ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

PDS16inEcplise

André Ramanlal
Tiago Oliveira

Orientadores Tiago Miguel Braga da Silva Dias
Pedro Miguel Fernandes Sampaio

Relatório de progresso realizado no âmbito de Projecto e Seminário
do curso de licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2015/2016

Abril de 2016

Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

PDS16inEcplise

39204 André Akshei Manoje Ramanlal

40653 Tiago José Vital Oliveira

Orientadores: Tiago Miguel Braga da Silva Dias

Pedro Miguel Fernandes Sampaio

Relatório de progresso realizado no âmbito de Projecto e Seminário,
do curso de licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2015/2016

Abril de 2016

Resumo

O projeto PDS16inEcplise consiste no desenvolvimento de uma ferramenta que visa facilitar a escrita de programas para o processador PDS16 usando a sua linguagem *assembly*. Este plug-in é, essencialmente, um editor de texto que integra funcionalidades para fazer uma verificação da semântica e da sintaxe em tempo de escrita de código. O desenvolvimento deste plug-in é baseado na framework Xtext integrada no Ambiente Integrado de Desenvolvimento (IDE) Ecplise [1].

Palavras-chave: Ambiente Integrado de Desenvolvimento; Processador PDS16; Assembly; Xtext; Eclipse; Plug-in.

Índice

Lista de Figuras

Lista de Tabelas

1 Introdução

1.1 Enquadramento

No domínio da Informática, um programa consiste no conjunto das instruções que define o algoritmo desenvolvido para resolver um dado problema usando um sistema computacional. Para que esse sistema possa realizar as operações definidas por estas instruções é pois necessário que as mesmas sejam descritas usando a linguagem entendida pela máquina, que consistem num conjunto de bits com valores lógicos diversos. Esta forma de codificação de algoritmos é bastante complexa e morosa, pelo que o processo habitual de desenvolvimento de um programa é feito com um maior nível de abstração, recorrendo a linguagens de programação. A Figura **1** mostra as diferentes fases deste processo quando aplicado ao domínio dos sistemas embebidos, em que as linguagens de programação mais utilizadas são o C e o C++.

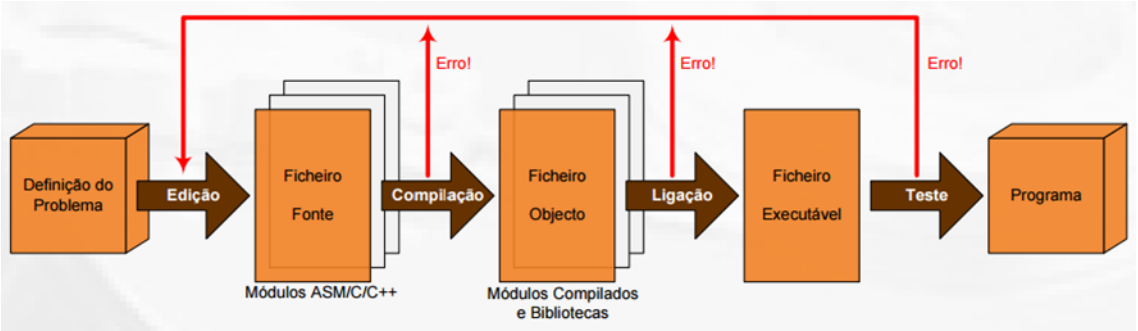

Figura 1 – Exemplo de um ciclo de desenvolvimento de um programa/aplicação. [1]

Após a definição do problema e elaboração do algoritmo para a sua solução, o programador começa a escrever o programa numa dada linguagem, resultando assim num ou vários ficheiros fonte. Estes são de seguida compilados através de um compilador ou *assembler*, que primeiramente verificam as regras sintáticas e semântica da linguagem e de seguida geram um ficheiro objeto correspondente a cada ficheiro fonte. O *linker* efetua a ligação entre os diversos ficheiros objeto que compõem o programa e as bibliotecas utilizadas, ficheiros partilháveis que podem conter código, dados e recursos em qualquer combinação. Deste último processo resulta um ficheiro com a descrição do algoritmo codificado pelos programadores em linguagem máquina, i.e. um ficheiro executável. Para garantir a correta implementação da solução desejada, são realizados um conjunto de testes sobre este ficheiro.

