA;

//main
int main()
{
    //variaveis locais
    int val=1;

    //configura o DVIC
    writeBus (val,dvicID,IRQ0_enable);
    writeBus (val,dvicID,IRQ0_prio);

    writeBus (output,parioId,write);
    while (1) {}

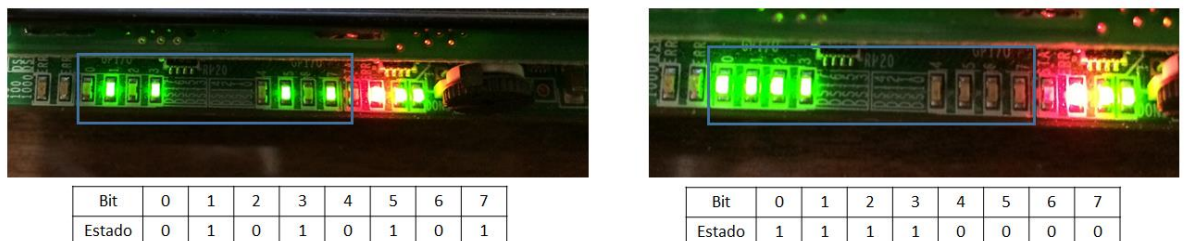
    return 0;
}

void INT0_ISR_vect(void)
{
    output=0x0F;
    writeBus (output,parioId,write);
}

```

**Figura 5.8 - Código C com interrupção externa**

Analisando o código C apresentado na Figura 5.8, o programa começa por configurar o periférico DVIC para gerar uma interrupção no processador quando for recebido um sinal do periférico de *input* do microprocessador. Depois de configurado o DVIC é enviado para o periférico de *output* o valor 0xAA contido na variável *output* e o processador entra num ciclo infinito até ocorrer alguma interrupção externa. Quando a interrupção externa ocorre o processador salta para o código presente na função *INT0\_ISR\_vect* e modifica o valor da variável *output* para 0x0F e envia este valor para o periférico de *output*. Finalizado o envio o microprocessador volta para o ciclo infinito até que ocorra nova interrupção.



**Figura 5.9 - Periférico de *output* antes e após a ocorrência da interrupção externa**



Na Figura 5.9 é apresentado o resultado do código apresentado na Figura 5.8. À esquerda é apresentado o estado do periférico de *output* antes da ocorrência da interrupção externa e à direita o estado do periférico depois da ocorrência da interrupção externa.

## 5.4. Testes ao *microkernel*

O *microkernel* para além de fornecer suporte ao desenvolvimento de aplicações *multithread*, é também uma forma de realizar um teste global ao compilador, uma vez que a implementação do *microkernel* inclui todos os componentes da linguagem C. Assim o seu correto funcionamento implica o correto funcionamento de todas as partes integrantes do compilador.

```

//main
int main()
{
    //variáveis locais
    char str[]{"Start scheduler..."};
    char *addr;
    addr=&str[0];

    //configuração da USART
    usartConfig();

    //criação das tarefas
    vTaskCreate(4,200);
    vTaskCreate(3,100);
    vTaskCreate(2,100);
    vTaskCreate(1,50);

    //print de uma msg para sinalizar o inicio do escalonador
    println(addr);

    //inicio do escalonador
    vStartSched();

    return 0;
}

void task1(void)
{
    char str[]{"TASK1"};
    char *addr;
    addr=&str[0];

    while(1)
    {
        println(addr);
        vTaskSuspend();
    }
}

void task2(void)
{
    char str[]{"TASK2"};
    char *addr;
    addr=&str[0];

    while(1)
    {
        println(addr);
        vTaskDelay(22);
    }
}

void task3(void)
{
    char str[]{"TASK3"};
    char *addr;
    addr=&str[0];

    while(1)
    {
        println(addr);
        vTaskDelay(33);
    }
}

void task4(void)
{
    while(1){}
}

```

Figura 5.10 - Código exemplo de uma aplicação de teste do *microkernel*

Na Figura 5.10 é apresentada uma pequena aplicação *multithread* onde cada tarefa imprime o seu nome exceto a *task4* que nesta aplicação executa a função de *idle task*. Analisando o código apresentado, a *task1* imprime uma vez e suspende-se sem nunca retomar a execução. Enquanto as *task2* e *task3* imprimem os respetivos nomes e “adormecem” por um determinado número de *systicks*, voltando a imprimir o mesmo texto após retomarem a execução.

```

Start scheduler...
TASK1
TASK2
TASK3
TASK2
TASK3
TASK2
TASK2
TASK3
TASK2
TASK3
TASK2
TASK2
TASK2
TASK3
TASK2
TASK2
TASK3

```

Figura 5.11 - Resultado da aplicação apresentada na Figura 5.10

Na Figura 5.11 é apresentado o resultado da aplicação descrita na Figura 5.10, como era esperado a *task1* executou uma única vez. As *task2* e *task3* foram obtendo o processador de forma intercalada e imprimiram as respectivas mensagens.

## 5.5. Teste à componente agnóstica do compilador

Para testar a componente agnóstica do compilador foi criada uma outra versão do *microkernel* apresentado na secção 4.2. Esta nova versão difere nos nomes das APIs e no algoritmo de escalonamento, assim uma mesma aplicação ao ser compilada para a nova versão, espera-se um comportamento diferente da mesma aplicação compilada para a versão anteriormente apresentada.

<pre> //API to create a new task void vTaskCreate(int prioridade, int stackSize);  //API to delay a task for a certain number of ticks void vTaskDelay(int ticks);  //API to suspend a task void vTaskSuspend(void); </pre>	<pre> //API to create a new task void createTask(int stackSize);  //API to delay a task for a certain number of ticks void delayTask(int id,int ticks);  //API to suspend a task void suspendTask(int id); </pre>
VO	V1

Figura 5.12 - Comparação das API's das duas versões do *microkernel*

Na Figura 5.12 é apresentado uma figura onde é possível visualizar as diferenças entre as APIs nas duas versões do *microkernel*. É importante salientar que os nomes e os parâmetros que cada API recebe foram alterados. Assim será necessário um tratamento

cuidado por parte do compilador ao identificar as APIs alvo e selecionar os parâmetros corretos a colocar nas APIs do *microkernel* pretendido.

Por forma a comprovar que a aplicação foi efetivamente migrada entre *microkernels* foi implementado na nova versão um algoritmo de escalonamento diferente do da primeira versão. Assim quando a aplicação foi migrada será espectável uma mudança no comportamento da aplicação. Na Figura 5.13 é apresentado o código do escalonador das duas versões do *microkernel*. Na versão 0 o escalonador escolhe a próxima tarefa baseando-se na prioridade da mesma e na disponibilidade desta para executar. Na versão 1 o escalonador atribui o mesmo tempo de execução a todas as tarefas, implementando assim um algoritmo de *round robin* simples.

<pre>//sched int getNextTask(void) {     int i, maior=255;      for(i=0;i&lt;maxTasks;i++)     {         if(List[i].estado == ready)         {             if(maior == 255){maior=i;}              if(List[i].prio &gt; List[maior].prio)             {                 maior = i;             }         }     }      return maior; }</pre>	<pre>//sched int task=0; int getNextTask(void) {     task++;     if(task &gt;= maxTasks){task=0;}     return task; }</pre>
V0	V1

**Figura 5.13 - Comparação dos algoritmos de escalonamento das duas versões do *microkernel***

Para demonstrar o correto funcionamento da componente agnóstica do compilador é apresentado um conjunto de figuras, ilustrando as diversas etapas envolvidas no processo de migração de uma aplicação para dois *microkernels* distintos. Para começar foi desenvolvida uma pequena aplicação utilizando as APIs genéricas fornecidas pela ferramenta. Na Figura 5.14 é apresentado o código da aplicação.

