PDS16inEcplise

André Ramanlal

Tiago Oliveira

Orientadores Tiago Miguel Braga da Silva Dias
Pedro Miguel Fernandes Sampaio

Relatório de progresso realizado no âmbito de Projeto e Seminário do
Curso de Licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2015/2016

Julho 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 Projeto e Seminário do
Curso de Licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2015/2016

Julho 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 diversas, como a verificação e sinalização de erros de sintaxe e de semântica, *highlighting* da sintaxe, *intellisense* e a integração com o assemblador DASM. A integração com esta ferramenta permite gerar o código máquina sem necessidade de sair do *IDE*, bem como o processamento das pseudo instruções e diretivas por ela suportadas.

O desenvolvimento da ferramenta PDS16inEclipse foi baseado na *framework* Xtext, tendo como plataforma alvo o Ambiente Integrado de Desenvolvimento (*IDE*) Eclipse. Nesta *framework* foi definida toda a linguagem *assembly* PDS16 através da criação de uma gramática própria e posteriormente dos analisadores necessários à realização das funcionalidades acima referidas.

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++.

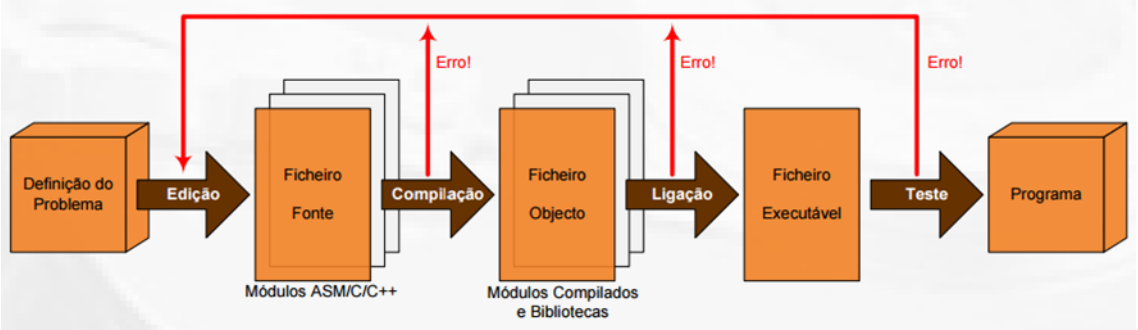


Figura 1 – Exemplo do ciclo de desenvolvimento de um programa/aplicação [1].

Após a definição do problema e a 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*, primeiramente verificando as regras sintáticas da linguagem e de seguida gerando 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, que correspondem a ficheiros partilháveis que podem conter código, dados ou recursos, em qualquer combinação. Deste último processo resulta um ficheiro com a descrição do algoritmo codificado pelos programadores em linguagem máquina localizável em memória, i.e. um ficheiro executável. Para garantir a correta implementação da solução desejada, é realizado um conjunto de testes sobre este ficheiro antes de se dar por concluído o processo de desenvolvimento do programa.

Os Ambientes Integrados de Desenvolvimento (*IDEs*) são hoje em dia aplicações que prestam 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 com

syntax highlighting, *intellisense*, geração automática de código, *refactoring*, mas também permite a integração com ferramentas externas tais como *debugger*, *linker*, compilador ou assemblador.

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 são orientadas a uma única 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 ao *IDE* de *plug-ins* ou *add-ons*[1] específicos para uma dada linguagem de programação.

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).

1.2 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", nomeadamente o da programação em *assembly*.

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*

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

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 genérico, 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 pretendeu-se implementar uma ferramenta para suportar o desenvolvimento de programas para sistemas baseados na arquitetura PDS16 usando a sua linguagem *assembly*. Esta ferramenta é essencialmente um *plug-in* para a plataforma Eclipse que oferece um editor de texto customizado e integra as seguintes funcionalidades:

- Verificação da sintaxe e da semântica 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;
- *Intellisense*, ou *auto-complete*, de modo a que o programador, intuitivamente, através de sugestões dadas pelo editor, consiga rapidamente escrever as instruções pretendidas sem a necessidade de consultar a definição das mesmas;
- *Syntax highlighting*, para permitir uma melhor legibilidade do código fonte;
- *Outline*, para assinalar numa janela os pontos importantes do código, tais como símbolos e algumas diretivas, para que o programador consiga navegar rapidamente entre essas zonas de código;
- Integração com um *assembler*, para permitir a assemblagem dos programas sem necessidade de ter que abandonar o *IDE* e visualizar no editor de texto os eventuais erros detetados neste processo.

A ferramenta desenvolvida é baseada na plataforma Eclipse, devido 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, bem como pelo 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 terem estado a utilizar esta plataforma aquando da frequência das unidades curriculares de programação dos primeiros semestres.

O desenvolvimento desta ferramenta foi conseguido recorrendo à *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 *plug-ins* para outras plataformas.

Assim, partindo como base deste nosso trabalho, será possível criar *plug-ins* para a plataforma IntelliJ e para os vários *browsers* como Google Chrome, Mozilla Firefox ou Microsoft Internet Explorer.

1.4 Estrutura do documento

Este documento encontra-se dividido em 4 (quatro) capítulos:

- Capítulo 1 – é feito o enquadramento do trabalho, em que âmbito se insere, e os objetivos definidos;
- Capítulo 2 – é apresentada uma visão pormenorizada sobre a arquitetura PDS16, a sua linguagem específica de domínio, bem como uma visão geral acerca do assemblador DASM;
- Capítulo 3 – é dada uma visão geral sobre a *framework* Xtext e é explicado, com base em exemplos concretos deste projeto, o processo de criação do *plug-in* PDS16inEclipse, incluindo as suas ferramentas e funcionalidades;
- Capítulo 4 – é feito um resumo do que poderá ser melhorado no futuro e são apresentadas sugestões para a continuação do desenvolvimento da ferramenta.

2 Arquitetura PDS16

A arquitetura PDS16 [6] consiste numa arquitetura a 16 *bits* baseada no modelo de *Von-Neumann* que adota a mesma filosofia das máquinas do tipo *Reduced Instruction Set Computer* (RISC), disponibilizando o seu modelo de programação dois bancos de 8 registos de 16 *bits* e cerca de 40 instruções distintas. O espaço de memória útil, que é partilhado não só para o armazenamento do código e dos dados dos programas mas também para a interação com periféricos, é endereçável ao *byte* e tem uma dimensão total de 64 kB. A arquitetura PDS16 inclui ainda mecanismos para suportar o atendimento e o processamento de pedidos de interrupção externos.

