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 sintaxe e da semântica em tempo de escrita de código. O desenvolvimento desta ferramenta é baseado na framework Xtext integrada no Ambiente Integrado de Desenvolvimento (IDE) Ecplise.

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 programável. Para que esse sistema possa realizar as operações definidas por estas instruções é pois necessário que as mesmas sejam apresentadas usando a linguagem entendida pela máquina, que consiste 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++.

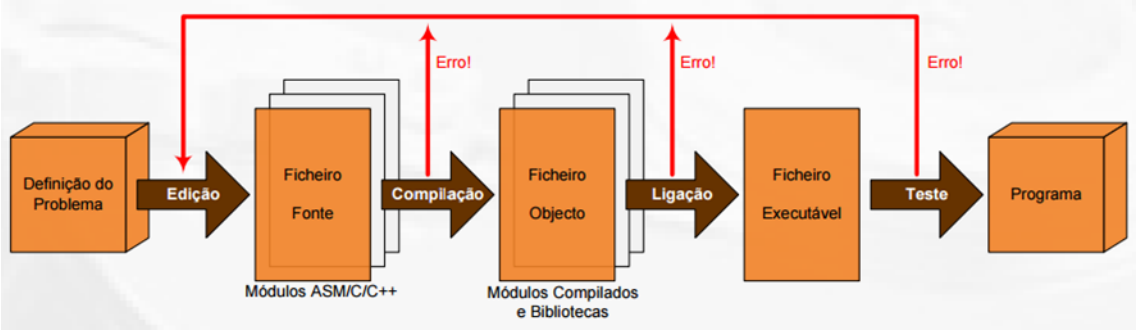


Figura 1 – Exemplo de um ciclo de desenvolvimento de um programa/aplicação. **[1]**

Após a definição do problema e elaboração do algoritmo para a sua solução, o programador começa a implementar o programa usando uma dada linguagem, obtendo-se assim um ou vários ficheiros fonte. De seguida, estes são traduzidos para a linguagem entendida pela máquina recorrendo a um compilador ou *assembler*, que primeiramente verificam as regras sintáticas 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 localizável em memoria, i.e. um ficheiro executável. Para garantir a correta implementação da solução desejada, são realizados um conjunto de testes sobre este ficheiro antes de se dar por concluído o desenvolvimento do programa.

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. A funcionalidade de *syntax highlighting* também facilita a leitura e análise do código fonte, para além de potenciar a deteção de erros de sintaxe. 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 [2] 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 [3] e o IntelliJ [4] cuja quota de mercado é, à data atual, superior a 80% [5]. 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.

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 Processador Didático Simples a 16 bits (PDS16) [6] 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 [7], que consiste num *assembler* de linha de comandos que apenas pode ser executado em sistemas compatíveis com o sistema operativo Windows da Microsoft. Desta forma, 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 sintaxe em tempo de escrita de código, de modo a que o programador possa ser alertado para 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 [8], onde se insere a utilização da arquitetura PDS16 no ISEL, e no facto dos alunos dos cursos de Licenciatura em Engenharia Informática e de Computadores (LEIC) e Licenciatura em Engenharia Eletrónica e Telecomunicações e de Computadores (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 [9], 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 Arquitetura PDS16

2.1 Introdução

O PDS16 trata-se de um processador a 16 bits que adota o modelo de *Von-Neumann* , ou seja, que utiliza o mesmo espaço de memória tanto para código como para dados. Este processador apresenta as seguintes características [6]:

- 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.

Para além destas caraterísticas, o processador também tem um mecanismo de interrupção que consiste na verificação de um pino ao fim de cada execução de uma instrução, e caso este esteja ativo (*active-low*) é gerada uma chamada a uma rotina ISR (*Interrupt Service Routine*) que executará a ação pretendida por quem interrompeu. Mas um das dificuldades que essa interrupção trás depois de executar código ISR é voltar a colocar os registos nos estados originais e retornar o programa no estado inicial antes da interrupção (registo PC). Para que uma interrupção tenha sucesso é necessário que o estado de execução seja preservado, neste caso salvaguardar o valor corrente do registo PC no do registo *Link*. Em relação aos restantes registos ou fica pela responsabilidade da ISR usar uma estrutura de dados ou existe uma duplicação de vários registos do CPU e que são comutados no momento da interrupção, pois a arquitetura do PDS16 não suporta o uso de um *Stack*.

As seguintes tabelas, mostram a sintaxe das instruções que o processador PDS16 suporta, estando elas divididas em secções como cinco secções: Load, Store, Aritmétrica, Lógica e Jump.

Operação		Assembly	Acção
Load	Immediate into low half word	ldi rd,#immediate8	rd = 0x00 immediate8
	Immediate into high word	ldih rd,#immediate8	rd = 0ximmediate8, LSB(rd)
	Direct	ld{b} rd,direct7	rd = [direct7]
	Indexed	ld{b} rd,[rbx,#idx3]	rd = [rbx+idx3]
	Based indexed	ld{b} rd,[rbx,rix]	rd = [rbx+rix]
Store	Direct	st{b} rs,direct7	[direct7] = rs
	Indexed	st{b} rs,[rbx,#idx3]	[rbx+idx3] = rs
	Based indexed	st{b} rs,[rbx,rix]	[rbx+rix] = rs
Aritmétrica	Add registers	add{f} rd,rm,rn	rd=rm+rn
	Registers with CY flag	addc{f} rd,rm,rn	rd=rm+rn+cy
	Constant	add{f} rd,rm,#const4	rd=rm+const4
	Constant with CY flag	adc{f} rd,rm,#const4	rd=rm+const4+cy
	Sub registers	sub{f} rd,rm,rn	rd=rm-rn
	Registers with borrow	sbb{f] rd,rm,rn	rd=rm-rn-cy
	Constant	sub{f} rd,rm,#const4	rd=rm-const4
	Constant with CY flag	sbb{f} rd,rm,#const4	rd=rm-const4-cy
Lógica	AND registers	anl{f} rd,rm,rn	rd=rm & rn
	OR registers	orl{f} rd,rm,rn	rd=rm | rn
	XOR registers	xrl{f} rd,rm,rn	rd=rm ^ rn
	NOT registers	not{f} rd,rm	rd=~rs
	Shift left register	shl rd,rm,#cont4,sin	rd=(rm,sin)<<const4
	Shift right register	shr rd,rm,#cont4,sin	rd=(rm,sin)>>const4
	Rotate right least significant bit	rrl rd,rm,#cont4	rd=(rm,l)>>const4
	Rotate right must significant bit	rrm rd,rm,#cont4	rd=(rm,m)>>const4
	Rotate with carry right	rcr rd,rm	rd=(rm,cy,r)
	Rotate with carry left	rcl rd,rm	rd=(rm,cy,l)
Jump	If zero	rbx,#offset8	If(Z) PC=rbx+(offset8<<1)
	If not zero	rbx,#offset8	If(!Z) PC=rbx+(offset8<<1)
	If carry	rbx,#offset8	If(CY) PC=rbx+(offset8<<1)
	If not carry	rbx,#offset8	If(!CY) PC=rbx+(offset8<<1)
	Unconditional	rbx,#offset8	PC=rbx+(offset8<<1)
	Unconditional and link	rbx,#offset8	R5=PC; PC=rbx+(offset8<<1)
No Op	No operation	nop	
Software interrupt	Interrupt return	iret	PSW=r0i; PC=r5i