```

int main()
{
    usartConfig();

    genericTaskCreate(4,200);
    genericTaskCreate(3,100);
    genericTaskCreate(2,100);
    genericTaskCreate(1,50);

    char str[]={"Start scheduler..."};
    char *addr;
    addr=&str[0];
    println(addr);

    vStartSched();

    return 0;
}

void task1(void)
{
    char str[]={"TASK1"};
    char *addr;
    addr=&str[0];

    while(1)
    {
        println(addr);
        delayMS(111);
    }
}

void task2(void)
{
    char str[]={"TASK2"};
    char *addr;
    addr=&str[0];

    while(1)
    {
        println(addr);
        delayMS(222);
    }
}

void task3(void)
{
    char str[]={"TASK3"};
    char *addr;
    addr=&str[0];

    while(1)
    {
        println(addr);
        delayMS(333);
    }
}

void task4(void)
{
    while(1){}
}

void delayMS(int mili)
{
    int i,j;

    for(i=0;i<mili;i++)
    {
        for(j=0;j<20000;j++)
        {
            asm(
                OR r0,r0,r0
            );
        }
    }
}

```

Figura 5.14 - Aplicação utilizando as API's genéricas

Analisando o código da Figura 5.14 a aplicação cria quatro tarefas, definindo uma prioridade diferente para cada tarefa para o caso do *microkernel* que tem a prioridade em conta no momento do escalonamento das tarefas. As tarefas imprimem uma mensagem com um texto identificativo de cada tarefa e fazem um *delay* por *pooling*. Na versão 0 do *microkernel* como o escalonador seleciona a próxima tarefa baseando-se na sua prioridade a tarefa 1 como está sempre pronta a executar e tem a maior prioridade será sempre a escolhida. Por outro lado na versão 1 do *microkernel* o escalonador cede o mesmo tempo de execução a todas as tarefas. Assim será de esperar que todas as tarefas tenham oportunidade de imprimir o seu texto.

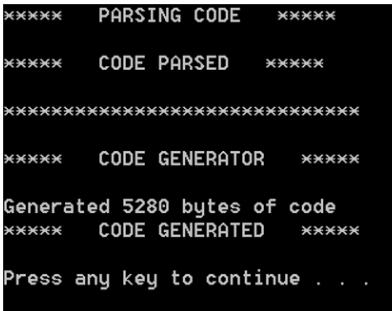
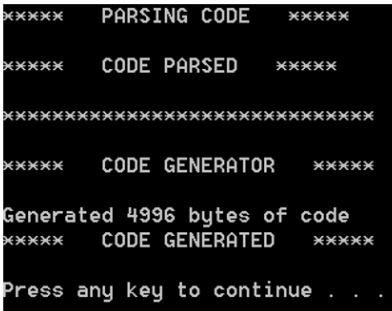
Command Arguments	DRTOS -o out.coe main.c	Command Arguments	TinyRTOS -o out.coe main.c
			

Figura 5.15 - Compilação para *microkernel* alvo

Para escolher o *microkernel* alvo basta inserir nos argumentos do compilador o nome do *microkernel* pretendido como exemplifica a Figura 5.15. Durante o processo de compilação é gerado um conjunto de ficheiros temporários onde as APIs genéricas são substituídas pelas APIs do *microkernel* pretendido. Na Figura 5.16 é apresentado um excerto dos ficheiros gerados para cada versão do *microkernel*.

<pre> } int main ( ) { usartConfig ( ) ; vTaskCreate(4,200);  vTaskCreate(3,100);  vTaskCreate(2,100);  vTaskCreate(1,50);  char str [ ] = { "Start scheduler..." } ; char * addr ; addr = &amp; str [ 0 ] ; println ( addr ) ; vStartSched ( ) ; return 0 ; } </pre>	<pre> } int main ( ) { usartConfig ( ) ; createTask(200);  createTask(100);  createTask(100);  createTask(50);  char str [ ] = { "Start scheduler..." } ; char * addr ; addr = &amp; str [ 0 ] ; println ( addr ) ; vStartSched ( ) ; return 0 ; } </pre>
---	---

Figura 5.16 - Ficheiros intermédios gerados

Depois dos ficheiros temporários serem gerados estes são dados como *input* ao compilador e o código é gerado a partir destes ficheiros.



Figura 5.17 - Resultado dos dois sistemas operativos a executar a mesma aplicação

Na Figura 5.17 é apresentado o conteúdo impresso pelas tarefas através da porta série. É possível verificar que na versão 0 do *microkernel* só a tarefa 1 imprime o texto devido à sua maior prioridade em relação às restantes. Enquanto na versão 1 do *microkernel* todas as tarefas têm a mesma porção de tempo de execução permitindo que todas imprimam o seu texto pela porta série.

## Capítulo 6

### CONCLUSÕES E TRABALHO FUTURO

---

Após a exposição de todos os passos e decisões tomadas ao longo da presente dissertação, neste capítulo são apresentadas as conclusões retiradas do trabalho desenvolvido. É também proposto um conjunto de sugestões por forma a melhorar as diversas ferramentas desenvolvidas e fornecer mais suporte ao programador.

#### 6.1. Conclusões

Com o aumento da competitividade do mercado, é cada vez maior a necessidade de rápido desenvolvimento de *software*. A presente dissertação visa contribuir para a aceleração no desenvolvimento de *software* para uma nova arquitetura, criada no âmbito de outra dissertação, cujo desenvolvimento ocorreu em paralelo. Foi também abordada uma forma de tornar o compilador independente do sistema operativo para o qual se desenvolve aplicações. Esta ferramenta de *porting* automático foi desenvolvida com o objetivo de ser agnóstica, ou seja, ser possível adicionar sistemas operativos alvo sem a necessidade de modificar o código fonte da ferramenta.