Nas secções seguintes apresentam-se, de forma sucinta, as principais características do modelo de programação da arquitetura PDS16. Aborda-se ainda o assemblador DASM, com enfase no seu modo de funcionamento.

2.1 Registos

A arquitetura PDS16 inclui dois bancos de registos, ilustrados na Figura 2, que visam suportar, de uma forma eficiente, o funcionamento nos seus dois modos de operação: o modo normal e o modo de interrupção.

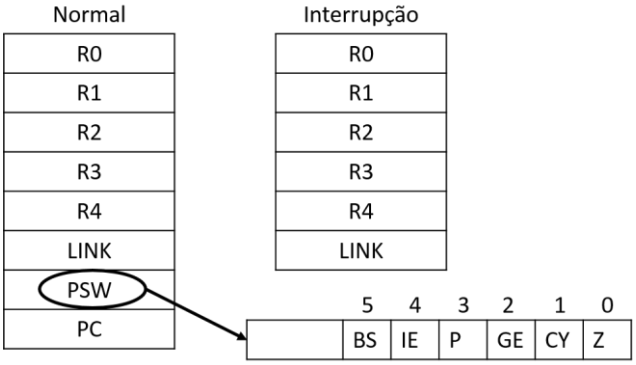


Figura 2 - Bancos de Registos PDS16.

O banco de registos acessível no modo normal disponibiliza ao programador 8 registos de 16 *bits*, denominados de `R0` a `R7`. Os registos de `R0` até ao `R4`, inclusive, são registos de uso geral que podem ser utilizados para guardar os valores das variáveis dos programas, passar parâmetros a rotinas, receber os valores devolvidos por elas, bem como para endereçar à memória e auxiliar na realização de cálculos intermédios, entre outras funcionalidades

O registo `R5` também pode ser utilizado como registo de uso geral, mas está intrinsecamente comprometido com a utilização de rotinas. Na verdade, este registo é usado implicitamente pela instrução `JMPL` para salvaguardar o valor corrente do *Program Counter* (`PC`) aquando da invocação de uma rotina, de modo a ser possível recuperar o fio de execução do

programa após a sua conclusão. Por este motivo, este registo também é denominado de *Link Register*.

Os registos `R6` e `R7` são os outros dois registos especiais do processador. O registo `R6`, não obstante também poder ser usado como operando fonte ou operando destino na realização de instruções, guarda os indicadores de erro e relacionais produzidos pela ALU (`Z`, `CY`, `GE` e `P`), bem como os parâmetros relativos ao modo de funcionamento do sistema (`IE` e `BS`), sendo por este motivo também denominado de *Processor Staus Word* (`PSW`). O significado destas *flags*, cujo posicionamento nos 16 *bits* que compõe o registo é ilustrado na Figura 3, é o seguinte:

- Z (Zero): Caso apresente o valor lógico 1, significa que o resultado da última operação realizada na ALU e que atualizou as *flags* produziu o valor zero.
- CY (*Carry/Borrow*): Esta *flag* apresenta o valor lógico 1 quando a última operação realizada na ALU e que atualizou as *flags* produziu um *carry/borrow out*.
- GE (*Greater or Equal*): Esta *flag* apresenta o valor lógico 1 quando, ao realizar-se uma subtração que atualize as *flags*, o diminuendo é maior ou igual ao diminuidor, considerando que os operandos da ALU pertencem ao conjunto dos números relativos.
- P (*Parity*): Esta *flag* fica ativa sempre que o valor produzido pela última operação realizada na ALU e que atualizou as *flags* contenha um número de *bits* com valor lógico 1 em quantidade ímpar.
- IE (*Interrupt Enable*): Quando esta *flag* toma o valor lógico 1 o mecanismo de interrupção está ativo, podendo a normal execução de um programa ser interrompida por uma ação externa.
- BS (*Bank Select*): Esta *flag* serve de seletor de banco de registos, ou seja, o banco de registos do modo normal está em utilização quando o seu valor é 0, enquanto se o seu valor for 1 é o banco de registos do modo interrupção que está em utilização.

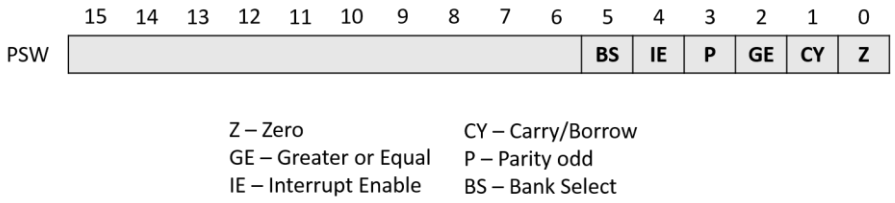


Figura 3 – Estrutura interna do registo `PSW`.

Finalmente, o registo `R7` corresponde ao *Program Counter* (`PC`), guardando o endereço de memória da próxima instrução a ser executada.

O banco de registos do modo interrupção destina-se a suportar um processamento mais rápido dos pedidos de interrupção externa. Para tal, este banco de registos disponibiliza ao programador seis novos registos de uso geral, em substituição dos registos `R0` a `R5` existentes no banco de registos do modo normal, conforme é ilustrado na Figura 2. Desta forma torna-se

possível, para a maior parte das situações, desenvolver rotinas de processamento dos pedidos de interrupção que evitem a necessidade de salvaguardar em memória o conteúdo dos registos por elas modificados e, com isso, minimizar o seu tempo de execução.

Importa ainda referir o comportamento especial dos registos `R0` e `R5` deste banco de registos, que efetivamente suportam o mecanismo de interrupção da arquitetura. O registo `R5` é utilizado para guardar o endereço de memória que deverá ser utilizado para fazer o retorno do modo de interrupção, sendo iniciado com este valor do `PC` no momento da entrada neste modo de funcionamento. Já o registo `R0` é utilizado para salvaguardar o valor do registo `PSW` durante a execução do troço de código responsável pelo processamento do pedido de interrupção, valor que é reposto automaticamente aquando da saída do modo de interrupção.

2.2 Conjunto de instruções

O conjunto de instruções disponibilizado pela arquitetura PDS16 compreende 40 instruções distintas, todas codificadas com um tamanho fixo de 16 *bits* e localizáveis em memória em endereços múltiplos de 2 *bytes*. Conforme pode ser visto na Tabela 1, estas instruções estão organizadas, fundamentalmente, em três grandes classes: processamento de dados, transferência de dados e controlo do fluxo de execução, incluindo suporte a rotinas.

Operação		Assembly	Ação
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 assembly PDS16.