Tabela 1 - Sintaxe das Instruções PDS16

Palavras-chave	Descrisão
rd	Registo destino
rs	Registo fonte
rbx	Registo base
rix	Registo de indexação que é multiplicado por dois se o acesso é a uma word.
Rm/rn	Registos que contêm os operando
immediate8	Constante de 8 bits sem sinal
direct7	7 bits sem sinal e que corresponde aos endereços dos primeiros 128 bytes ou 64 words.
idx3	índice de 3 bits sem sinal a somar ao registo base RBX
#const4	Constante de 4 bits sem sinal
offset8	Constante de 8 bits com sinal [-128..+127] *words*
rbx	Registo base
f	(*flags*) colocado à direita da mnemónica indica que o registo PSW não é atualizado
sin	(*serial in*) valor lógico do bit a ser inserido à esquerda ou à direita.

Tabela 2 - Palavras-chave da Sintaxe PDS16

2.2 Modelo de programação (ISA)

Como já referido anteriormente, o ISA do PDS16 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.

2.2.1 Mapa de memória

Como este processador segue a arquitetura de *Von-Neumann*, é usado apenas uma memória para código e dados de 32K*16. O *bus* de dados é de 16 bits (*word*), mas o processador permite realizar leituras e escritas de 8 bits (*byte*). No caso da leitura de oito *bits*, são lidos sempre 16 *bits* da memória, mas é o processador que gere os *bytes* a ler. Por exemplo para um endereço par é selecionado o *byte* de maior peso e para um endereço ímpar é selecionado o *byte* de menor peso. Em relação ao programa em si, é necessário que as instruções estejam sempre alinhadas a 16 *bits* ou seja em endereços pares.

2.2.2 Registos

O *Register file* da estrutura do PDS16 é composto por 8 registos de R0 a R7 de 16 *bits* cada. Os registos de R0 até ao R4 inclusive são registo usados para a manipulação de dados temporários sem recorrer a memória para armazenar. O registo R7, com o nome de PC (*Program Counter*), guarda o endereço de memória da instrução exatamente a seguir à que está a ser executada de momento. Este registo é útil pois certas operações que usam *offset*, como o *jump*, podem somá-lo ao endereço da instrução corrente, resultando num salto para determinada instrução. O registo R5, denominado *Link*, foi criado com o intuito de salvaguardar o valor corrente do registo R7, no caso de um salto com ligação, para ser possível regressar ao ponto inicial quando necessário ou no caso de interrupção onde é assegurado o retorno ao endereço exatamente a seguir ao da evocação da

rotina. Por fim o registo R6, o PSW, serve de controlo de *flags* onde a cada uma corresponde um *bit*. Conforme a seguinte imagem nem todos os bits estão ocupados com *flags*:

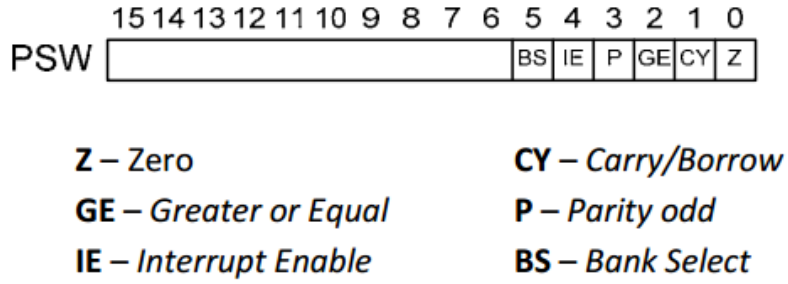


Figura 2 – *Flags* do registo PSW

2.2.3 Instruções

2.2.3.1 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 conteúdo 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]
```

2.2.3.2Processamento 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.

2.2.3.3 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.

2.3 Assemblador DASM

Seja qual for a linguagem de programação adotada para desenvolver um programa existe a necessidade de compilar o código fonte produzido para se obter o correspondente código interpretável pela máquina. Para o processador PDS16, foi criado um assemblador denominado DASM [7], uni modelar, que a partir de um ficheiro de texto escrito em linguagem assembly PDS16 produz o ficheiro com a designação correspondente em linguagem máquina, i.e. o ficheiro executável do programa. Este ficheiro, com extensão HEX, adota o formato Intel HEX80. É portanto um ficheiro de texto constituído por caracteres ASCII organizados em tramas, contendo cada trama uma marca de sincronização, o endereço físico dos *bytes* contidos na trama e um código para deteção de erros de transmissão.

Sendo o DASM um assemblador didático uni modular, ou seja, não permite o desenvolvimento de aplicações usando múltiplos ficheiros fontes, não existe a necessidade de uma ferramenta de ligação. Pelo mesmo motivo a localização em memória das instruções e das variáveis e constantes é estática e estabelecida no ficheiro fonte.

A execução do programa DASM também produz um ficheiro com extensão LST. Este consiste numa listagem das operações realizadas pelo DASM, pelo qual o texto original de cada instrução no ficheiro fonte, acrescido do endereço de memória em que foi localizado e do respetivo código maquina. Caso existam erros de compilação, os mesmos são assinalados na respetiva instrução com uma mensagem identificadora do seu tipo e da possível causa.

2.3.1 Diretivas

Para além das instruções assembly PDS16, o assemblador DASM reconhece e processa um outro conjunto de comandos [10]. Estes comandos visam não só facilitar a organização em memória do código e dos dados dos programas, mas também a utilização de símbolos para representação de valores, e.g. endereços e constantes.

No que respeita à organização dos programas em memória, é possível definir-se as três secções base geradas por quase todos os compiladores:

1. "`.DATA`" – que aloja as variáveis globais com valor inicial;
2. "`.BSS`" – que aloja as variáveis globais sem valor inicial;
3. "`.TEXT`" – que aloja as instruções do programa;

Para além destas secções, permite ainda que o programador defina outras secções. Para tal, deve usar-se a diretiva *.section* para definir uma expressão do tipo "`.SECTION section_name`", em que `section_name` corresponde ao nome da secção desejada.

De notar que estas diretivas apenas definem o início de uma zona de memória contígua onde se podem localizar as instruções e os valores definidos para as variáveis . Para estabelecer o valor do endereço em que uma secção deverá ser localizada deve usar-se a diretoria *.org* que

define uma expressão do tipo: "`.ORG  expression`", em que "`expression`" deverá corresponder o valor de endereço pretendido.

O assemblador DASM disponibiliza um outro conjunto de diretivas que permite reservar e definir o valor inicial de posições de memória. As diretivas *.word* e *.byte* podem definir dois tipos de expressões:

1. ".WORD" – define uma/várias palavra/s em memória;
2. ".BYTE" – define um/vários byte/s em memória;
3. ".ASCII", ".ASCIIZ" – define uma string ascii não terminada por zero, e terminada por zero, respetivamente;
4. ".SPACE" – reserva espaço para um ou vários bytes, com possibilidade de serem inicialização com um valor definido pelo programador.

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

3.1 Introdução

Xtext é uma *framework* para o desenvolvimento de linguagens de programação, as denominadas DSL (*Domain-Specific Languages*). Com o Xtext é possível definir uma linguagem com toda a sua gramática resultando uma infraestrutura que inclui *parser*, *linker*, *typechecker*, compilador e também a possibilidade de ter um editor utilizando uma plataforma do Eclipse [3], IntelliJ IDEA [4] ou browsers.

Decidimos utiliza-la para a realização de um *plug-in* para a linguagem de *assembly* PDS16, utilizando como recurso o livro "Implementing Domain-Specific Languages with Xtext and Xtend" [11].

Para o desenvolvimento de um *plug-in* utilizando esta framework, é necessário instalar o *plug*-in da *framework* no IDE de desenvolvimento, neste caso o Eclipse, e a criação de um "*Xtext Project*".

3.2 Arquitetura

Xtext é uma framework Eclipse desenvolvida que tem como base a linguagem de programação Java. Para desenvolver uma linguagem primeiro tem que ser definida a sintaxe da mesma, neste caso a definição de uma gramática será o primeiro passo.

A *framework* Xtext oferece ao utilizador a oportunidade de implementar diversas funcionalidades como o *highlighting*, validação e *parser*. Estas podem ser implementadas em Java, ou numa linguagem específica criada à base de Java, o Xtend. A linguagem de programação Xtend está totalmente integrada com a linguagem Java obtendo assim todos os recursos e suporto que o Java têm como as bibliotecas, e oferendo outras funcionalidades como o *type inference*, métodos de extensão, expressões lambdas e *multi-line template expressions*. Todos os aspetos da DSL implementados em Xtext podem ser implementados em Xtend em vez do Java pois é mais fácil de utilizar e permite escrever código mais legível.

Após definir a gramática no ficheiro com a extensão *".xtext"*, serão geradas todas as classes necessárias para poderem ser implementadas as funcionalidades disponíveis de uma forma mais prática, com o manuseamento de objetos e referências que refletem a linguagem criada.

3.3 Gramática

Para definir uma linguagem de programação, temos de estudar a sua gramática e ter em atenção as possíveis formas de escrever uma determinada regra da sintaxe. O nosso projeto visa criar um *plug-in* para *assembly* do processador PDS16, nesse sentido estudamo-lo através a documentação [10], [6] e [7].

A linguagem é definida através de regras que podem referenciar outras regras ou palavras-chave. Por cada regra definida é criada uma classe com métodos e atributos conforme a definição da regra, mas qualquer regra poderá depender de outra regra. Para isso a geração automática das classes cria também a dependência das classes com as outras. Como por exemplo nas seguintes regras da Figura 3:

```
PDS16ASM:
    instuctions+=Statement*;

Statement:
    Instructions | Label | Directive;

Logica: Anl | Orl | Xrl | Not | Shl | Shr | Rr | Rc;

Anl: ('anl' | 'anlf') OperationsWithThreeRegisters;

OperationsWithThreeRegisters: rd=Register ',' rm=Register ',' rn=Register ;
```

Figura 3 – Excerto de código de uma gramática Xtext

Na Figura 3 podemos ver que a regra *OperationsWithThreeRegisters* depende de *Anl* e que por sua vez depende de *Logica* e esta de *Instructions* e assim consecutivamente ate chegar a regra *PDS16ASM*.

Essa dependência é tratada pelo Xtext gerando automaticamente classes em Java quando o *Modeling Workflow Engine 2* (MWE2 [12]) é corrido, resolvendo essa dependência pela extensão a classe da que depende criando assim uma hierarquia entre as regras de uma DSL, como o exemplo da Figura 4.

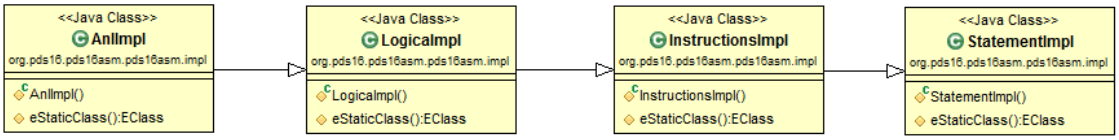


Figura 4 - Classes geradas pela framework

O MWE2 trata-se de um gerador de código configurável, que permite ao utilizador descrever composições de objetos arbitrários por meio de uma sintaxe simples e concisa que permite declarar instâncias de objetos, valores de atributos e referências.

Esta geração automática não é feita ao acaso, desta forma é possível ter em *runtime* uma estrutura de toda a hierarquia da gramática, para que possa ser usada noutras funcionalidades.

3.3.1 Regras da gramática

Parser Rules são regras que definem uma sequência de outras regras conjugadas com palavras-chaves. Como por exemplo o código da Figura 5.

```
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 5 - Código exemplo da definição das regras

Pegando como exemplo a nossa implementação da gramática, *Statement* é uma regra que na sua definição é uma das referências para outra regra. Neste caso na 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. "*Ret*" e "*Nop"* são apenas é constituídas por palavras-chave, não dependendo de nenhuma outra regra. A regra "*Jump*" que é mais complexa pode ser definida por uma destas palavras-chaves, seguida pela regra "*OperationWithOffset*".

Terminal Rules tratam-se de um tipo de regra que é definida por uma sequência de caracteres (*token)* também denominada por *token rule* ou *lexer rule*.

```
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 6 - 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.

Um terminal pode retornar um tipo que por definição se trata de uma *String*. Mas é possível manipular o tipo de retorno para um tipo específico. O terminal *HEX* é a definição de um número hexadecimal, mas retornando um inteiro em vez de *String*. Para que isso fosse possível foi necessário redefinir o método "*bindIValueConverter*" na classe que representa o *RunTimeModule* do projeto em questão, neste caso "*Pds16RunTimeModule*", Figura 7. Este método retorna a classe responsável pela conversão dos tipos de retorno das regras definidas na gramática.