Os objetivos inicialmente propostos foram atingidos, conseguindo-se assim no final do trabalho um compilador para a linguagem de programação C funcional e que suporta total acesso aos periféricos da arquitetura alvo. Foi também conseguido um *microkernel* para auxiliar no desenvolvimento de aplicações *multithread*. Por fim, no leque de apoio ao desenvolvimento, foi também conseguida uma ferramenta capaz de abstrair o programador do sistema operativo para o qual a aplicação está a ser desenvolvida, diminuindo assim o esforço de engenharia necessário à portabilidade de aplicações desenvolvidas. No campo da depuração de *software* foi conseguido um simulador fiável da arquitetura alvo que provou ao longo da dissertação ser uma ferramenta essencial na deteção de erros de *software*, tanto no desenvolvimento de funcionalidades do compilador como no desenvolvimento de aplicações. Todos os testes realizados às ferramentas desenvolvidas obtiveram resultado positivo provando assim o seu correto funcionamento e que todos os objetivos propostos foram cumpridos.

As limitações do compilador desenvolvido, nomeadamente a falta de suporte a apontadores para funções e a sua limitada capacidade de pré-processamento, são devido à necessidade de desenvolver um *front end* de raíz, de modo a obter completo controlo sobre o processo de compilação, essencial para integrar a componente de agnosticismo no processador.

## 6.2. Trabalho Futuro

Apesar de o compilador traduzir de forma correta programas escritos em linguagem C, o compilador ainda não suporta a funcionalidade de apontadores para funções. Assim propõe-se acrescentar esta característica, pois traria uma maior facilidade à implementação do *microkernel* presente na dissertação e permitiria compilar sistemas operativos já existentes, pois a maioria deles utiliza apontadores para funções como forma de chamar as funções que definem o comportamento das tarefas.

Outra proposta de melhoramento recai sobre o pré-processador, pois este apenas suporta a funcionalidade de *#include*, assim seria crucial fornecer suporte para o *#define* para que o compilador seja capaz de compilar um sistema operativo comercial como o FreeRTOS. Isto será realizado, removendo o pré-processamento embutido no *front-end*, e utilizando em vez o pré-processador C.

Depois de introduzidas as duas funcionalidades supracitadas seria interessante fazer o *porting* do FreeRTOS para a arquitetura alvo e introduzir o FreeRTOS no repositório e utilizá-lo como um sistema operativo alvo das aplicações.

Quanto à implementação da ferramenta de agnosticismo seria uma mais valia se os ficheiros que contêm as informações acerca das APIs dos sistemas operativos fossem guardadas em notação XML de forma a padronizar a sua interpretação.



## REFERÊNCIAS BIBLIOGRÁFICAS

---

- [1] C. Batten, “Pipelining Structural & Data Hazards Iron Law of Processor Performance,” 2012.
- [2] “1.3 Compiler Architecture.” [Online]. Available: <https://lambda.uta.edu/cse5317/notes/node5.html>. [Accessed: 05-Oct-2013].
- [3] T. Parr, “Antlr,” 2006. [Online]. Available: <http://www.antlr.org/>. [Accessed: 01-Oct-2014].
- [4] “The LEX & YACC Page.” [Online]. Available: <http://dinosaur.compilertools.net/>. [Accessed: 02-Oct-2013].
- [5] “gnu.org.” [Online]. Available: <http://www.gnu.org/software/bison/>. [Accessed: 06-Oct-2014].
- [6] R. Mak, *Writing compilers and interpreters*, 3ª edição. 2009.
- [7] D. M. Ritchie, “Chistory.” [Online]. Available: <ftp://cm.bell-labs.com/who/dmr/chist.html>. [Accessed: 07-Oct-2013].
- [8] B. Kernighan and D. Ritchie, *The C Programming Language*, 2nd Editio. .
- [9] B. Hailpern and D. M. Ritchie, “C Session,” pp. 671–698.
- [10] D. Ritchie, B. Labs, and K. Thompson, “História,” pp. 55–95, 1973.
- [11] G. Team, “GNU GCC.” [Online]. Available: <https://gcc.gnu.org/>. [Accessed: 29-Sep-2014].
- [12] F. Bellard, “Tiny C Compiler.” [Online]. Available: <http://bellard.org/tcc/>. [Accessed: 20-Feb-2014].
- [13] A. Silberschatz, P. Galvin, and G. Gagne, *Operating system concepts*. 1998.
- [14] M. Rouse, “Portability,” 2005. [Online]. Available: <http://searchstorage.techtarget.com/definition/portability>. [Accessed: 29-Sep-2014].