Os Ambientes Integrados de Desenvolvimento (*IDEs*) são hoje em dia um enorme apoio no desenvolvimento destes programas, uma vez que não só disponibilizam diversas ferramentas para apoio à produção do código, e.g. um editor de texto, a geração automática de código ou o

refactoring, como ainda possibilitam a interação com outras ferramentas e aplicações, como é o caso dos compiladores, *linkers*, *debuggers*, controladores de versão, etc.

Recorrendo a estas aplicações, um programador consegue ver a sua produtividade maximizada nas diferentes fases do processo de geração do ficheiro executável correspondente ao seu programa. Por exemplo, a geração automática de código permite poupar bastante tempo na escrita do código fonte do programa, bem como ter o código sempre bem indentado e estruturado. Já a funcionalidade de *syntax highlighting* facilita a leitura e análise do código fonte, para além de potenciar a deteção de erros de sintaxe e/ou de semântica. A utilização de um compilador integrado no IDE também permite acelerar o processo de geração do ficheiro executável, pois evita a saída do editor, a subsequente instanciação do compilador num processo aparte e, caso a compilação seja abortada devido a erros, a procura da linha associada a esse erro novamente no editor com vista à sua correção.

Atualmente, existem IDEs para quase todas as linguagens de programação em uso. Algumas destas aplicações suportam apenas uma linguagem de programação, como por exemplo o Kantharos ou o DRJava que apenas suportam PHP ou Java, respetivamente. Não obstante, há vários IDEs no mercado que permitem desenvolver programas e aplicações usando várias linguagens de programação, tais como o Eclipse [1] e o IntelliJ [2] cuja quota de mercado é, à data atual, superior a 80% [3]. Esta versatilidade é normalmente conseguida à custa da adição de *plug-ins* ou *add-ons*[1] específicos para uma dada linguagem de programação ao IDE. Estes podem ser criados a partir de bibliotecas que dão o suporte à criação dos mesmos.

Apesar da maioria destes IDEs e dos seus *plug-ins* e *add-ons* estarem normalmente associados ao desenvolvimento de programas utilizando linguagens de alto nível, como é o caso do C, C++, C# ou Java, muitas destas aplicações também oferecem suporte à codificação dos programas, ou dos seus módulos, usando linguagens de mais baixo nível, tal como o *assembly* (e.g. o Eclipse).

[1] Programas que ajudam adicionar novas funcionalidades aos plug-ins.

1.2 Motivação

A arquitetura PDS16 [4] foi desenvolvida no Instituto Superior de Engenharia de Lisboa (ISEL), em 2008, com o objetivo de suportar não só uma mais fácil compreensão mas também o ensino experimental dos conceitos básicos subjacentes ao tema "Arquitetura de Computadores". Esta arquitetura a 16 bits adota a mesma filosofia das máquinas do tipo *Reduced Instruction Set Computer* (RISC), oferecendo o seu *Instruction Set Architecture (*ISA) ao programador 6 registos de uso geral e cerca de 40 instruções distintas, organizadas em três classes: 6 instruções para controlo do fluxo de execução, 18 instruções de processamento de dados e 12 instruções de transferência de dados. O espaço de memória útil, que é partilhado para o armazenamento do código e dos dados dos programas, é endereçável ao byte e tem uma dimensão total de 64 kB.

Atualmente, o desenvolvimento de programas para esta arquitetura pode ser feito utilizando a própria linguagem máquina ou *assembly*. A tradução do código *assembly* para linguagem máquina é realizada recorrendo à aplicação dasm [5], que consiste num *assembler* de linha de comandos que apenas pode ser executado em sistemas compatíveis com o sistema operativo Windows da Microsoft.

Assim, o ciclo de geração de um programa passa por codificá-lo em linguagem *assembly* utilizando um editor de texto simples, tal como o Notepad, e posteriormente invocar a aplicação dasm a partir de uma janela de linha de comandos. Sempre que existam erros no processo de compilação, é necessário voltar ao editor de texto para corrigir a descrição *assembly* do programa e invocar novamente o *assembler*.