```
class Pds16asmRuntimeModule extends AbstractPds16asmRuntimeModule {

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

}
```

Figura 7 - Código da classe Pds16asmRuntimeModule

A classe Pds16asmValueConverter implementa a interface *IValeuConverterService*, onde através de anotação de métodos, são definidas as regras em que se pretende converter o tipo de retorno, e qual a classe responsável pela conversão, Figura 8.

```
class Pds16asmValueConverter extends DefaultTerminalConverters implements IValueConverterService{

    @Inject
    private HEXValueConverter hexValueConverter

    @ValueConverter(rule = "HEX")
    def IValueConverter<Integer> getHexConverter() {
        return hexValueConverter;
    }
}
```

Figura 8 - Excerto da classe PDS16asmValueConcerter

Como presente na figura, a anotação "*@ValueConverter(rule="HEX")*", indica que o método por ela anotado, retornará um conversor do tipo de retorno (neste caso para *Integer*) para a regra com o nome *"HEX"*, sendo que se trata de uma instância da classe *HEXValueConverter*, que por sua vez terá de implementar a interface *IValueConverter*.

3.3.2 Definição dos elementos do analisador de regras

Existem certas regras de uma linguagem, como as regras de semântica, que não podem ser definidas através das regras anteriores, logo essas têm 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 caso exista um erro, retirando essa função ao compilador, pois não é possível compilar com erros de validação. No caso do nosso no trabalho verificamos os limites dos números conforme o tipo, por exemplo o *offset8* que se trata de um valor a 8 bits com sinal. A Figura 9 mostra o código que permite essa validação.

```
@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 9 - Exemplo de um validador

3.4 Integração com a plataforma Eclipse

A framework Xtext disponibiliza a biblioteca de desenvolvimento de linguagens sobre a forma de *plug-in*. Para fazer uso da mesma, esta pode ser instalada em várias plataformas suportadas, adicionando assim novas funcionalidades aos IDEs, neste caso o Eclipse.

3.4.1 Configuração do plug-in

Após desenvolver a gramática da linguagem Assembly PDS16 usando a framework Xtext, decidimos disponibilizar o *software* desenvolvido para poder ser utilizado noutras máquinas. Para tal foi necessário criar um *plug-in* que incorporasse as bibliotecas que permitem ter um editor de texto com as funcionalidades implementadas.

Para gerar o *plug-in* começámos por criar um *Feature Project* onde foram adicionados os projetos, e respetivas dependências, que o *plug-in* final deverá conter para o correto funcionamento do editor de texto.

De seguida foi criado um projeto do tipo *Update Site* para conseguirmos criar e disponibilizar o *plug-in* de modo a poder ser instalado remotamente, alojando-o numa página web.

Neste projeto tivemos apenas de referenciar o *feature project* criado anteriormente e efetuar a operação *build all*, que gera todos os ficheiros necessários para a instalação do mesmo. No processo de *deploy* tivemos em conta o controlo de versões do *plug-in*, podendo este ser atualizado pelo utilizador quando for lançado uma nova versão do software.

Para uma descrição mais pormenorizada, consultar “A.1 - Deploy do plug-in para o Ecplise”.

3.4.2 Syntax Highlight

Uma das características do *plug-in* é o suporte *highlighting* para ajudar o utilizador a distinguir os vários tipos que a gramática pode suportar. No nosso caso, dividimos a coloração da sintaxe em cinco tipos: diretivas, instruções, comentários, *labels* e texto. Cada tipo tem a sua específica cor e estilo de letra.

Para colorir a sintaxe da gramática, a biblioteca Xtext oferece a classe *DefaultHighlightingConfiguration* que implementa a *IHighlightingConfiguration*. Esta contem cores predefinidas para certos tipos, no entanto resolvemos criar a classe *Pds16HighlithingConfiguarion* para associar a cada tipo uma cor e um formato, como se pode verificar na Figura 10.

```
class Pds16asmHighlightingConfiguration extends DefaultHighlightingConfiguration{

    public static final String DIRECTIVES = "Directives";
    //...

    override configure(IHighlightingConfigurationAcceptor acceptor) {
        addType(acceptor,DIRECTIVES, 127, 0, 85, SWT.BOLD);
        //...
    }

     def addType( IHighlightingConfigurationAcceptor acceptor, String s, int r, int g, int b, int style ){
         var TextStyle textStyle = new TextStyle();
         textStyle.setBackgroundColor(new RGB(255, 255, 255));
         textStyle.setColor(new RGB(r, g, b));
         textStyle.setStyle(style);
         acceptor.acceptDefaultHighlighting(s, s, textStyle);
     }

}
```

Figura 10- Excerto de código de Pds16HighlightingConfiguration

Aqui é redefinido o método *configure* que regista no parâmetro recebido (*aceptor*) todos os estilos que o utilizador pretenda utilizar, associando-os a um *id*.

Após registar os estilos a utilizar, ainda é necessário associa-los aos *tokens* da sintaxe gramatical para que os mesmos sejam aplicados. Neste caso, *tokens* são os nomes das regras e terminais, e também caracteres como a virgular e parênteses. Para efetuar esta associação criamos a classe *Pds16TokenAtributeIdMapper* que estende de *DefaultAntlrTokenAttributeIdMapper*, Figura 11.

```
class Pds16asmTokenAtributeIdMapper extends DefaultAntlrTokenToAttributeIdMapper{

    override String calculateId(String tokenName, int tokenType) {
        switch(tokenType){

            //...
            case InternalPds16asmLexer.Word:
                return Pds16asmHighlightingConfiguration.DIRECTIVES

            case InternalPds16asmLexer.Add:
                return Pds16asmHighlightingConfiguration.RULES

            case InternalPds16asmLexer.RULE_SL_COMMENT:
                return Pds16asmHighlightingConfiguration.COMMENTS

            case InternalPds16asmLexer.RULE_IDLABEL:
                return Pds16asmHighlightingConfiguration.LABEL

            case InternalPds16asmLexer.RULE_STRING:
                return Pds16asmHighlightingConfiguration.TEXT
        }
        return super.calculateId(tokenName,tokenType);
    }
}
```

Figura 11 - Excerto de código de Pds16TokenAtributeIdMapper

O método redefinido, *caculateId*, trata de retornar o *id* do estilo a associar a cada *token*, dado o nome do *token* associado a cada regra, e o id da mesma, *tokenName* e *tokenType* respetivamente.