- [15] M. Rouse, “device-agnostic (device agnosticism),” 2007. [Online]. Available: <http://searchconsumerization.techtarget.com/definition/device-agnostic-device-agnosticism>. [Accessed: 29-Sep-2014].
- [16] W. Kahan, “IEEE standard 754 for binary floating-point arithmetic,” *Lect. Notes Status IEEE*, pp. 1–23, 1996.



## ANEXOS

### Anexo A Instruction Set

**Tabela 5 - Instruções lógicas e aritméticas**

Mnemónica	Operação	Descrição
ADD	$Rd=Ra+Rb$	Soma de dois registos
ADDI	$Rd=Ra+IMM$	Soma de um registo com um imediato
SUB	$Rd=Ra-Rb$	Subtração de dois registos
SUBI	$Rd=Ra-IMM$	Subtração de um registo com um imediato
OR	$Rd=Ra   Rb$	OU lógico de dois registos
ORI	$Rd=Ra   IMM$	OU lógico de um registo com um imediato
AND	$Rd=Ra \& Rb$	E lógico de dois registos
ANDI	$Rd=Ra \& IMM$	E lógico de um registo com um imediato
XOR	$Rd=Ra \wedge Rb$	OU exclusivo de dois registos
XORI	$Rd=Ra \wedge IMM$	OU exclusivo de um registo com um imediato
NOT	$Rd=!Ra$	Nega um registo
BRL	$Rd=Ra \ll Rb$	<i>Shift</i> lógico à esquerda
BRR	$Rd=Ra \gg Rb$	<i>Shift</i> lógico à direita

**Tabela 6 - Instruções de salto e salto condicional**

Mnemónica	Operação	Descrição
BR	$PC=PC+Rb$	Salta para a localização de PC mais o offset dado por Rb
BRI	$PC=PC+IMM$	Salta para a localização de PC mais o offset dado pelo imediato
BRA	$PC=Rb$	Salto absoluto para a localização dada pelo valor de Rb
BRAI	$PC=IMM$	Salto absoluto para a localização dada pelo imediato
BEQ	$PC=PC+Rb$ se $Ra==0$	Salta para a localização de PC mais o <i>offset</i> dado por Rb se o valor de Ra for zero

BNE	$PC=PC+Rb$ se $Ra \neq 0$	Salta para a localização de PC mais o <i>offset</i> dado por Rb se o valor de Ra for diferente de zero
BGE	$PC=PC+Rb$ se $Ra \geq 0$	Salta para a localização de PC mais o <i>offset</i> dado por Rb se o valor de Ra for maior ou igual a zero
BGT	$PC=PC+Rb$ se $Ra > 0$	Salta para a localização de PC mais o <i>offset</i> dado por Rb se o valor de Ra for maior que zero
BLE	$PC=PC+Rb$ se $Ra \leq 0$	Salta para a localização de PC mais o <i>offset</i> dado por Rb se o valor de Ra for menor ou igual a zero
BLT	$PC=PC+Rb$ se $Ra < 0$	Salta para a localização de PC mais o <i>offset</i> dado por Rb se o valor de Ra for menor que zero

**Tabela 7 - Instruções de transferência de dados**

Mnemónica	Operação	Descrição
LW	$Rd=@(Ra+Rb)$	Carrega um valor do endereço da memória dado pelo endereço da soma de Ra com Rb
LWI	$Rd=@(Ra+IMM)$	Carrega um valor do endereço da memória dado pelo endereço da soma de Ra com um imediato
SW	$ @(Ra+Rb)=Rd$	Guarda o valor de Rd no endereço de memória dado pela soma de Ra com Rb
SWI	$ @(Ra+IMM)=Rd$	Guarda o valor de Rd no endereço de memória dado pela soma de Ra com um imediato
MOV	$Rd=Ra$	Copia o valor de Ra para Rd
MOVI	$Rd=IMM$	Copia o valor do imediato para Rd
PUSH	$Stack=Ra$	Guarda Ra no topo da <i>stack</i>
POP	$Rd=Stack$	Coloca em Rd o valor do topo da <i>stack</i>

**Tabela 8 - Outras instruções**

Mnemónica	Operação	Descrição
NOP		Não faz nenhuma operação durante um ciclo de relógio
RET		Retorna de uma subrotina
RETI		Retorna de uma interrupção
CMP	Rd = 0, se Ra == Rb Rd = 1, se Ra > Rb Rd = 2, se Ra < Rb	Compara o registo Ra com o registo Rb e coloca o resultado em Rd, esta instrução é sempre seguida de uma instrução de salto condicional

## Anexo B Diagrama de classes do *scanner*

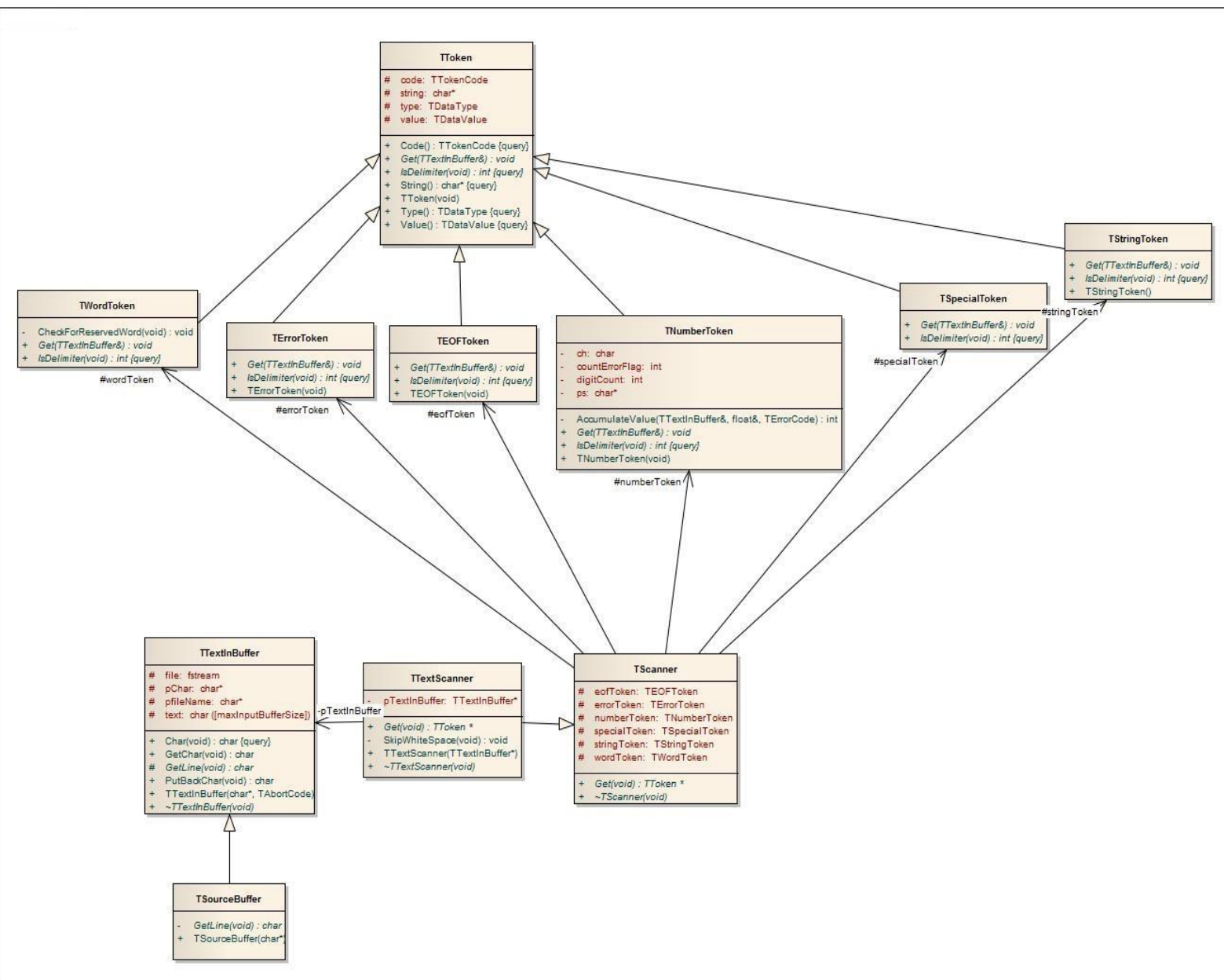


Figura 6.1 – Diagrama de classes do *scanner*

## Anexo C Fluxograma de declarações

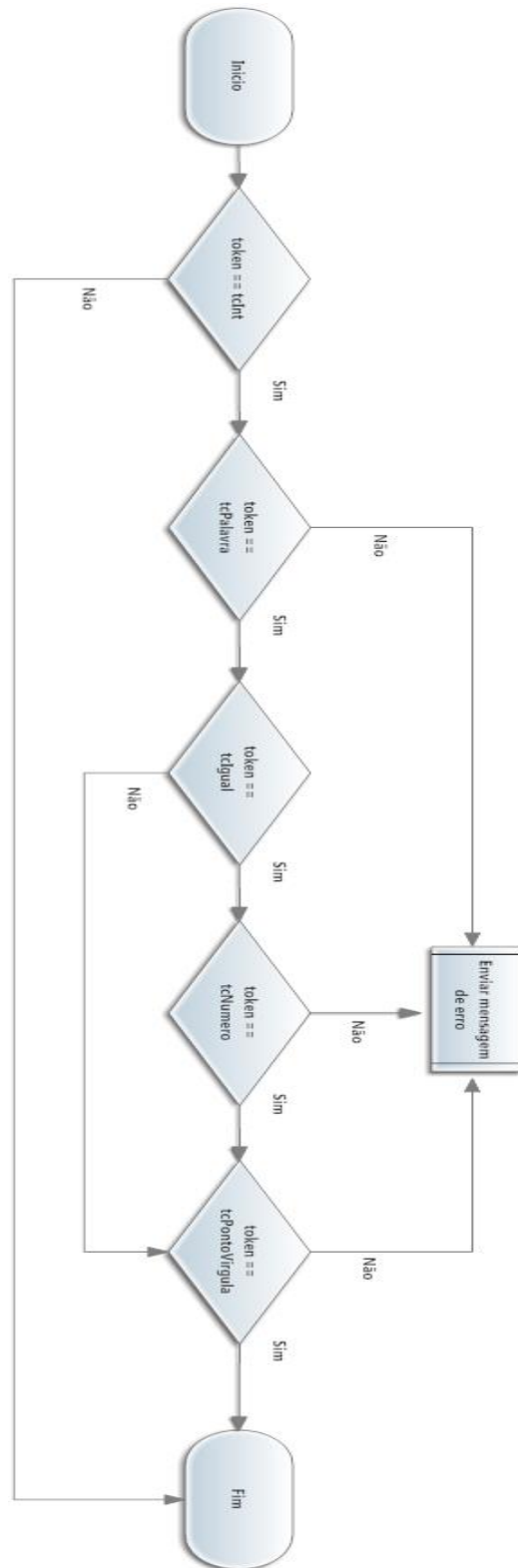


Figura 6.2 – Fluxograma de declaração de variáveis

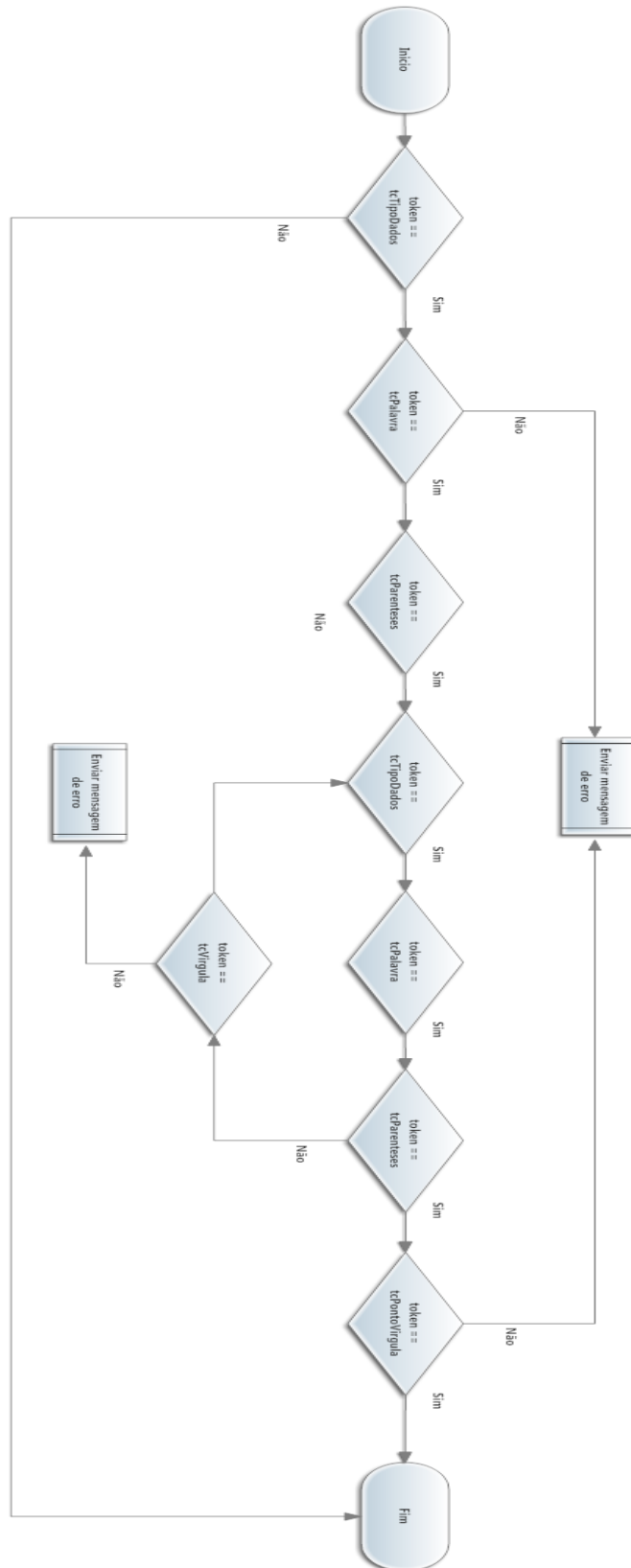


Figura 6.3 – Fluxograma de declaração de funções



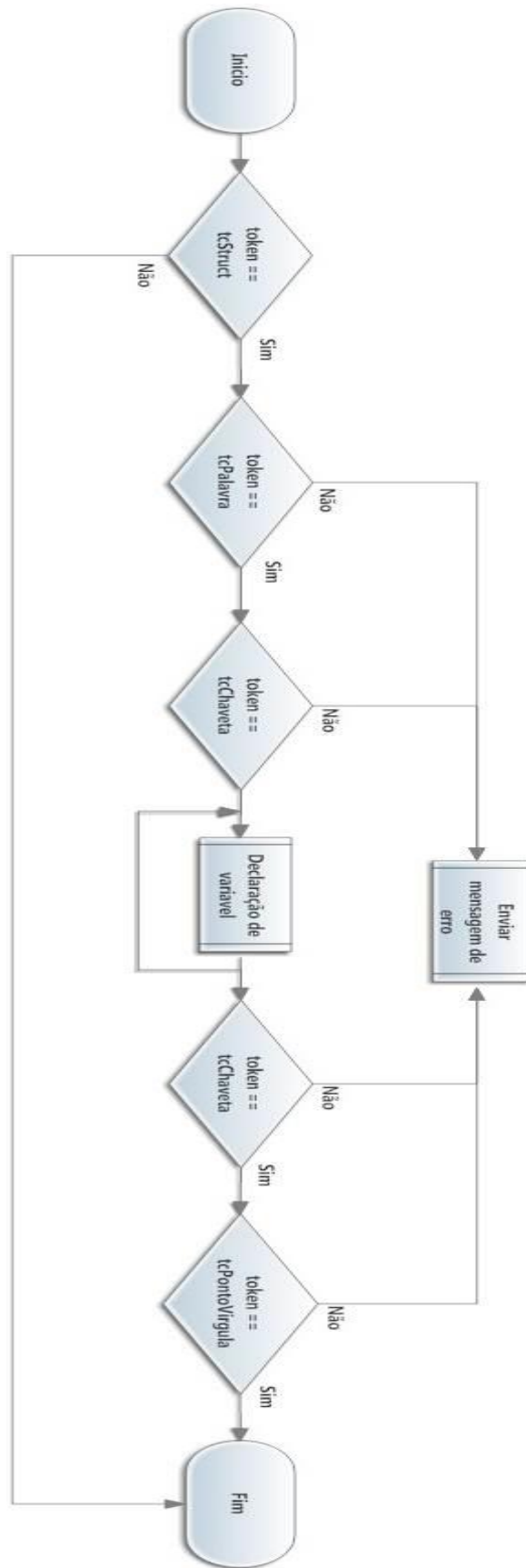


Figura 6.4 – Fluxograma de declaração de estruturas

## Anexo D Diagrama de classes da classe *TParser*



Figura 6.5 – Diagrama de classes da classe *TParser*

## Anexo E Fluxogramas das operações de divisão e resto da divisão

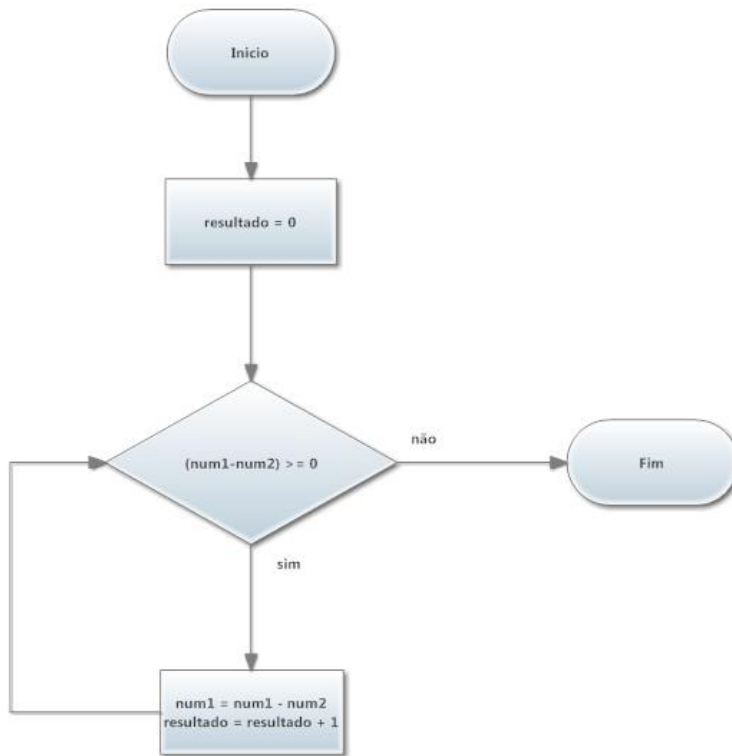


Figura 6.6 – Algoritmo para obter a divisão à custa de subtrações

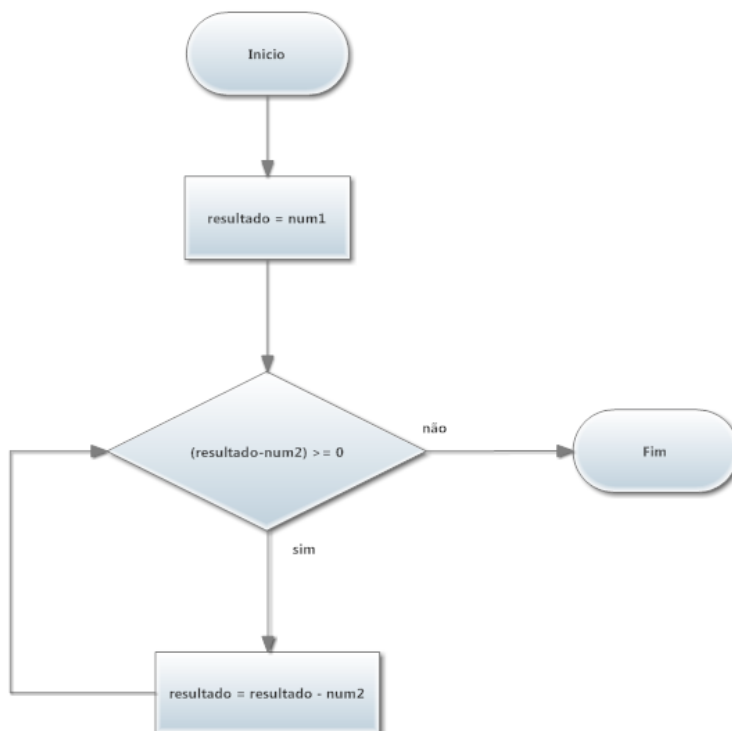


Figura 6.7 – Algoritmo para obter o resto da divisão à custa de subtrações

## Anexo F Bibliotecas de *floats*

0x0043	<b>POP</b>	r11	0x0079	<b>ADD</b>	r1,r4,r5
0x0044	SWI	r11,r31,3	0x007A	MOVI	r8,24
0x0045	<b>POP</b>	r3	0x007B	<b>BSR</b>	r11,r1,r8
0x0046	<b>POP</b>	r2	0x007C	ANDI	r11,r11,1
0x0047	IMM	127	0x007D	MOVI	r10,4
0x0048	ANDI	r4,r2,65535	0x007E	<b>CMP</b>	r9,r11,r0