1.3 Objetivos

Com este trabalho pretende-se implementar um IDE para suportar o desenvolvimento de programas para o processador PDS16 usando a linguagem *assembly* e com as seguintes ferramentas e funcionalidades:

- Um editor de texto que integre ferramentas para fazer uma verificação da semântica e da sintaxe em tempo de escrita de código, de modo a que o programador possa ser alertado de eventuais erros na utilização da linguagem mais cedo e dessa forma otimizar a sua produtividade;
- *Syntax highlighting*, para permitir uma melhor legibilidade do código fonte;
- Integração com um *assembler*, para permitir a compilação dos programas sem necessidade de ter que abandonar o IDE e visualizar no editor de texto os eventuais erros detetados neste processo.

O IDE a desenvolver será baseado na plataforma Eclipse, atendendo à sua maior utilização na produção de programas e aplicações no domínio dos sistemas embebidos [6], onde se insere a utilização da arquitetura PDS16 no ISEL, e no facto dos alunos dos cursos de LEIC e LEETC do ISEL já terem experiência na utilização desta plataforma quando iniciam a frequência da unidade curricular Arquitetura de Computadores.

Para tal, será desenvolvido um *plug-in* para a arquitetura PDS16 utilizado a *framework* Xtext [7], que é uma *framework* genérica para o desenvolvimento de linguagens específicas de domínio (*DSL*). Para além da sua grande atualidade, a *framework* Xtext apresenta ainda a grande vantagem de, com base numa mesma descrição de uma DSL, permitir gerar automaticamente *plug-ins* também para a plataforma IntelliJ e para vários *browsers.*

2. PDS16 DSL – Linguagem de Domínio Especifico

O PDS16 trata-se de um processador a 16 bits que, entre outras, apresenta as seguintes características [4]:

- Arquitetura LOAD/STORE baseada no modelo de Von Neumman;
- ISA, instruções de tamanho fixo que ocupam uma única palavra de memória;
- Banco de registos (Register File) com 8 registos de 16 bits;
- Possibilidade de acesso à palavra (word) e ao byte.

Como já referido anteriormente, o seu ISA oferece aos programadores 3 conjuntos diferentes de instruções: transferência de dados, processamento de dados e controlo de execução, apresentando todas elas a mesma dimensão (16 bits).

Cada instrução pode ser dividida em 4 campos ordenados, seguindo a seguinte forma:

[Símbolo:] Instrução [Operando Destino][,Operando Fonte 1] [,Operando Fonte 2] [;comentário]

- **Símbolo:** Serve para referir o nome de uma variável, uma constante ou um endereço da memória, sendo que se trata de uma palavra, única no documento, seguida de ":"
- **Instrução:** Pode tratar-se de uma instrução PDS16 ou uma diretiva para o *assembler*.
- **Operando:** Tratam-se dos parâmetros da instrução em causa (caso a mesma possua algum), em que o seu tipo e número dependem da própria instrução.
- **Comentário:** O compilador ignora os seus caracteres. Existem 2 tipos de comentários: 1) comentário de linha: inicializado pelo caracter ";" e que abrange todos os caracteres até há mudança de linha; 2) comentário em bloco, inicializado por "/*" e terminado por "*/", abrangendo todos os caracteres entre eles.

2.1 Instruções

2.1.1 Acesso a memória de dados

As instruções de acesso a memória são as responsáveis pela leitura e escrita na memória, *load* e *store* respetivamente, sendo que no assembly de PDS16 se traduzem nas instruções "`ld`" e "`st`" e todas as suas derivadas.

Nestas instruções de transferência de dados entre o banco de registos e a memória, caso se pretenda o acesso ao byte e não à palavra, deverá acrescentar-se o caracter "b" à direita da mnemónica.

O acesso à memória pode ser feito usando duas formas de endereçamento distintas:

- Direto;
- Baseado e indexado, em que o índice pode ser definido por uma constante ou pelo valor de um registo.