Palavras-chave	Descrição
rd	Registo destino
rs	Registo fonte
rbx	Registo base
rix	Registo de indexação (conteúdo é multiplicado por dois para acesso à palavra)
Rm/rn	Registos que contêm os operandos
immediate8	Constante de 8 bits sem sinal
direct7	Constante de 7 bits sem sinal e que corresponde aos endereços dos primeiros 128 bytes ou 64 words da memória
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
Rbx	Registo base
F	Sufixo que quando colocado à direita da mnemónica indica que o registo PSW não é atualizado
Sin	Valor lógico do bit a ser inserido à esquerda ou à direita.

Tabela 2 - Palavras-chave da sintaxe PDS16.

2.2.1 Processamento de dados

Estas instruções têm como objetivo o processamento dos dados através da realização 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, esses parâmetros correspondem a um dos 8 registos do processador. Contudo, em algumas instruções (i.e. `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 tem sempre como destino um dos registos do banco de registos do processador.

Por definição, todas as instruções de processamento de dados também afetam o registo de estado do processador (`PSW`), atualizando o valor dos *bits* relativos aos indicadores relacionais e de excesso de domínio produzidos pela ALU (ver Tabela 1). Não obstante, para algumas destas instruções, pode adicionar-se o sufixo "`f`" à mnemónica da instrução para indicar que o registo `PSW` não deverá ser afetado na sequência da sua execução. Nestas situações, 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.

Para além das instruções já mencionadas, existem duas outras para fazer o carregamento de constantes nos registos do processador, i.e. `LDI` e `LDIH`. A instrução `LDI` permite carregar uma constante, codificada em código binário natural com 8 *bits*, num registo. Por sua vez, a instrução `LDIH` suporta o carregamento de constantes codificadas em código binário (natural e dos complementos) com 16 *bits* nos registos do processador. Para tal, esta instrução apenas afeta a parte alta (*bits* 8 a 15) do registo alvo, mantendo inalterado o valor da parte baixa (*bits* 0 a 7) desse registo. Para ilustrar esta operação, apresenta-se de seguida um troço de código *assembly* que carrega a constante -1 para o registo `R0`:

```
LDI  R0, #0xFF
LDIH R0, #0xFF
```

2.2.2 Transferência de dados

As operações de transferência de dados, são responsáveis pela troca de dados entre o subsistema de memória e o banco de registos, uma vez que as operações de processamento de dados não usam operandos em memória. Estas operações podem ser efetuadas a 16 *bits* (*word*), ou a 8 *bits* (*byte*). A operação `LDB` transfere da memória um *byte* para o registo destino, com a particularidade de implicitamente fazer a extensão para 16 *bits*, sem sinal, do *byte* transferido da memória. 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, caso se pretenda o acesso ao *byte* ao invés da palavra, deverá acrescentar-se o sufixo "`B`" à mnemónica da instrução (ver Tabela 1).

As operações de transferência de dados entre o subsistema de memória e o banco de registos podem ser realizadas usando dois modos de endereçamento distintos: o direto e o baseado indexado.

No modo de endereçamento direto, a posição de memória a aceder para realizar uma operação de leitura ou de escrita de dados é especificada usando apenas uma constante, codificada na própria instrução em código binário natural com 7 *bits*. Para aumentar a eficiência da codificação, o valor desta constante é determinado tendo em conta o tipo de dados que a instrução manipula, i.e. uma palavra ou um *byte*. Logo, para as instruções `LD` e `ST` a constante permite acesso direto às primeiras 128 palavras do espaço de memória (endereços `0x0` a `0xFE`), enquanto nas instruções `LDB` e `STB` assegura acesso direto apenas aos primeiros 128 *bytes* (endereços `0x0` a `0x7F`). Isto é conseguido ao nível da micro arquitetura do processador, onde, para as instruções `LD` e `ST`, o valor da constante é multiplicado por 2 antes de ser colocado no barramento de endereço.

Por outro lado, o endereço da posição de memória a aceder no modo de endereçamento baseado indexado é definido à custa de dois parâmetros: um valor base e um índice. Independentemente da instrução considerada, o valor da base é sempre obtido do banco de registos, enquanto o valor do índice pode ser obtido também de um desses registos ou definido usando uma constante codificada em código binário natural com 3 *bits* (ver Tabela 1).

Pelas razões anteriormente apresentadas, aquando da execução das instruções `LD` e `ST` o valor do índice é automaticamente multiplicado por dois na micro arquitetura antes de ser colocado no barramento de endereço do processador.

2.2.3 Controlo do fluxo de execução

Para controlar o fluxo de execução dos programas, a arquitetura PDS16 disponibiliza ao programador uma instrução de salto incondicional e quatro instruções de salto condicional, as quais avaliam o valor das *flags Zero* e *Carry* nas formas direta e complementar (ver Tabela 1).

Independentemente da instrução considerada, o modo de endereçamento subjacente é sempre o mesmo: endereçamento baseado indexado tomando, implicitamente, o `PC` como registo destino. O valor da base pode ser obtido de qualquer um dos 8 registos do processador, enquanto o valor do índice consiste numa constante, codificada em código dos complementos com 7 *bits*. Para melhorar a eficiência da codificação, o índice é multiplicado por dois antes de ser somado ao valor obtido do registo base, já que o resultado desta operação terá que corresponder sempre a um número par (note-se que as instruções são codificadas com 16 *bits*, ocupando duas posições de memória consecutivas).

A arquitetura PDS16 também oferece uma instrução de salto incondicional com ligação (`JMPL`) para dar suporte à implementação de rotinas. A sintaxe desta instrução é idêntica à anteriormente descrita (ver Tabela 1), pelo que apenas se distingue da instrução `JMP` pelo facto de, para além de atualizar o `PC` com o valor do endereço de memória correspondente ao salto, também atualizar o registo `R5` (`LR`) com o valor atual do `PC`, isto é, o endereço da posição de memória subsequente à da instrução `JMPL`. Estas duas operações acontecem em simultâneo, sendo portanto indivisíveis no tempo.

2.3 Subsistema de memória

A arquitetura PDS16 implementa o modelo desenvolvido por *John Von Neumann*, pelo que o seu subsistema de memória deve ser visto como um espaço de memória único que é partilhado para o armazenamento do código e dos dados dos programas, bem como para a interação com periféricos. Este espaço de memória, que respeita o formato de organização *big-endian*, tem uma dimensão total de 64 kB e pode ser endereçado em ordem a uma palavra de 16 *bits* ou a um *byte*. Independentemente do número de *bytes* de dados transferidos por uma dada instrução, por uma questão de eficiência, o acesso ao subsistema de memória é sempre realizado a 16 *bits*. Por este motivo, todas as palavras, correspondam elas a instruções ou a dados, têm que estar localizadas em memória em endereços múltiplos de dois *bytes* (i.e. alinhadas em ordem à palavra).