0x0049	IMM	127	0x007F	BEQ	r9,r10
0x004A	ANDI	r5,r3,65535	0x0080	MOVI	r11,1
0x004B	MOVI	r8,23	0x0081	ADDI	r12,r12,1
0x004C	<b>BSR</b>	r6,r2,r8	0x0082	<b>BSR</b>	r1,r1,r11
0x004D	<b>BSR</b>	r7,r3,r8	0x0083	BRI	0x61
0x004E	ANDI	r6,r6,255	0x0084	MOVI	r10,39
0x004F	ANDI	r7,r7,255	0x0085	<b>CMP</b>	r9,r6,r0
0x0050	MOVI	r10,6	0x0086	BNE	r9,r10
0x0051	<b>CMP</b>	r9,r4,r0	0x0087	MOVI	r10,36
0x0052	BNE	r9,r10	0x0088	<b>CMP</b>	r9,r7,r0
0x0053	MOVI	r10,3	0x0089	BEQ	r9,r10
0x0054	<b>CMP</b>	r9,r6,r0	0x008A	MOVI	r10,3
0x0055	BNE	r9,r10	0x008B	<b>CMP</b>	r9,r5,r4
0x0056	<b>MOV</b>	r1,r3	0x008C	BLE	r9,r10
0x0057	BRI	0x93	0x008D	<b>SUB</b>	r1,r5,r4
0x0058	MOVI	r10,6	0x008E	BRI	0x2
0x0059	<b>CMP</b>	r9,r5,r0	0x008F	<b>SUB</b>	r1,r4,r5
0x005A	BNE	r9,r10	0x0090	MOVI	r2,0
0x005B	MOVI	r10,3	0x0091	MOVI	r3,1
0x005C	<b>CMP</b>	r9,r7,r0	0x0092	MOVI	r8,23
0x005D	BNE	r9,r10	0x0093	<b>BSR</b>	r11,r1,r8
0x005E	<b>MOV</b>	r1,r2	0x0094	ANDI	r11,r11,1
0x005F	BRI	0x8B	0x0095	<b>CMP</b>	r9,r11,r0
0x0060	MOVI	r11,1	0x0096	MOVI	r10,8
0x0061	BSL	r11,r11,r8	0x0097	BNE	r9,r10
0x0062	<b>OR</b>	r4,r4,r11	0x0098	<b>CMP</b>	r9,r2,r8
0x0063	<b>OR</b>	r5,r5,r11	0x0099	MOVI	r10,5
0x0064	SUBI	r6,r6,127	0x009A	BGE	r9,r10
0x0065	SUBI	r7,r7,127	0x009B	BSL	r1,r1,r3
0x0066	MOVI	r10,5	0x009C	SUBI	r12,r12,1
0x0067	<b>CMP</b>	r9,r6,r7	0x009D	ADDI	r2,r2,1
0x0068	BLE	r9,r10	0x009E	BRI	-0xB
0x0069	<b>SUB</b>	r12,r6,r7	0x009F	<b>CMP</b>	r9,r2,r8
0x006A	<b>BSR</b>	r5,r5,r12	0x00A0	MOVI	r10,4
0x006B	<b>MOV</b>	r12,r6	0x00A1	BLT	r9,r10
0x006C	BRI	0x4	0x00A2	MOVI	r1,0
0x006D	<b>SUB</b>	r12,r7,r6	0x00A3	IMM	65535
0x006E	<b>BSR</b>	r4,r4,r12	0x00A4	MOVI	r12,65409
0x006F	<b>MOV</b>	r12,r7	0x00A5	<b>CMP</b>	r9,r4,r5
0x0070	MOVI	r8,31	0x00A6	MOVI	r10,5
0x0071	<b>BSR</b>	r6,r2,r8	0x00A7	BGE	r9,r10
0x0072	<b>BSR</b>	r7,r3,r8	0x00A8	MOVI	r11,1
0x0073	MOVI	r10,15	0x00A9	MOVI	r8,31
0x0074	<b>CMP</b>	r9,r6,r0	0x00AA	BSL	r11,r11,r8
0x0075	BNE	r9,r10	0x00AB	<b>OR</b>	r1,r1,r11
0x0076	MOVI	r10,12	0x00AC	BRI	0x38
0x0077	<b>CMP</b>	r9,r7,r0	0x00AD	MOVI	r10,39
0x0078	BNE	r9,r10	0x00AE	<b>CMP</b>	r9,r6,r0

0x00AF	BEQ	r9,r10	0x00E1	MOVI	r8,31
0x00B0	MOVI	r10,36	0x00E2	BSL	r11,r11,r8
0x00B1	<b>CMP</b>	r9,r7,r0	0x00E3	<b>OR</b>	r1,r1,r11
0x00B2	BNE	r9,r10	0x00E4	IMM	32895
0x00B3	MOVI	r10,3	0x00E5	ANDI	r1,r1,65535
0x00B4	<b>CMP</b>	r9,r5,r4	0x00E6	MOVI	r8,23
0x00B5	BLE	r9,r10	0x00E7	ADDI	r12,r12,127
0x00B6	<b>SUB</b>	r1,r5,r4	0x00E8	BSL	r12,r12,r8
0x00B7	BRI	0x2	0x00E9	<b>OR</b>	r1,r1,r12
0x00B8	<b>SUB</b>	r1,r4,r5	0x00EA	<b>MOV</b>	r15,r1
0x00B9	MOVI	r2,0	0x00EB	<b>RET</b>	
0x00BA	MOVI	r3,1			
0x00BB	MOVI	r8,23			
0x00BC	<b>BSR</b>	r11,r1,r8			
0x00BD	ANDI	r11,r11,1			
0x00BE	<b>CMP</b>	r9,r11,r0			
0x00BF	MOVI	r10,8			
0x00C0	BNE	r9,r10			
0x00C1	<b>CMP</b>	r9,r2,r8			
0x00C2	MOVI	r10,5			
0x00C3	BGE	r9,r10			
0x00C4	BSL	r1,r1,r3			
0x00C5	SUBI	r12,r12,1			
0x00C6	ADDI	r2,r2,1			
0x00C7	BRI	-0xB			
0x00C8	<b>CMP</b>	r9,r2,r8			
0x00C9	MOVI	r10,4			
0x00CA	BLT	r9,r10			
0x00CB	MOVI	r1,0			
0x00CC	IMM	65535			
0x00CD	MOVI	r12,65409			
0x00CE	<b>CMP</b>	r9,r5,r4			
0x00CF	MOVI	r10,5			
0x00D0	BGE	r9,r10			
0x00D1	MOVI	r11,1			
0x00D2	MOVI	r8,31			
0x00D3	BSL	r11,r11,r8			
0x00D4	<b>OR</b>	r1,r1,r11			
0x00D5	BRI	0xF			
0x00D6	<b>ADD</b>	r1,r4,r5			
0x00D7	MOVI	r8,24			
0x00D8	<b>BSR</b>	r11,r1,r8			
0x00D9	ANDI	r11,r11,1			
0x00DA	MOVI	r10,4			
0x00DB	<b>CMP</b>	r9,r11,r0			
0x00DC	BEQ	r9,r10			
0x00DD	MOVI	r11,1			
0x00DE	ADDI	r12,r12,1			
0x00DF	<b>BSR</b>	r1,r1,r11			
0x00E0	MOVI	r11,1			

Figura 6.8 - Biblioteca *assembly* para somas e subtrações de *floats*

0x0043	<b>POP</b>	r12	0x007D	ANDI	r12,r12,1
0x0044	SWI	r12,r31,3	0x007E	<b>CMP</b>	r10,r12,r0
0x0045	<b>POP</b>	r3	0x007F	BNE	r10,r11
0x0046	<b>POP</b>	r2	0x0080	SUBI	r6,r6,1
0x0047	IMM	127	0x0081	BRI	-0x5
0x0048	ANDI	r4,r2,65535	0x0082	MOVI	r7,31
0x0049	IMM	127	0x0083	MOVI	r13,31
0x004A	ANDI	r5,r3,65535	0x0084	MOVI	r11,3
0x004B	MOVI	r9,23	0x0085	<b>BSR</b>	r12,r4,r7
0x004C	<b>BSR</b>	r6,r2,r9	0x0086	ANDI	r12,r12,1
0x004D	<b>BSR</b>	r7,r3,r9	0x0087	<b>CMP</b>	r10,r12,r0
0x004E	ANDI	r6,r6,255	0x0088	BNE	r10,r11
0x004F	ANDI	r7,r7,255	0x0089	SUBI	r7,r7,1
0x0050	MOVI	r11,8	0x008A	BRI	-0x5
0x0051	<b>CMP</b>	r10,r4,r0	0x008B	MOVI	r11,3
0x0052	BNE	r10,r11	0x008C	<b>BSR</b>	r12,r5,r13
0x0053	MOVI	r11,5	0x008D	ANDI	r12,r12,1
0x0054	<b>CMP</b>	r10,r6,r0	0x008E	<b>CMP</b>	r10,r12,r0
0x0055	BNE	r10,r11	0x008F	BNE	r10,r11
0x0056	MOVI	r1,0	0x0090	SUBI	r13,r13,1
0x0057	IMM	65535	0x0091	BRI	-0x5