O acesso direto trata-se de instruções que acedem exatamente à posição de memória indicada no seu operando:

```
ld rx, label_name
```

Esta instrução coloca no registo "`rx`" o contudo da posição de memória indicado pelo símbolo "`label_name`. Neste caso o acesso é direto pois não são efetuados quaisquer cálculos para definir a posição de memória requerida.

Por outro lado no que respeita ao acesso baseado e indexado, já são efetuados cálculos de modo a obter-se o endereço pretendido, uma vez que se tem um índice (constante) que deverá ser somado ao endereço base:

```
ldb rx, [ry, #const]
```

Neste caso o registo "`ry`" deverá ter o endereço base, ao qual ainda será adicionado o valor `const` (no caso de acesso a palavra 2*`const`) para se obter o endereço de memória de onde se obterá o valor que será transferido para o registo "`rx`".

O acesso baseado indexado também permite a definição do valor do índice recorrendo a um dos registos do processador, sendo que neste caso o endereço de memória a aceder é dado pelo resultado da soma entre dois registos:

```
ldb rx, [ry, rz]
```

2.1.2 Processamento de Dados

Estas instruções têm como objetivo o processamento dos dados através de operações aritméticas ou lógicas. Com exceção da instrução `NOT`, que apenas tem um operando fonte, todas as outras instruções têm dois operandos fonte. Regra geral, estes parâmetros correspondem a um dos 8 registos do processador. Contudo, em algumas instruções (`ADD`, `SUB`, `ADC` e `SBB`), o segundo operando pode corresponder a uma constante codificável em código binário natural com 4 bits. O resultado das operações realizadas é sempre um dos registos do banco de registos do processador.

Por definição, todas estas instruções afetam o registo de estado do processador (`PSW`), atualizando o valor das *flags* relativas aos indicadores relacionais e de excesso de domínio produzidos pela Unidade Lógica e Aritmética (ALU). Contudo, para algumas destas instruções, pode adicionar-se o caracter "f" depois da mnemónica para indicar que o registo `PSW` não deverá ser afetado. Nesta situação, caso o registo destino da operação seja o registo `R6` (i.e. o próprio `PSW`), este registo é afetado com o resultado da operação realizada.

Exemplo de operação envolvendo 2 registos como operandos fonte:

```
add rx, ry, rz
```

Esta instrução guarda em "`rx`" o resultado da soma entre os registos "`ry`" e "`rz`", modificando o registo PSW com o resultado das flags da operação.

Exemplo de operação envolvendo um registo e uma constante como operandos fonte:

```
add rx, ry, #const
```

Esta instrução guarda em "`rx`" o resultado da soma entre os registos "`ry`" e a constante "`const`", codificável em código binário natural com 4 bits, modificando o registo PSW com o resultado das flags da operação.

Para além das instruções acima mencionadas, existem duas instruções que permitem iniciar a parte baixa (bits 0 a 7) ou a parte alta (bits 8 a 15) de um registo com uma constante, codificada em código binário natural com 8 bits.

```
ldi rx, #const
ldih ry, #const
```

De notar que a constante `ldi` inicia os bits da parte alta do registo com o valor 0, ao passo que a instrução `ldih` não altera a parte baixa do registo aquando do carregamento da constante para a sua parte alta.

2.1.3 Controlo de Fluxo de Execução

Independentemente da natureza da instrução de salto considerada, i.e. salto incondicional ou salto condicional, o esquema de endereçamento subjacente é sempre o mesmo: endereçamento baseado e indexado, conseguido à custa da soma de uma constante de 8 bits a um dos oito registos do processador. Esta constante, codificada em código dos complementos, é previamente multiplicada por 2, uma vês que representa o número de instruções a saltar.

Exemplo de instrução de salto incondicional:

```
jmp LAB1
```

Este salto é calculado usando o "`PC`" como registo base (registo que contem a posição atual de execução) e um valor de constante que permita atingir o endereço de memória correspondente ao símbolo "`LAB1`".