2.4 Exceções

Um mecanismo de exceção visa o tratamento de eventos inesperados, síncronos ou assíncronos, que ocorrem durante a execução de um programa e que têm impacto, direta ou indiretamente, nessa mesma execução. Neste caso, a arquitetura PDS16 suporta dois mecanismos de exceção: *Hard Reset* e Interrupção.

O mecanismo *Hard Reset* corresponde à ativação de um sinal diretamente proveniente de uma entrada externa do sistema [10]. Como o próprio nome indica, este mecanismo leva a que a unidade de controlo volte ao seu estado inicial, interrompendo a execução do programa em curso. Para tal, a implementação deste mecanismo consiste em forçar o carregamento do valor 0, simultaneamente, no registo `PC`, levando a que a execução do programa recomece novamente a partir da primeira posição de memória (*boot*), e no registo `PSW`, originando a seleção do banco de registos do modo normal (*flag* `BS` com o valor 0) e bloqueando o atendimento de interrupções externas (*flag* `IE` com o valor 0).

O mecanismo de interrupção permite notificar o sistema da ocorrência de eventos externos, síncronos ou assíncronos, e que precisam de ser processados. Esta notificação é feita ativando um sinal também proveniente de uma entrada externa do sistema e que é sensível a nível lógico 0 [10]. Contudo, para que esse pedido de interrupção possa ser atendido e processado pelo sistema é necessário que no registo `PSW` a *flag* `IE` também se encontre ativa, i.e. tome o valor lógico 1. Nestas condições, assim que a instrução que está a ser realizada termina a sua execução, o sistema comuta para o modo de interrupção, o que compreende a realização, em simultâneo, das seguintes operações [11]:

- Copiar o valor do registo `PSW` para o registo `R0` do banco de registos do modo interrupção;
- Afetar a *flag* `IE` do registo `PSW` com o valor lógico 0, inibindo assim o atendimento de novos pedidos de interrupção;
- Afetar a *flag* `BS` do registo `PSW` com o valor lógico 1, tornando desta torna ativo o banco de interrupção;
- Copiar o valor do registo `PC` para o registo `LR` do banco de registos do modo de interrupção;
- Colocar o valor `0x2` no registo `PC`, vetorizando desta forma o processamento para o ponto de entrada da interrupção.

O retorno ao modo normal é feito recorrendo à instrução IRET, que garante a indivisibilidade entre as várias operações subjacentes à realização desta operação. Esta instrução faz a cópia, em simultâneo, do valor dos registos `LR` e `R0` do banco de interrupção para os registos `PC` e `PSW`, respetivamente. Considerando que o registo `R0` do banco de interrupção mantém o valor copiado do registo `PSW` no momento do atendimento do pedido de interrupção, então estas operações permitem repor o estado do programa interrompido, incluindo a permissão para atendimento de interrupções, sendo também restabelecida a utilização do banco de registos normal para a execução do programa.

2.5 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 a arquitetura PDS16, existe um assemblador, uni modelar, denominado DASM [7] que a partir de um ficheiro de texto escrito em linguagem assembly PDS16 produz o ficheiro com a descrição correspondente em linguagem máquina, i.e. o ficheiro executável do programa.

Pelo facto do DASM ser um assemblador uni modular com objetivos de utilização didáticos, esta ferramenta não só não suporta o desenvolvimento de aplicações usando múltiplos ficheiros fontes como ainda inclui diretivas específicas para fazer, de forma estática, a localização em memória das instruções, variáveis e constantes definidas nos ficheiros fonte dos programas. Por tudo isto, não existe a necessidade de uma ferramenta de ligação e localização.

O ficheiro executável gerado pelo DASM tem a extensão HEX e adota o formato Intel HEX80 [12]. 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.

A execução do programa DASM também produz um ficheiro com extensão LST. Este é destinado a ser impresso ou consultado pelo utilizador, pois contém diversas informações de auxílio, tal como o texto original adicionado do código de cada instrução e respetivo endereço de localização em memória. Caso existam erros de compilação, os mesmos são assinalados para a respetiva instrução, com uma mensagem identificadora do seu tipo e da possível causa.

2.5.1 Escrita de programas

Quando um programador escreve o seu programa num ficheiro fonte deve ter em conta que o assemblador DASM lê o ficheiro segundo a ordem *top down*, e que 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 de 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:** Trata-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 assemblador ignora os seus caracteres. Existem dois 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.

2.5.2 Diretivas

Para além das instruções assembly PDS16, o assemblador DASM reconhece e processa um outro conjunto de comandos [13]. 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. "`.TEXT`" – que aloja as instruções do programa;
2. "`.DATA`" – que aloja as variáveis globais com valor inicial;
3. "`.BSS`" – que aloja as variáveis globais sem valor inicial.

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 pode 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:

1. "`.WORD`" – define uma/várias palavra/s em memória;
2. "`.BYTE`" – define um/vários *byte/s* em memória;
3. "`.ASCII`", "`.ASCIIZ`" – definem 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 inicializados com um valor definido pelo programador.

Existe também a possibilidade de serem atribuídos valores a símbolos através das diretivas "`.EQU`" e "`.SET`", sendo que na primeira o valor é atribuído de forma permanente e no segundo de forma temporária.

3 *Framework* Xtext

Xtext é uma *framework* desenvolvida com base na linguagem de programação Java que é utilizada principalmente para o desenvolvimento de linguagens de programação e de linguagens de domínio específico, as denominadas DSL. Uma grande vantagem da *framework* Xtext é a sua integração com a Eclipse Modeling Framework (EMF) [14], o que permite a conversão de código escrito usando uma dada linguagem para um modelo que, posteriormente, pode ser transformado num outro modelo ou serializado para uma outra linguagem [15]. O motivo pelo qual é necessário associar este modelo ao código resulta da necessidade de, do ponto de vista da implementação, existir uma estrutura *meta-data* à qual referir aquando da descrição das regras da linguagem.

A *framework* Xtext possibilita ainda, mediante a definição de toda a sintaxe e a semântica de uma linguagem, construir de uma forma quase automática uma infraestrutura que poderá incluir *parser*, *linker*, *typechecker* e compilador, bem como suporte à edição de texto escrito usando essa linguagem [16]. Por exemplo, a *framework* possibilita a criação de um editor baseado em plataformas como os ambientes de desenvolvimento Eclipse ou IntelliJ IDEA, bem como através de um *browser*.