0x0058	MOVI	r8,65409	0x0092	<b>ADD</b>	r12,r7,r13
0x0059	BRI	0x4F	0x0093	ADDI	r12,r12,1
0x005A	MOVI	r11,8	0x0094	<b>BSR</b>	r12,r1,r12
0x005B	<b>CMP</b>	r10,r5,r0	0x0095	ANDI	r12,r12,1
0x005C	BNE	r10,r11	0x0096	MOVI	r11,2
0x005D	MOVI	r11,5	0x0097	<b>CMP</b>	r10,r12,r0
0x005E	<b>CMP</b>	r10,r7,r0	0x0098	BEQ	r10,r11
0x005F	BNE	r10,r11	0x0099	ADDI	r8,r8,1
0x0060	MOVI	r1,0	0x009A	SUBI	r9,r6,23
0x0061	IMM	65535	0x009B	<b>BSR</b>	r1,r1,r9
0x0062	MOVI	r8,65409	0x009C	ADDI	r8,r8,127
0x0063	BRI	0x45	0x009D	IMM	127
0x0064	MOVI	r12,1	0x009E	ANDI	r1,r1,65535
0x0065	BSL	r12,r12,r9	0x009F	MOVI	r9,23
0x0066	<b>OR</b>	r4,r4,r12	0x00A0	BSL	r8,r8,r9
0x0067	<b>OR</b>	r5,r5,r12	0x00A1	<b>OR</b>	r1,r1,r8
0x0068	SUBI	r6,r6,127	0x00A2	IMM	32768
0x0069	SUBI	r7,r7,127	0x00A3	ANDI	r4,r2,0
0x006A	MOVI	r11,5	0x00A4	IMM	32768
0x006B	<b>CMP</b>	r10,r6,r7	0x00A5	ANDI	r5,r3,0
0x006C	BLE	r10,r11	0x00A6	<b>XOR</b>	r4,r4,r5
0x006D	<b>SUB</b>	r8,r6,r7	0x00A7	<b>OR</b>	r1,r1,r4
0x006E	<b>BSR</b>	r5,r5,r8	0x00A8	<b>MOV</b>	r15,r1
0x006F	<b>MOV</b>	r8,r6	0x00A9	<b>RET</b>	
0x0070	BRI	0x4			
0x0071	<b>SUB</b>	r8,r7,r6			
0x0072	<b>BSR</b>	r4,r4,r8			
0x0073	<b>MOV</b>	r8,r7			
0x0074	MOVI	r9,8			
0x0075	<b>BSR</b>	r4,r4,r9			
0x0076	<b>BSR</b>	r5,r5,r9			
0x0077	<b>MUL</b>	r4,r4,r5			
0x0078	<b>MOV</b>	r1,r4			
0x0079	<b>ADD</b>	r8,r6,r7			
0x007A	MOVI	r6,31			
0x007B	MOVI	r11,3			
0x007C	<b>BSR</b>	r12,r1,r6			

Figura 6.9 - Biblioteca *assembly* para multiplicação de *floats*

0x0043	<b>POP</b>	r12	0x007B	<b>ADDI</b>	r13,r13,1
0x0044	<b>SWI</b>	r12,r31,3	0x007C	<b>BRI</b>	-0x5
0x0045	<b>POP</b>	r3	0x007D	<b>MOV</b>	r1,r13
0x0046	<b>POP</b>	r2	0x007E	<b>MOVI</b>	r9,23
0x0047	<b>IMM</b>	127	0x007F	<b>MOVI</b>	r17,23
0x0048	<b>ANDI</b>	r4,r2,65535	0x0080	<b>MOVI</b>	r18,1
0x0049	<b>IMM</b>	127	0x0081	<b>MOVI</b>	r11,7
0x004A	<b>ANDI</b>	r5,r3,65535	0x0082	<b>BSR</b>	r12,r1,r9
0x004B	<b>MOVI</b>	r9,23	0x0083	<b>ANDI</b>	r12,r12,1
0x004C	<b>BSR</b>	r6,r2,r9	0x0084	<b>CMP</b>	r10,r12,r0
0x004D	<b>BSR</b>	r7,r3,r9	0x0085	<b>BNE</b>	r10,r11
0x004E	<b>ANDI</b>	r6,r6,255	0x0086	<b>MOVI</b>	r11,4
0x004F	<b>ANDI</b>	r7,r7,255	0x0087	<b>CMP</b>	r10,r17,r0
0x0050	<b>MOVI</b>	r11,8	0x0088	<b>BLT</b>	r10,r11
0x0051	<b>CMP</b>	r10,r4,r0	0x0089	<b>BSL</b>	r1,r1,r18
0x0052	<b>BNE</b>	r10,r11	0x008A	<b>SUBI</b>	r17,r17,1
0x0053	<b>MOVI</b>	r11,5	0x008B	<b>BRI</b>	-0xA
0x0054	<b>CMP</b>	r10,r6,r0	0x008C	<b>MOV</b>	r8,r17
0x0055	<b>BNE</b>	r10,r11	0x008D	<b>MOVI</b>	r9,8
0x0056	<b>MOVI</b>	r1,0	0x008E	<b>BSR</b>	r5,r5,r9
0x0057	<b>IMM</b>	65535	0x008F	<b>MOVI</b>	r9,7
0x0058	<b>MOVI</b>	r8,65409	0x0090	<b>BSL</b>	r4,r4,r9
0x0059	<b>BRI</b>	0x55	0x0091	<b>MOVI</b>	r19,0
0x005A	<b>MOVI</b>	r11,8	0x0092	<b>MOV</b>	r21,r4
0x005B	<b>CMP</b>	r10,r5,r0	0x0093	<b>MOVI</b>	r22,4
0x005C	<b>BNE</b>	r10,r11	0x0094	<b>SUB</b>	r24,r21,r5
0x005D	<b>MOVI</b>	r11,5	0x0095	<b>CMP</b>	r23,r24,r0
0x005E	<b>CMP</b>	r10,r7,r0	0x0096	<b>BLT</b>	r23,r22
0x005F	<b>BNE</b>	r10,r11	0x0097	<b>SUB</b>	r21,r21,r5
0x0060	<b>MOVI</b>	r1,0	0x0098	<b>ADDI</b>	r19,r19,1
0x0061	<b>IMM</b>	65535	0x0099	<b>BRI</b>	-0x5
0x0062	<b>MOVI</b>	r8,65409	0x009A	<b>MOV</b>	r1,r19
0x0063	<b>BRI</b>	0x4B	0x009B	<b>MOVI</b>	r9,23
0x0064	<b>MOVI</b>	r12,1	0x009C	<b>MOVI</b>	r11,3
0x0065	<b>BSL</b>	r12,r12,r9	0x009D	<b>BSR</b>	r12,r1,r9
0x0066	<b>OR</b>	r4,r4,r12	0x009E	<b>CMP</b>	r10,r12,r0
0x0067	<b>OR</b>	r5,r5,r12	0x009F	<b>BNE</b>	r10,r11
0x0068	<b>SUBI</b>	r6,r6,127	0x00A0	<b>BSL</b>	r1,r1,r18
0x0069	<b>SUBI</b>	r7,r7,127	0x00A1	<b>BRI</b>	-0x4
0x006A	<b>MOVI</b>	r11,5	0x00A2	<b>ADDI</b>	r8,r8,127
0x006B	<b>CMP</b>	r10,r6,r7	0x00A3	<b>IMM</b>	32895
0x006C	<b>BLE</b>	r10,r11	0x00A4	<b>ANDI</b>	r1,r1,65535
0x006D	<b>SUB</b>	r8,r6,r7	0x00A5	<b>MOVI</b>	r9,23
0x006E	<b>BSR</b>	r5,r5,r8	0x00A6	<b>BSL</b>	r8,r8,r9
0x006F	<b>MOV</b>	r8,r6	0x00A7	<b>OR</b>	r1,r1,r8
0x0070	<b>BRI</b>	0x4	0x00A8	<b>IMM</b>	32768
0x0071	<b>SUB</b>	r8,r7,r6	0x00A9	<b>ANDI</b>	r9,r2,0
0x0072	<b>BSR</b>	r4,r4,r8	0x00AA	<b>IMM</b>	32768
0x0073	<b>MOV</b>	r8,r7	0x00AB	<b>ANDI</b>	r12,r3,0
0x0074	<b>MOVI</b>	r13,0	0x00AC	<b>XOR</b>	r9,r9,r12
0x0075	<b>MOV</b>	r17,r4	0x00AD	<b>OR</b>	r1,r1,r9
0x0076	<b>MOVI</b>	r18,4	0x00AE	<b>MOV</b>	r15,r1
0x0077	<b>SUB</b>	r20,r17,r5	0x00AF	<b>RET</b>	
0x0078	<b>CMP</b>	r19,r20,r0			
0x0079	<b>BLT</b>	r19,r18			
0x007A	<b>SUB</b>	r17,r17,r5			

Figura 6.10 - Biblioteca *assembly* para divisão de *floats*

## Anexo G Código *microkernel*