Exemplo de instrução salto condicional:

```
jz rx,#const
```

Neste caso, o salto apenas irá ocorrer se a flag "`Z`" (zero) estiver ativa, ou seja tomar o valor lógico 1. Nessa situação, o salto será para a posição de memória dada pela soma do registo "`rx`" e a constante "`#const`".

Existe também uma instrução de salto incondicional com ligação (`JMPL`), cuja semântica é idêntica à anteriormente descrita.

2.2 Diretivas

Com o intuito de organizar o código existem as diretivas [8], estas definem as zonas de memória associadas a código ou a dados do programa. As diretivas não só dão corpo e estrutura ao código, mas visam ainda definir símbolos que podem ser utilizados como constantes dos programas.

Assim a linguagem definiu algumas secções base que são comuns em quase todos os programas:

- "`.DATA`" – zona de dados inicializados;
- "`.BSS`" – zona de dados não inicializados;
- "`.TEXT`" –zona de código.

Para além destas, permite ainda que o programador defina o seu próprio tipo de secção através da expressão "`.SECTION section_name`", substituindo `section_name` pelo nome da secção desejada.

Estas secções definem o início de posições de memória contiguas onde se encontrará o código. Para estabelecer o valor inicial para o contador de endereço de memória da secção corrente é utilizada outra diretiva, "`.ORG expression`", sendo que "`expression`" deverá conter o valor pretendido para o valor inicial.

Para ocupar estas posições de memória, para além de instruções, podem ser guardadas variáveis em memória, para isso existem as seguintes diretivas:

- ".WORD" – define uma/várias palavra/s em memória (16bits);
- ".BYTE" – define um/vários byte/s em memória (8bits);
- ".ASCII", ".ASCIIZ" – define uma string ascii não terminada por zero, e terminada por zero respetivamente (8bits por caracter);
- ".SPACE" – reserva espaço para vários bytes, com possibilidade de serem inicializados com um byte.

Existe também a possibilidade de serem atribuídos valores a símbolos através das diretivas ".EQU" e ".SET", sendo que a primeira é atribuído de forma permanente e o segundo temporária.

3. Framework Xtext

Xtext é uma *framework* para o desenvolvimento de linguagem de programação, as chamadas DSL, *Domain-Specific Languages*. Com o Xtext é possível definir uma linguagem com toda a sua gramatica resultando uma infraestrutura que incluí *parser*, *linker*, *typechecker*, compilador e também a possibilidade de ter um editor através do Ecplise [1], Intellij IDEA [2] e também através do browser.

Esta *framework* foi desenvolvida com o intuito de ser fácil de aprender e ser possível em poucos minutos descrever uma linguagem simples e também ser possível extrair o projeto em forma de um *plug*-in para a sua portabilidade entre máquinas.

Esta *framework* foi usada para a realização de um plug-in para uma linguagem de *assembly* PDS16. Não havendo nenhum editor de texto para a mesma, comprometemos a usar esta biblioteca.

Para começar a trabalhar tivemos de instalar o *plug*-in da *framework* no nosso IDE neste caso o Ecplise. Após criar um novo projeto e definir o nome da linguagem e a sua extensão, começamos a desenvolver. O primeiro passo é definir a sintaxe da linguagem, ou seja definir as regras.

3.1 Regras (Parser Rules)

Parser Rules são regras que definem uma sequência de outras regras conjugando com palavras-chaves. Como por exemplo o código da figura 2.

```
Statement:
    Instructions | Label | Directive;

Label:
    labelName=ID ':' value=(Label | LabelDirective | Instructions);

Instructions:
    Load | Store | Aritmetica | Logica | Jump | Nop | Ret;

JumpOp:
    ('jz' | 'je' | 'jnz' | 'jne' | 'jc' | 'jbl' | 'jnc' | 'jae' | 'jmp' | 'jmpl')
        (OperationWithOffset | op=ID | '$');

Nop: instruction='nop';

Ret: instruction=('ret' | 'iret') ;
```

Figura 2 - Código exemplo da definição das regras

Statement é uma regra que em que a sua definição é uma das referências para outra regra. Neste caso se virmos a regra "*Label"* podemos ver que a sua definição já contem palavras-chaves como ":" e um identificador "*labelName*" que é o tipo ID considerado um terminal. A regra "*Ret*" em que apenas é definida pela palavra-chave "*ret*" ou "*iret*". A regra "*Jump*" que é mais complexa pode ser definida por uma destas palavras-chaves, seguida pela regra "*OperationWithOffset*".