Esta secção do relatório está organizada segundo o processo adoptado no desenvolvimento do *plug-in*, ou seja, na secção 3.1 é descrita, de um modo geral, a arquitetura de um projeto baseado na *framework* Xtext para auxiliar o leitor a ter uma melhor perceção do problema e da própria *framework*. Na secção 3.2 apresenta-se a técnica de descrição de uma linguagem através do ponto crucial de um projeto deste tipo, a gramática, e é ainda abordada a implementação de validadores da linguagem. Por fim, a secção 3.3 trata da integração da ferramenta numa plataforma, neste caso a plataforma Eclipse, passando pela descrição de algumas funcionalidades adicionadas, onde a implementação é específica para a plataforma considerada neste projeto, e o modo de geração do *plug-in*.

3.1 Arquitetura

A *framework* Xtext oferece ao programador a oportunidade de descrever diferentes aspetos relacionados com a sua linguagem de programação, como por exemplo o *highlighting*, validação e *parsing*. Estes podem ser implementados em Java ou numa linguagem específica criada à base de Java, denominada Xtend [17]. Esta linguagem está totalmente integrada com a linguagem Java, obtendo assim todos os recursos e suporte que o Java disponibiliza, tais como as bibliotecas, e também outras funcionalidades como o *type inference*, métodos de extensão, expressões lambda e *multi-line template expressions*.

Sendo assim, aquando da criação de um projeto Xtext, e independentemente da linguagem de programação adotada (i.e. Java ou Xtend), este encontrar-se-á divido em vários *packages* com diferentes responsabilidades, sendo que os mais importantes são:

- org.example.dslDomain – contém a definição da gramática da linguagem considerada e todos os seus componentes de *Runtime* (*parser*, *validator*, etc.);
- org.example.dslDomain.Ui (na integração com o Eclipse) – *package* de integração com a plataforma de desenvolvimento que contém implementações relacionadas com a interface visual (*highlighting*, *outline*, etc).

Cada um destes *packages* contém um módulo responsável por indicar que componentes deverão ser utilizadas para cada tarefa. Por exemplo, neste projeto, em que a linguagem considerada foi denominada de "*Pds16asm*", existem os módulos "*Pds16asmRuntimeModule*" e "*Pds16asmUiModule*", sendo que cada um deles contém métodos para a associação de cada um dos componentes acima mencionados. Estes módulos estendem de módulos abstratos que associam componentes por definição, sendo que no caso de necessidade de alterar um deles, basta redefinir o método por ele responsável fazendo-o retornar o componente pretendido (Figura 4) [18].

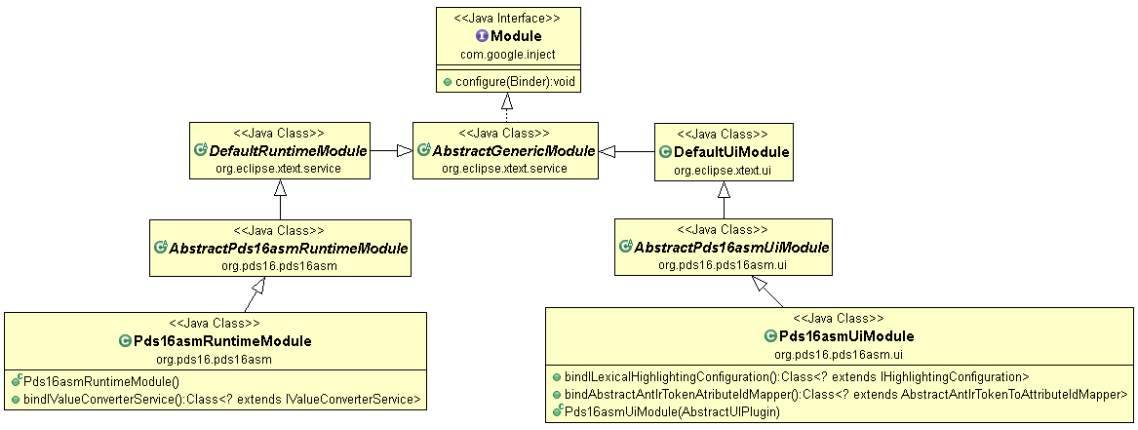


Figura 4 - Diagrama de classes referente à organização de Módulos.

Dado este modo de desenvolvimento, constata-se que o ponto principal de um projeto baseado na *framework* Xtext é a sua gramática (descrita num ficheiro com extensão *".xtext"*), pois aí é definida toda a sintaxe da linguagem através da qual será gerado um conjunto de classes que formam o modelo domínio da linguagem, necessário para a implementação das funcionalidades requeridas. Esta geração é efetuada pelo *Modeling Workflow Engine* (MWE2) [19].

3.1.1 Modeling Workflow Engine (MWE2)

O MWE2 é baseado no modelo *Plain Old Java Object* (POJO [20]), sendo responsável pela inicialização de todas ações para a geração de um *plug-in*, onde é possível declarar instâncias de objetos e atributos de valor e de referência [15]. Através da configuração do MW2 é possível definir a forma como o código é gerado, como por exemplo definir se a linguagem é ou não *case-sensitive*.

O ficheiro de configuração contém uma componente denominada de *Generator* que sendo o *entry point* para a geração do *plug-in* da linguagem está acessível ao programador através da classe *XtextGenerator* (Figura 5).

```
module org.pds16.GeneratePds16asm

import org.eclipse.xtext.xtext.generator.*
import org.eclipse.xtext.xtext.generator.model.project.*

var rootPath = ".."

Workflow {

    component = XtextGenerator {
        configuration = {
            project = StandardProjectConfig {
            code = {
        }
        language = StandardLanguage {
            name = "org.pds16.Pds16asm"
            fileExtensions = "asm"

            serializer = {
            validator = {
            parserGenerator = {
            scopeProvider = {
        }
    }
}
```

Figura 5 - Excerto do ficheiro de configuração GeneratePds16asm.mwe2.