Depois de ter ambas as classes definidas, apenas é necessário o registar que pretendemos utiliza-las em vez das classes que calculam o *highlighting* por definição. Para isso é necessário redefinir na classe que define o *UiModule* do projeto, neste caso *AbstractPds16UiModule*, os métodos responsáveis por este trabalho, Figura 12.

```
class Pds16asmUiModule extends AbstractPds16asmUiModule {

    def Class<? extends IHighlightingConfiguration> bindILexicalHighlightingConfiguration () {
       return Pds16asmHighlightingConfiguration;
    }

    def Class<? extends AbstractAntlrTokenToAttributeIdMapper> bindAbstractAntlrTokenAtributeIdMapper() {
        return Pds16asmTokenAtributeIdMapper;
    }

}
```

Figura 12 - Código da classe AbstractPds16asmUiModule

3.4.3 Gerador

A framework disponibiliza a opção de criar um compilador, mas nesta etapa do projeto decidimos usar um assemblador externo, o DASM. Existe uma classe, *Pds16asmGenerator*, que é responsável para eventual geração de código após a escrita de um programa. Esta classe contem apenas a definição de um método, *doGenerate*. Este método é chamado automaticamente, por definição, sempre que um ficheiro já tenha sido validado e analisado, ou seja sempre que já não contenha qualquer erro de validação.

```
class Pds16asmGenerator extends AbstractGenerator {
    val String SYSTEM_ENV_DASM = "DASM_PATH"

    override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {

        if(!existsEnd(resource)){
            val errors = new ArrayList<LinedError>()
            errors.add(new LinedError("Missing '.end' at the end of file",1))
            generateErrors(errors,resource)
            return
        }

        val InputStream output = executeDasm(resource)

        val List<LinedError> errors = DasmErrorParser.getErrorsFromStream(output);

        generateErrors(errors, resource);
    }

    def boolean existsEnd(Resource resource) {//verifica a existencia de ".end" no ficheiro fonte

    def void generateErrors(List<LinedError> errors, Resource resource) {
        if(!errors.isEmpty){
            var rel = resource.URI.toString().substring("platform:/resource".length)
            val sharedFile = ResourcesPlugin.workspace.root.findMember(rel);
            errors.forEach[error |
                {
                    val marker = sharedFile.createMarker(IMarker.PROBLEM)
                    marker.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_ERROR)
                    marker.setAttribute(IMarker.LINE_NUMBER, error.line)
                    marker.setAttribute(IMarker.MESSAGE, error.description)
                }
            ]
        }
    }

    def InputStream executeDasm(Resource resource){//evoca o assemblador dasm retornando o output do mesmo
}
```

Figura 13 - Excerto de código da classe Pds16asmGenerator

Na nossa implementação do método *doGenerate*(Figura 13Figura **12**), começámos por verificar se existe o elemento "*.end*" no ficheiro, pois caso não exista, o ficheiro não será válido, e não é necessária a chamada ao assemblador.

Após esta verificação, evocamos o assemblador DASM com o *path* do programa em questão através de um *ProcessBuilder* (classe usada para criar processos do sistema operativo). Após ser feita esta chamada, é capturado o *output* retornado pelo processo em formato de

InputStream<String>. Este é processado de forma a obter eventuais erros, para isso foi criada a classe *DasmErrorParser*, contendo apenas um método estático, que dado um *InputStream* recebido como parâmetro retorna uma lista de objetos do tipo *LinedError* que contém a descrição e a linha do erro do ficheiro fonte.

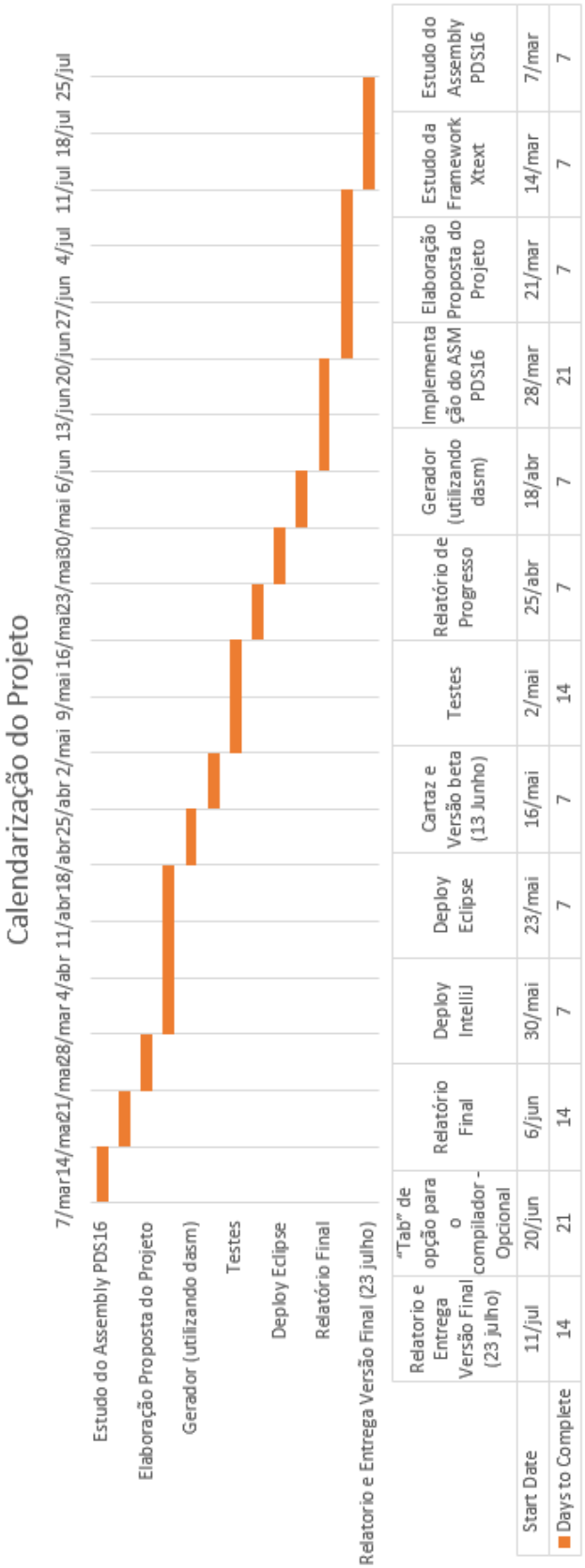
Tendo uma lista de erros, iteramos sobre a mesma, e por cada erro criamos uma marca, *IMarker*, no ficheiro fonte, com a gravidade da mensagem, neste caso erro (*IMarker.SEVERITY_ERROR*), na respetiva linha e com a descrição gerada pelo assemblador DASM.