3.2 Regras Terminais

Regra terminal é também uma regra mas só que esta definida por uma sequência de caracteres também chamadas por *tokens rules* ou *lexer rules*. Um terminal pode retornar um tipo, por definição eles retornam sobre a forma de *String: ecore::EString*. Mas é possível converter o tipo de retorno para um típico especifico desde que seja uma instancia de *ecore::EDataType*. Para isso é necessário implementar a interface "*IValueConverter*" e criar o respetivo converter de *String* para o tipo pretendido.

```
terminal ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

terminal HEX returns ecore::EInt: SIGN? ('0x'|'0X')(('0'..'9')|('a'..'f')|('A'..'F'))+;
```

Figura 3 - Código exemplo da definição regras terminais

O primeiro terminal, *ID*, começa com um caracter de 'a' a 'z' ou por um '_' seguindo de nenhum ou mais caracteres incluindo números. O terminal *HEX* é a definição de um número hexadecimal, mas retornando um número inteiro em vez da *String*. Para que isso fosse possível foi necessário acrescentar um método a classe "*Pds16RunTimeModule*" o código seguinte da figura 4.

```
class Pds16asmRuntimeModule extends AbstractPds16asmRuntimeModule {

    override Class<? extends IValueConverterService> bindIValueConverterService() {
        return  Pds16asmValueConverter
    }

}
```

Figura 4 - Código da classe Pds16asmRuntimeModule

Este método apresentado na classe, retorna uma instância de "*Pds16asmValueConverter*" que por sua vez retorna uma instância de um "*ValueConverter*" específico dependendo do tipo do terminal, que têm a função para converter de *String* para inteiro.

3.3 Validadores

Existem certas regras de uma linguagem que não podem ser definidas, logo essas tem que ser verificadas no ato da compilação. Mas tal como um editor de texto o Xtext permite que sejam feitas essas verificações ao decorrer da escrita do código indicando o erro. Os validadores da *framework* permitem analisar determinado conteúdo e indicar ao utilizador do erro, retirando essa função ao compilador, pois não é possível compilar com erros de validações. No caso do nosso no trabalho verificamos os limites dos números conforme o tipo, por exemplo o offset8 que só pode estar compreendido entre minimio valor a 8 bits com sinal e o máximo valor a 8 bits com sinal. A figura 5 mostra o código que permite validar o valor.

```
@Check
def checkOffset8(Offset8OrLabel o){
    if(o.number == null)
        return;
    var Integer value = o.number.value;
    if(value < MIN_8BIT_WITH_SIGNAL  || value > MAX_8BIT_WITH_SIGNAL)
        error('Number should be between' + MIN_8BIT_WITH_SIGNAL + ' and ' + MAX_8BIT_WITH_SIGNAL,
                Pds16asmPackage.Literals.OFFSET8_OR_LABEL__NUMBER,
                "Invalid Number")
}
```

Figura 5 - Exemplo de um validador

3.4 Compilador

A própria *framework* disponibiliza a opção de criar um compilador, mas nesta etapa do projeto decidimos usar um compilado externo, o DASM [5]. Para isso, é feita uma chamada ao compilador externo através do "*ProcessBuilder*", passando como input o ficheiro fonte, ASM, capturando o output após a compilação. Com esse output analisamos e conseguimos determinar se o ficheiro fonte tem erros. Caso tenha erros, assinalamos no ficheiro fonte com os erros na respetiva linha e com a mensagem que o compilador deu.

4. Progresso do Projeto

Relativamente à calendarização do trabalho que havia sido apresentada na "Proposta de Projeto", decorridas estas 7 semanas de realização de trabalho podemos concluir que a execução do projeto está a decorrer conforme o previsto, apesar de algumas das suas fases terem tido uma duração ligeiramente superior ao inicialmente previsto. Ainda assim, no global, a execução do projeto não está atrasada, tendo já sido alcançados os seguintes objetivos:

- **Estudo do *Assembly* PDS16:** Estudo da linguagem com base na documentação de Arquitetura de Computadores, capítulos 13 [1] e 15 [2].