Este tipo de componente é constituído por fragmentos que são representados por classes que têm acesso a alguns recursos disponibilizados pelo componente *Generator*, como a gramática da linguagem e o mecanismo para a geração do código [15]. Para a geração do código são utilizadas duas gramáticas ANTLR [21], geradas pela Xtext: uma para produção do "*parser*", de onde resulta uma *Abstract Syntax Tree* [22] (AST), e outra que é utilizada para o processamento

dos eventos do editor do *Eclipse* [23]. A sintaxe do ficheiro de gramática Xtext é igual à do ANTLR, à exceção de que, para a verificação semântica, o ANTLR contém código Java embebido na descrição da gramática, enquanto a Xtext faz uso de injeção de dependências para referir que tipos devem ser gerados para essa mesma verificação [24]. Estas dependências são resolvidas utilizando a *framework* Google Guice [25], que dá suporte à injeção de dependências usando anotações para configurar objetos em Java. Este tipo de dependências é um padrão de desenho usado para remover dependências *hard-coded* resultando assim classes com fraca dependência entre elas [15].

3.2 Gramática

Com o estudo das instruções da arquitetura PDS16 e das diretivas (e mnemónicas) da ferramenta DASM [13] [6] [7], foi possível definir uma gramática para esta linguagem, tendo em conta as possíveis formas de escrever as suas instruções e comandos.

Apesar de ser possível converter uma instrução específica num dado código de modelo, é desejável fazer-se essa conversão recorrendo a um *template* representativo, de modo a obter-se um modelo final coerente para as várias instruções. No caso de um projeto baseado na *framework* Xtext, a gramática toma o papel de *template* de código. Assim, a gramática é definida através de regras que podem referenciar outras regras ou palavras-chave. Por cada regra definida é gerada uma classe modelo com, dependendo da definição da regra, métodos e/ou atributos. Na geração das classes modelo, será adicionada a dependência entre elas, como por exemplo nas seguintes regras apresentadas na Figura 6:

```
Directive:
    value =(Bss | Data | End | Text | Equ | Org | Section | Set | LabelDirective);

LabelDirective: Ascii | Asciiz | Byte | Word | Space;

Space:  tag='.space' size=Number (',' byteValue=Number)?;
```

Figura 6 – Excerto de código de uma gramática Xtext.

Nesta figura podemos apontar que a regra "*Space*" depende de "*LabelDirective*", o que irá traduzir-se numa dependência entre elas. Essa dependência é tratada pela Xtext gerando automaticamente classes em Java quando o MWE2 é executado, resolvendo essa dependência pela extensão entre classes, criando assim uma hierarquia entre as regras de uma DSL (Figura 7). Por outro lado, é de verificar que a regra "*LabelDirective*" não irá depender da regra "*Directive*", pois o seu valor está a ser guardado na propriedade "*value*", sendo que este poderá tomar vários valores diferentes.

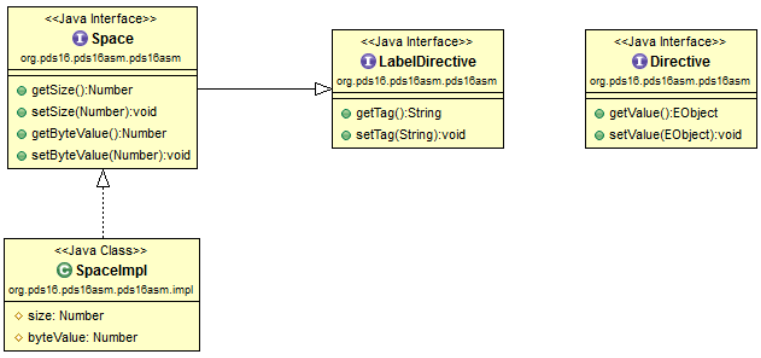


Figura 7 - Classes geradas pela framework.

Na Figura 7 pode-se ainda apontar que, como existem propriedades na regra às quais são atribuídos valores, na interface gerada irão existir *getters* e *setters* referentes às mesmas, como é o caso da interface "*Space*". Na sua implementação ("*SpaceImpl*"), estas propriedades traduzir-se-ão em campos, acedidos através dos *getters* e *setters* implementados de acordo com a declaração em "*Space*".

3.2.1 Regras da gramática

O corpo de um ficheiro de gramática Xtext é composto essencialmente por uma sequência de regras, definidas pelo programador ao implementar uma dada DSL.

Antes de passar para a definição de regras, é necessário que o programador tenha o conhecimento dos elementos de sintaxe disponíveis para a definição das mesmas. Estes elementos consistem em operadores que auxiliam na construção das regras, de modo a conseguir-se implementar definições específicas para cada uma delas. A Tabela 3 apresenta alguns exemplos destes elementos.

Elemento da Sintaxe	Significado
default (sem operador)	Exatamente um
? sufixo	Um ou nenhum
*** sufixo**	Zero ou mais
+ sufixo	Um ou mais
| infixo	Or
& infixo	And
.	Caracter universal
('0'..'9')	Range

Tabela 3 - Elementos da sintaxe gramatical Xtext.

Dadas estas definições de elementos, pode-se fazer a definição das regras de gramática necessárias, que podem ser de dois tipo Xtext: *Parser Rules* ou *Terminal Rules*.

Parser Rules são regras não terminais, ou seja que definem uma sequência de outras regras conjugadas com palavras-chaves, não definindo um *token*[2] mas sim uma árvore de regras terminais e não terminais, denominadas de *parse tree* ou *node model*, respetivamente.Assim, uma regra contém em primeiro lugar o seu nome, que deve ser único na gramática, seguido do caracter ':' e da definição da mesma. No código da Figura 8 são apresentadas algumas *parser rules* definidas para o projeto.

```
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 8 - Código exemplo da definição das regras.

Tendo como exemplo a nossa implementação da gramática, "*Statement*" é uma regra que na sua definição apenas contém referências para outras regras, enquanto "*Ret*" e "*Nop"* são apenas constituídas por palavras-chave, não dependendo portanto de nenhuma outra regra.

Por outro lado, na regra "*Label"* podemos verificar que a sua definição compreende uma palavra-chave (':'), uma propriedade "*labelName*" (que contém um valor do tipo *ID*, considerado um terminal) e uma referência para outra regra.

A regra "*Jump*" é um pouco diferente das anteriores, pois apesar de o seu primeiro elemento ser uma palavra-chave, o segundo elemento pode tomar diferentes tipos de valor:

- Referência para uma *parser rule* – "*OperationWithOffset*";
- Referência para uma *terminal rule* – "*ID*";
- Palavra-chave – '$'.

As *Terminal Rules*, também denominadas de *token rule*s ou *lexer rules*, são regras representadas por *tokens*, usualmente definidos por expressões regulares. Estas regras são definidas pela palavra-chave "*terminal*", seguida do nome da regra (por convenção, escrito em

[2]Sequência de caracteres.

letras maiúsculas), do caracter ':' e da expressão regular que a define. A Figura 9 contém a definição de duas regras terminais implementadas no projeto.