```
void restoreContext(void)
{
    asm(
        POP r30
        POP r29
        POP r28
        POP r27
        POP r26
        POP r25
        POP r24
        POP r23
        POP r22
        POP r21
        POP r20
        POP r19
        POP r18
        POP r17
        POP r16
        POP r15
        POP r14
        POP r13
        POP r12
        POP r11
        POP r10
        POP r9
        POP r8
        POP r7
        POP r6
        POP r5
        POP r4
        POP r3
        POP r2
        POP r1
        RET
    );
}
```

**Figura 6.11** – Código para restaurar o contexto de uma tarefa



```
void saveContext(void)
{
    asm(
        PUSH r1
        PUSH r2
        PUSH r3
        PUSH r4
        PUSH r5
        PUSH r6
        PUSH r7
        PUSH r8
        PUSH r9
        PUSH r10
        PUSH r11
        PUSH r12
        PUSH r13
        PUSH r14
        PUSH r15
        PUSH r16
        PUSH r17
        PUSH r18
        PUSH r19
        PUSH r20
        PUSH r21
        PUSH r22
        PUSH r23
        PUSH r24
        PUSH r25
        PUSH r26
        PUSH r27
        PUSH r28
        PUSH r29
        PUSH r30
    );
}
```

**Figura 6.12 – Código para salvar o contexto de uma tarefa**