4 Progresso do Projeto

Relativamente à calendarização do trabalho que havia sido apresentada na "Proposta de Projeto", decorridas estas 14 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 antecipado. Ainda assim, no global, a execução do projeto não está atrasada, tendo já sido alcançados os seguintes objetivos:

- **Estudo do *Assembly* PDS16:** Estudo da arquitetura PDS16 com base na documentação utilizada na UC Arquitetura de Computadores do ISEL, capítulos 13 [6] e 15 [7].
- **Estudo da *Framework* Xtext:** Estudo da *framework* com base na documentação disponibilizada na Web e em bibliografia de referência [9].
- **Elaboração Proposta do Projeto:** Foi elaborada a proposta do projeto depois do estudo do *assembly* PDS16 e da *framework* Xtext, tendo sido realizada uma proposta de calendarização com os prazos a seguir.
- **Implementação da DSL PDS16:** Foi definida a sintaxe gramatical bem como a coloração da linguagem utilizando a *framework* Xtext [9], criando também validadores para certos aspetos da linguagem que ajudam ao utilizador informando os erros como por exemplo a validação semântica.
- **Integração com um assemblador:** Para gerar os ficheiros executáveis correspondentes aos programas desenvolvidos utilizando o *plug-in* PDS16inEclipse optou-se por utilizar o assemblador DASM. Para tal, invoca-se esta aplicação passando-lhe 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*.
- **Deploy Eclipse:** Como objetivo do trabalho, foi possível criar um *plug-in* com o software desenvolvido usando a framework Xtext para a plataforma Eclipse. Para tal foi criado um projeto do tipo *update site* para conseguirmos criar e disponibilizar o *plug-in*. Este foi disponibilizado com a criação de uma página web [13] que oferece dois meios de instalação: via *url* ou pasta zipada. Na página web também disponibilizamos um guia de instalação e os primeiros passos para começar a usar o editor de texto.
- **Deploy IntelliJ:** Optamos por não realizar este ponto, uma vez que não achamos viável de momento pois o *plug-in* tem como objetivo principal a plataforma Eclipse, e existem dependências diferentes para cada plataforma, optando assim pela correção de erros de implementação da gramática reportados pelos utilizadores.
- **Elaboração do Cartaz:** Foi elaborado o cartaz do projeto com o intuito de apresentar o projeto que esta a ser realizado. No cartaz damos uma visão geral do que se trata e apresentamos as principais características bem como as funcionalidades que já são suportadas pelo *plug-in*.

Face ao exposto, à data atual prevemos cumprir a calendarização inicialmente definida que se apresenta na Tabela 3.



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

Referências

[1] 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].

[2] "Dr Java," [Online]. Available: http://www.drjava.org/.

[3] "IDE Ecplise," [Online]. Available: http://www.eclipse.org.

[4] "Intellij, IDE," [Online]. Available: https://www.jetbrains.com/idea/.

[5] 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].

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

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

[8] 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].

[9] "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].

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

[11] L. Bettini, Implementing Domain-Specific, Packt Publishing, 2013.

[12] "MWE2 Documentation," [Online]. Available: https://eclipse.org/Xtext/documentation/306_mwe2.html. [Acedido em 10 6 2016].

[13] "PDS16inEclipse," [Online]. Available: http://tiagojvo.github.io/PDS16inEclipse/.

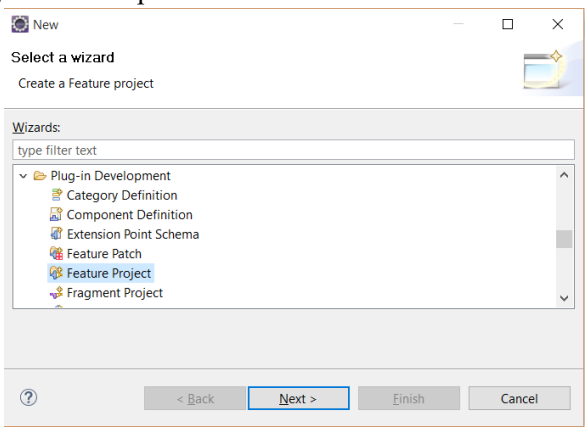
A.1 - Deploy do plug-in para o Eclipse

Após o desenvolvimento do editor de texto para a linguagem Assembly PDS16, usando a framework Xtext, decidimos publicar o software para poder ser instalado em outras máquina.

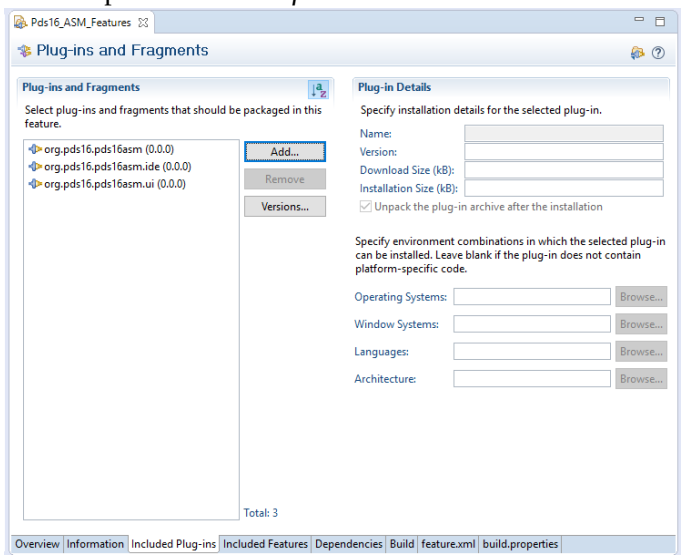
Como o software têm que ser acoplado com um IDE neste caso o Eclipse, criámos um plug-in que adicionará as novas funcionalidades ao IDE. Este não só contém o software desenvolvido como também as dependências do mesmo. No processo de *deploy* tivemos em conta o controlo de versões do plug-in, podendo este ser atualizado manualmente pelo utilizador quando for lançado uma nova versão do software.

Para a criação do plug-in efetuamos os seguintes passos:

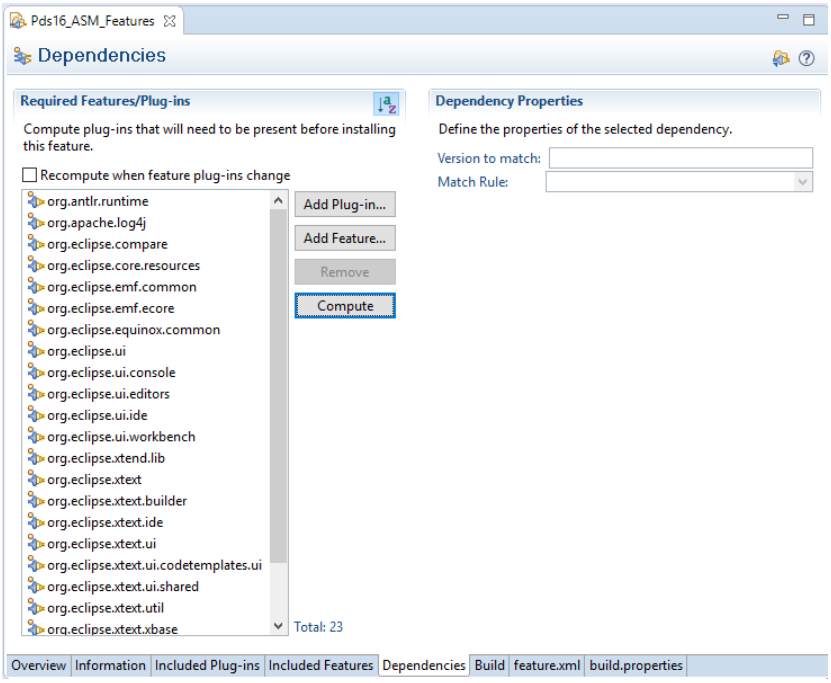
1. Criar um "Feature Project" no eclipse.