```
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 9 - Código exemplo da definição regras terminais.

O primeiro terminal, "*ID*", começa com um caracter na gama de 'a' a 'z' (maiúsculo ou minúsculo) ou, em alternativa, pelo caracter '_', seguido de zero ou mais caracteres (devido ao elemento de sintaxe '*'), que podem ser algarismos.

Um terminal retorna sempre um tipo que, por definição, se trata de uma *String*, sendo possível manipulá-lo para o tipo específico pretendido. O terminal "*HEX*" trata a definição de números em código hexadecimal, mas ao contrário do terminal "*ID*" retorna um número inteiro em vez de *String*.

Para alterar o tipo de retorno de uma *terminal* rule é necessário indicar na sua definição o tipo que se pretende retornar, o que se consegue recorrendo à utilização da palavra-chave "*returns*" seguida do tipo pretendido. Após essa indicação é necessário definir um modo de fazer a tradução de *string* para o novo tipo de retorno. Para isso há que redefinir o método "*bindIValueConverter*" na classe que representa o *RunTimeModule* do projeto em questão, neste caso "*Pds16RunTimeModule*", Figura 10. 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 10 - Código da classe Pds16asmRuntimeModule.

A classe "*Pds16asmValueConverter*" implementa a interface "*IValeuConverterService*", em que, através de anotação de métodos para injeção de dependências (como descrito nas secções anteriores), são definidas as regras para a conversão do tipo de retorno e qual a classe responsável por esta operação, Figura 11.

```
class Pds16asmValueConverter extends DefaultTerminalConverters implements IValueConverterService{

    @Inject
    private HEXValueConverter hexValueConverter

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

Figura 11 - Excerto da classe PDS16asmValueConcerter.

Conforme ilustrado na Figura 11, a anotação "*@ValueConverter(rule="HEX")*" indica que o método por ela anotado retornará, para a regra com o nome "*HEX*", um conversor do seu tipo de retorno. Neste caso, trata-se de um conversor para *Integer*, sendo que este é uma instância da classe "*HEXValueConverter*", que por sua vez terá de implementar a interface "*IValueConverter*" (Figura 12).

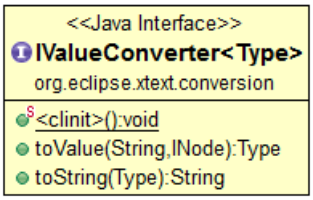


Figura 12 - Interface *IValueConverter*.

3.2.2 Definição dos elementos do analisador de regras

Como em todos as linguagens, existe a necessidade de validar regras de semântica, pois o significado dos dados inseridos pelo utilizador, apesar de poderem corresponder ao tipo requerido, podem não fazer sentido num determinado contexto, e.g. a atribuição de um valor a uma variável do tipo *byte* que excede os limites dos valores possíveis para um *byte*. Estas validações não são possíveis de realizar com base, exclusivamente, nas definições de uma gramática (*par*ser *rules* e *terminal rules*), pelo que têm que ser verificadas no ato de escrita de código.

A *framework* Xtext disponibiliza um mecanismo de validação que satisfaz essa necessidade, permitindo assim analisar o conteúdo de uma regra e notificar o utilizador de eventuais erros. Estas verificações são da responsabilidade da classe responsável pela realização das validações, no nosso caso "*Pds16asmValidator*", que é gerada pela própria *framework*. O mecanismo é semelhante ao mencionado nas secções anteriores, onde é feita a injeção de dependências através de anotação de métodos. Neste caso, trata-se da anotação "*Check*" sobre um método que recebe como parâmetro uma instância da classe representativa da regra a analisar.

No caso concreto do nosso no projeto, são verificados os limites dos números presentes nas regras. Por exemplo, para uma regra que contenha um número representativo de um *offset* a 8 *bits* com sinal, será verificado se o número inserido pelo utilizador está compreendido entre os limites permitidos, emitindo-se um *warning* caso contrário (Figura 13).

```
@Check
def checkOffset8(Offset8OrLabel o){
    if(o.number == null)//in case of this instance has the value from a label
        return;
    var Integer value = o.number.value;
    if(value < MIN_8BIT_WITH_SIGNAL  || value > MAX_8BIT_WITH_SIGNAL)
        warning('Number should be between' + MIN_8BIT_WITH_SIGNAL + ' and ' + MAX_8BIT_WITH_SIGNAL,
                Pds16asmPackage.Literals.OFFSET8_OR_LABEL__NUMBER,
                "Invalid Number")
}
```

Figura 13 - Exemplo de um validador.

3.3 Integração com a plataforma Eclipse

A *framework* Xtext permite que seja utilizada a definição de uma linguagem para a integração com um *IDE* para que possa ser gerada uma ferramenta sob a forma de um *plug-in*. Para fazer uso do *plug-in* , este pode ser instalado nas plataformas suportadas, adicionando-se assim novas funcionalidades a esses *IDEs*. No nosso caso, gerámos um *plug-in* compatível com a plataforma Eclipse, permitindo assim que um programador possa desenvolver programas em *assembly* PDS16, com adição de algumas funcionalidades específicas a este *IDE*.

3.3.1 Syntax Highlight

Uma das funcionalidades implementadas no *plug-in* é o suporte para *highlighting*, de modo a ajudar o utilizador a distinguir os vários tipos que a gramática pode suportar. No âmbito deste projeto, foram considerados cinco tipos de estilos: diretivas, nome instruções, comentários, símbolos (i.e. *labels*) e texto. Cada tipo tem associado uma cor e estilo de letra específicos.

Para colorir a sintaxe da gramática, a biblioteca Xtext oferece a classe "*DefaultHighlightingConfiguration*" (uma implementação de "*IHighlightingConfiguration*"). Apesar de esta conter cores predefinidas para certos tipos, resolvemos criar a classe "*Pds16HighlithingConfiguarion*" para associar a cada tipo uma cor e um formato customizado, de modo a adotar um esquema de cores e estilos o mais semelhante possível ao empregue pelos editores de texto *assembly* clássicos (Figura 14). Aqui é redefinido o método "*configure*" que regista todos os estilos que o utilizador pretende utilizar no parâmetro recebido ("*aceptor*"), associando-os a um *id*.

```
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 14- Excerto de código de Pds16HighlightingConfiguration.