- **Estudo da *Framework* Xtext:** Estudo da *framework* com base na documentação disponibilizada na Web.

- **Elaboração Proposta do Projeto:** Foi elaborada a proposta do projeto depois do estudo do *assembly* PDS16 e da *framework* que vai servir de suporte ao projeto, tendo sido realizada uma proposta de calendarização com os prazos a cumprir.

- **Implementação do ASM PDS16:** Foi definida a sintaxe gramatical da linguagem utilizando a *framework* Xtext [3], criando também validadores para certos aspetos da linguagem que ajudam ao utilizador informando os erros.

- **Gerador (Utilizando PDS16):** Para compilar o ficheiro foi invocado o compilador dasm, passando um ficheiro fonte como entrada e recebendo o resultado da compilação como saída. Esta informação é utilizada para verificação da existência de erros de compilação e, caso existam, assinalá-los no ficheiro fonte com a mensagem de erro produzida pelo *assembler*.

Face ao exposto, à data atual prevemos cumprir a calendarização inicialmente definida que se apresenta na **Erro! A origem da referência não foi encontrada.**.

	Relatorio e Entrega Versão Final (23 julho)	"Tab" de opção para o compilador - Opcional	Relatório Final	Deploy IntelliJ	Deploy Eclipse	Cartaz e Versão beta (13 Junho)	Testes	Relatório de Progresso	Gerador (utilizando dasm)	Implementação do ASM PDS16	Elaboração Proposta do Projeto	Estudo da Framework Xtext	Estudo do Assembly PDS16
Start Date	11/jul	20/jun	6/jun	30/mai	23/mai	16/mai	2/mai	25/abr	18/abr	28/mar	21/mar	14/mar	7/mar
Days to Complete	14	21	14	7	7	7	14	7	7	21	7	7	7

Tabela 1 - Diagrama de Gantt relativo à previsão da execução do trabalho.

Referências

[1] “IDE Ecplise,” [Online]. Available: http://www.eclipse.org.

[2] “Intellij, IDE,” [Online]. Available: https://www.jetbrains.com/idea/.

[3] O. White, “IDEs vs. Build Tools: How Eclipse, IntelliJ IDEA & NetBeans users work with Maven, Ant, SBT & Gradle,” 2014. [Online]. Available: http://zeroturnaround.com/rebellabs/ides-vs-build-tools-how-eclipse-intellij-idea-netbeans-users-work-with-maven-ant-sbt-gradle/. [Acedido em 25 03 2016].

[4] J. Paraíso, “PDS16. Arquitetura de Computadores – Textos de apoio às aulas teóricas (págs. 13-1 – 13-27),” Lisboa, 2011.

[5] J. Paraíso, “Desenvolvimento de Aplicações. Arquitetura de Computadores – Textos de apoio às aulas teóricas (págs. 15-2 – 15-5),” Lisboa, 2011.

[6] C. Ajluni, “Eclipse Takes a Stand for Embedded Systems Developers,” [Online]. Available: http://www.embeddedintel.com/search_results.php?article=142. [Acedido em 30 03 2016].

[7] “Xtext 2.5 Documentation, Eclipse Foundation,” 2013. [Online]. Available: http://www.eclipse.org/Xtext/documentation/2.5.0/Xtext%20Documentation.pdf. [Acedido em 05 02 2016].

[8] J. Paraíso, “QuickRef_V2,” [Online]. Available: http://pwp.net.ipl.pt/cc.isel/ezeq/arquitetura/sistemas_didaticos/pds16/hardware/QuickRef_V2.pdf.

[9] T. Dias, “Elaboração de Ficheiros Executáveis,” 2013. [Online]. Available: https://adeetc.thothapp.com/classes/SE1/1314i/LI51D-LT51D-MI1D/resources/2334. [Acedido em 27 03 2016].