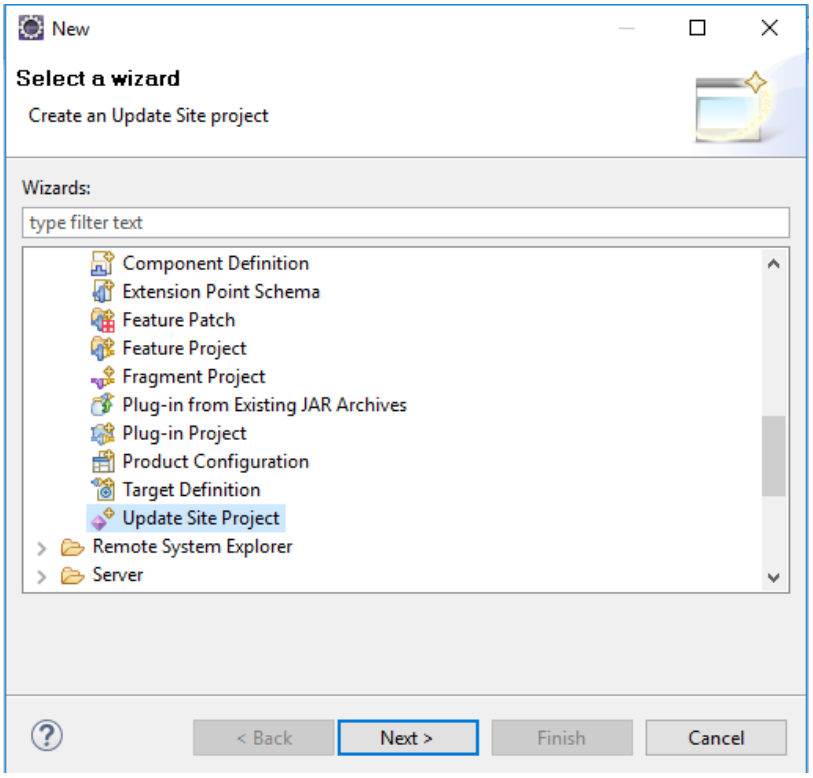
2. Abrir o ficheiro feauture.xml no projeto "*Feature*" criado anteriormente e abrir a *tab* "*plug-in*". Nessa tab clicar no botão "*Add*" e adicionar os respetivos projetos. Neste caso foram adicionados três projetos correspondentes ao *software* em desenvolvimento.



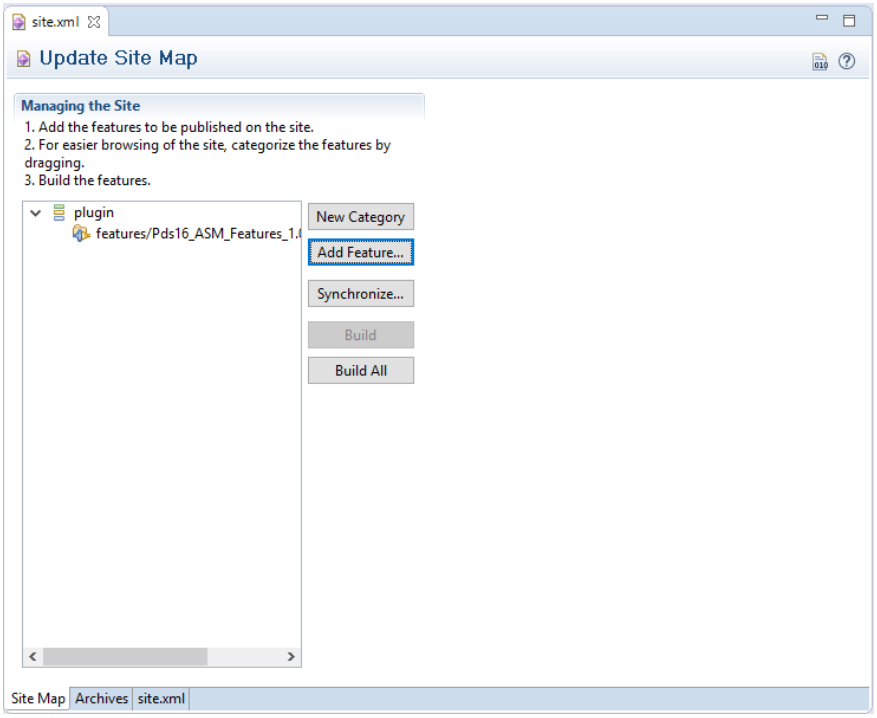
3. Na *tab* "*Dependencies*" clicar em "*Compute*" para incluir automaticamente todas as bibliotecas dos quais os projetos do passo anterior são dependentes.



4. Criar um "*Update Site Project*"



5. Neste último passo é necessário adicionar o projeto "*Feature*" criado anteriormente ao projeto "*Update Site*". Para isso abrimos o ficheiro "*site.xml*" e no *tab* "*Site Map*" clicar em "*Add Feature*" e selecionamos o projeto "*Feature*" criado. De seguida clicar no botão "*BuildAll*" para construir todos os features e plug-ins necessários para o *"Update Site"*.



Finalizados todos estes passos recorremos a uma funcionalidade do repositório Github que permite gerar um website com conteúdo desejado. Ao gerar a página automaticamente é criado um novo *branch* com o nome predefinido de "*gh-pages"*. De seguida basta fazer *push* do conteúdo do projeto "*Update Site*" criado, para esse *branch* para que seja possível instalar o plug-in no IDE Ecplise através do *link* do website alojado no *Github*.