```

int *nextTask=1;
//interrupcao do timer pa gerar o systick
void SYSTICK_ISR_vect(void) naked
{
    //SALVAGUARDA DO CONTEXTO DA TAREFA QUE VAMOS PARAR
    saveContext();

    //guarda a nova stack
    asm(
        SWI r31,r0,1
    );
    List[idRunTask].stack=*nextTask;

    //colocamos o estado da tarefa que estava a correr a ready
    novamente
    //e fazemos update aos soft timers
    if(List[idRunTask].estado == run)
    {
        List[idRunTask].estado = ready;
        updateSoftTimers();
    }
    //procuramos a nova tarefa a correr
    idRunTask = getNextTask();
    //mudar a stack para a stack da tarefa selecionada
    *nextTask=List[idRunTask].stack;
    asm(
        LWI r31,r0,1
    );

    //alteramos o estado da tarefa pa run
    List[idRunTask].estado=run;

    //RESTAURO DO CONTEXTO DA NOVA TAREFA A EXECUTAR
    restoreContext();
}

```

**Figura 6.13 – Código da interrupção do *systick***

```

//systick config
void sysTickConfig(void)
{
    //configura o DVIC
    int val=1;
    writeBus(val,dvicID,IRQ_SYSTICK_enable); //tenho de
    mudar estas labels para o systick
    val=31;
    writeBus(val,dvicID,IRQ_SYSTICK_prio);

    //configura o systick
    val=1000; //para o simulador
    usar 10 para hardware 1000
    val *= 1000;
    writeBus(val,systickID,SYSTICK_count);
    writeBus(val,systickID,SYSTICK_reload);
    val=1;
    writeBus(val,systickID,SYSTICK_enable);
}

```

**Figura 6.14 – Código para configurar o periférico *systick* e habilitar a sua interrupção no DVIC**

```

void getPos(void)
{
    asm(
        LWI    r1,r16,2
        MOVI   r2,0
        SW     r1,r2,r0
    );
}

```

**Figura 6.15 – Função para obter o endereço de salto da primeira execução de uma tarefa**

```

//sched
int getNextTask(void)
{
    int i, maior=255;

    for(i=0;i<maxTasks;i++)
    {
        if(List[i].estado == ready)
        {
            if(maior == 255){maior=i;}

            if(List[i].prio > List[maior].prio)
            {
                maior = i;
            }
        }
    }

    return maior;
}

```

**Figura 6.16 – Função que escalona as tarefas**

```

//update das tasks com delays
void updateSoftTimers(void)
{
    int i;

    for(i=0;i<maxTasks;i++)
    {
        if(List[i].sleepTicks > 0)
        {
            List[i].sleepTicks--;
            if(List[i].sleepTicks == 0)
            {
                List[i].estado=ready;
            }
        }
    }
}

```

**Figura 6.17 – Função que faz o *update* da informação das tarefas que estão no estado de *delay***

```

void refreshStack(void)
{
    int *ptr=0;
    int *ptr2;
    int i;

    for(i=0;i<maxTasks;i++)
    {
        if(List[i].stack != 0)
        {
            ptr2=List[i].stack;
            ptr2 += 31;
            *ptr2=*ptr;
            ptr2 -= 14;
            *ptr2=32;
        }
    }
}

```

**Figura 6.18 – Função que cria uma *stack* virtual para a primeira execução de uma tarefa**

```

void vStartSched(void)
{
    int *ptr=1;
    //configurar systick
    sysTickConfig();

    //chama o vTaskCaller para obter o endereço de salto caso seja a
    primeira vez que corre a tarefa
    vTaskCaller();

    //injeta o valor em todas as stacks
    refreshStack();

    //obtem a proxima task a correr
    idRunTask = getNextTask();

    //mudar a stack para a stack da tarefa selecionada
    *ptr=List[idRunTask].stack;
    asm(
        LWI r31,r0,1
    );

    //alteramos o estado da tarefa pa run
    List[idRunTask].estado=run;

    //context restore
    restoreContext();

    //nao deve chegar aqui, mas fica por segurança durante o
    desenvolvimento
    while(1){}
}

```

**Figura 6.19 – Função que inicializa o escalonador e cede o processador à primeira tarefa a executar**

```

int idGen=0;
int stackPosGen=4050;
void vTaskCreate(int prioridade, int stackSize)
{
    List[idGen].id=idGen;
    List[idGen].prio=prioridade;
    List[idGen].estado=ready;
    List[idGen].stack=stackPosGen - 31;    //o menos 31 e pa simular a
stack com um contexto

    //atualiza os geradores para os novos valores
    stackPosGen -= stackSize;
    idGen++;
}

```

**Figura 6.20 – API para criar uma nova tarefa**

```

void vTaskDelay(int ticks)
{
    List[idRunTask].sleepTicks=ticks;
    List[idRunTask].estado=wait;
    //força um ponto de escalonamento
    int val=1;
    writeBus(val,dvicID,IRQ_SYSTICK_pend);
}

```

**Figura 6.21 – API para colocar uma tarefa no estado de *delay***

```

void vTaskSuspend()
{
    List[idRunTask].estado=suspend;
    //força um ponto de escalonamento
    int val=1;
    writeBus(val,dvicID,IRQ_SYSTICK_pend);
}

```

**Figura 6.22 – API para colocar uma tarefa no estado de *suspend***