A.2 - Instalação do Plug-in

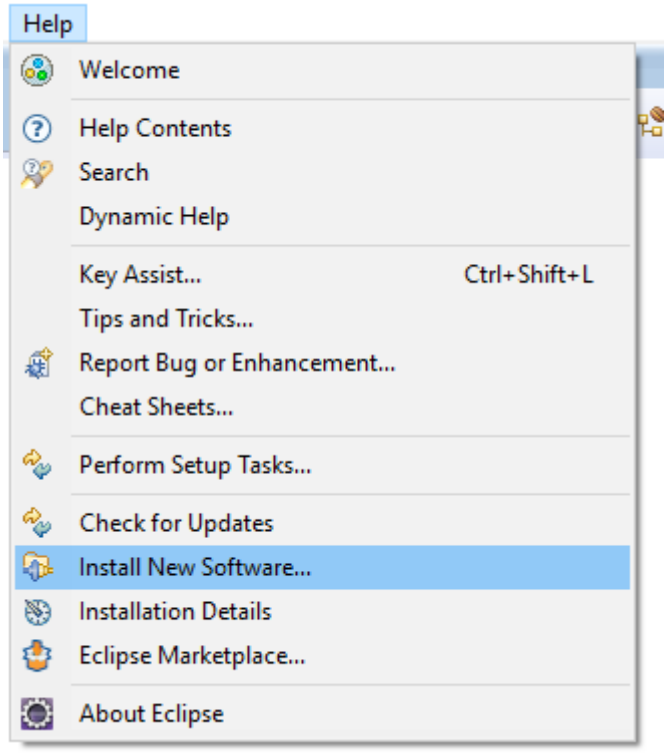
Para fazer o correto uso do editor de texto é necessário instalar o *plug-in* e definir uma variável de ambiente com a path do assemblador DASM (http://pwp.net.ipl.pt/cc.isel/ezeq/arquitetura/sistemas_didaticos/pds16/ferramentas/dasm.exe)
O plug-in pode ser instalado no IDE Ecplise de duas maneiras, fazendo download do ficheiro ZIP ou instalar usando este link: http://tiagojvo.github.io/PDS16inEclipse/.

Para a instalação do *plug-in* seja qual for a fonte é necessário seguir os seguintes passos:

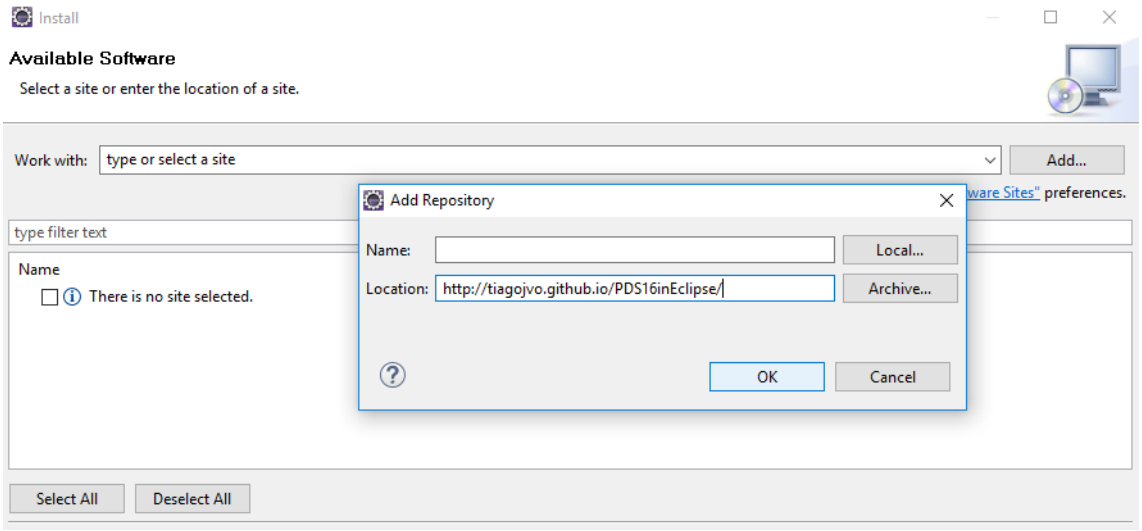
1 Definir uma variável de ambiente com o nome "DASM_PATH" com a respetiva *path* do assemblador, reiniciando de seguida o Windows para que esta fique disponível.

2 Efetuar os seguintes passos no IDE Eclipse:

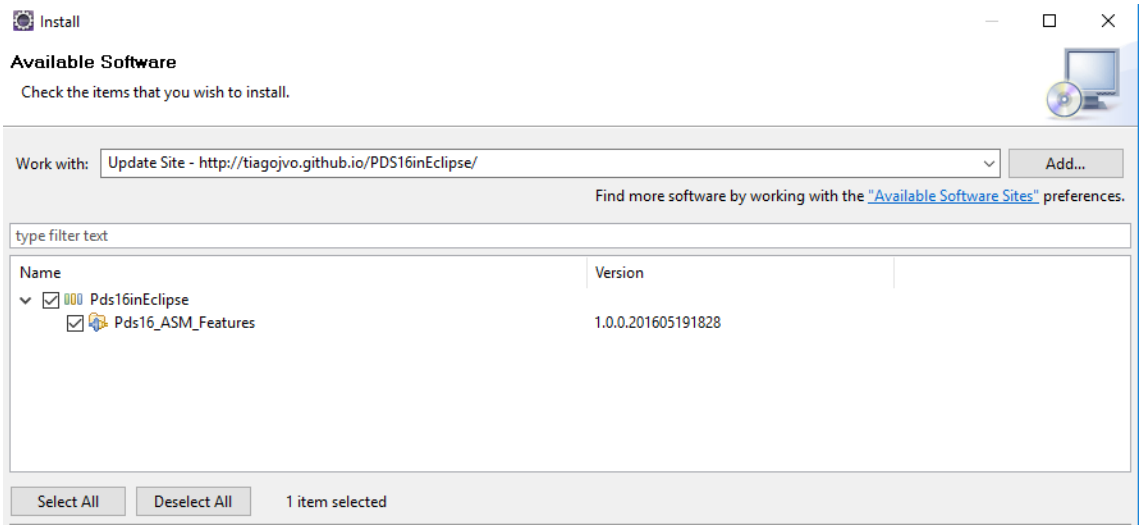
a. Clicar na tab "*Help*" -> "*Install New Software*";



b. Clicar em "*Add*" e no campo "*Location*" colocar o endereço web do plug-in ou em alternativa, descompactar a pasta "*.zip*" e selecionar o ficheiro "*contente.jar*" presente na raiz da pasta descompactada;



c. Selecionar o software "PDS16inEcplise" e prosseguir a instalação.



Utilização:

Para utilizar o plug-in basta seguir os seguintes passos no IDE Eclipse:

1 Criar um novo projecto do tipo *Java Project*;

2 No projeto criado adicionar um novo ficheiro dando-lhe a extensão "*.asm*".