Após registar os estilos a utilizar, é ainda necessário associá-los aos *tokens* da sintaxe gramatical para que os mesmos sejam aplicados. Neste caso, os *tokens* correspondem aos nomes das regras e terminais, bem como a caracteres como a vírgula ou os parênteses. Para efetuar esta associação criámos a classe *Pds16TokenAtributeIdMapper*, que estende de *DefaultAntlrTokenAttributeIdMapper*, Figura 15. O método redefinido, "*caculateId*", trata de

retornar o *id* do estilo a associar a um dado *token*, dado o seu nome e o seu tipo (*id*), *tokenName* e *tokenType*, respetivamente.

```
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 15 - Excerto de código de Pds16TokenAtributeIdMapper.

Depois de ter ambas as classes definidas, apenas é necessário registar que pretendemos utiliza-las em vez das classes que calculam o *highlighting* por definição. Este registo é efetuado através da classe que define o *UiModule* do projeto, neste caso *AbstractPds16UiModule*, os métodos responsáveis por este trabalho, Figura 16.

```
class Pds16asmUiModule extends AbstractPds16asmUiModule {

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

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

}
```

Figura 16 - Código da classe AbstractPds16asmUiModule.

3.3.2 Outline

O *Outline* é uma funcionalidade que permite ao programador navegar facilmente entre o seu código. Trata-se de uma janela que mostra, de acordo com algumas definições, a estrutura do ficheiro aberto na área de edição, listando assim os elementos que o ficheiro contém. Ao selecionar-se um elemento desta janela irá ser selecionado o elemento correspondente no editor de texto. Consequentemente, esta lista de elementos permite implementar atalhos para certas zonas do código, conforme o elemento definido.

Os elementos a mostrar na janela *Outline* podem ser configurados e variar conforme a linguagem de programação alvo. No nosso caso, definimos que apenas alguns dos elementos do *assembly* PDS16 devem constar na lista do *outline*, sendo estes os seguintes: símbolos (i.e. *labels*) e algumas diretivas, tais como *end*, *text*, *data*, *bss*, *section*, *org*, *equ* e *set*. Limitámos os elementos pois não faria sentido para o utilizador ter mencionadas todas as instruções nesta lista, deixando

de ser prático. Assim, com apenas estes elementos, o utilizador consegue navegar entre secções de código e dados diferentes e selecionar *labels*, que normalmente são associadas a instruções importantes (por exemplo, na definição de rotinas e na implementação de estruturas do tipo if/else, while, etc) ou a variáveis em memória.

Para definir os elementos que pretendemos disponibilizar no *outline* temos que filtrá-los. Para esse efeito usamos a classe gerada *Pds16asmOutlineTreeProvider*, que estende de *DefaultOutlineTreeProvider*, onde é feito *override* ao método "*_createNode*", como se pode verificar na Figura 17. Este método recebe como parâmetro o nó acima (na lista de elementos já presentes no *outline*) e o elemento do modelo a analisar, "*parentNode*" e "*modelElement*", respetivamente, com o objetivo de criar um novo nó através do elemento e adicioná-lo ao nó já presente no *outline*, i.e. "*parentNode*".

```
class Pds16asmOutlineTreeProvider extends DefaultOutlineTreeProvider{


    override _createNode(IOutlineNode parentNode, EObject modelElement){
        if (modelElement instanceof Label){
            setOutline(parentNode,modelElement)
        }

        else if (modelElement instanceof Directive){
            var element = (modelElement as DirectiveImpl).value
            if(element instanceof Bss || element instanceof Data ||
               element instanceof End || element instanceof Text ||
               element instanceof Equ || element instanceof Org  ||
               element instanceof Set || element instanceof Section){
                    setOutline(parentNode,element)
            }
        }
    }

    def setOutline(IOutlineNode parentNode, EObject obj){
        var Object text = textDispatcher.invoke(obj);
        if (text == null && isLeafDispatcher.invoke(obj))
            return;
        var Image image = imageDispatcher.invoke(obj);
        createEObjectNode(parentNode, obj, image, text, true);
    }
}
```

Figura 17 - Excerto de código de Pds16asmOutlineTreeProvider.

Para desenvolvermos um *outline* personalizado para a linguagem *assembly* PDS16, tivemos que analisar o "*modelElement*" de modo a pudermos rejeitar a criação de um novo nó caso este não pertença ao conjunto pretendido. Concretamente, só é criado um novo nó caso este seja uma instância de *Label* ou um elemento específico de *Directive*. Para a criação do nó é chamado o método auxiliar "*setOutline*" onde é calculado o texto e a imagem associados ao objeto, necessários para a criação do nó e adição à lista de *outline* (chamada a "*createEObjectNode*").

Com o objetivo de especificar com maior detalhe o nome que exibido em cada elemento do *Outline*, utilizámos a classe *Pds16asmLabelProvider* (gerada pela *framework*), que estende de

DefaultEObjectLabelProvider, em que para cada tipo de regra da gramática suportada pelo nosso *outline* é calculado o nome a apresentar no elemento final (Figura 18).

```
class Pds16asmLabelProvider extends DefaultEObjectLabelProvider {

    @Inject
    new(AdapterFactoryLabelProvider delegate) {
        super(delegate);
    }

    def text(Label l) {
        return 'Label: ' + l.labelName.substring(0, l.labelName.length-1)
    }

    def text(Equ ele){
        return 'Equ: ' + ele.id
    }

    def text(Org ele){
        return 'Org '
    }

    def text(Section ele){
        return 'Section: ' + ele.id
    }

    def text(Set ele){
        return 'Set: ' + ele.id
    }
}
```

Figura 18 - Excerto de código de Pds16asmLabelProvider.

Cada método "*text*" recebe como parâmetro um objeto que representa o elemento, através do qual é possível aceder a propriedades específicas do objeto que ajudam na construção da *label* que aparecerá na janela de *outline*. Devido ao facto de estes métodos terem entre si um objeto de um tipo diferente como parâmetro, é possível serem invocados através do "*textDispatcher*" (presente na Figura 17) por reflexão, com base no tipo de objeto a analisar [23].

3.3.3 Gerador

A *framework* Xtext disponibiliza os meios necessários à criação de um compilador através da classe responsável pela geração de código (gerada pela *framework*). No nosso projeto essa classe é a "*Pds16asmGenerator*", que contém apenas a definição de um método, "*doGenerate*" e que é invocado automaticamente, por definição, sempre que se guarda um ficheiro que já tenha sido validado e analisado, ou seja, que não contenha qualquer erro de validação. Neste método é gerado o código compilado referente a um dado ficheiro fonte com base nos objetos modelo da linguagem presentes na AST recebida como parâmetro.

Uma vez que este projeto não tinha como objetivo a definição de um novo assemblador, mas pelo contrário a utilização de um assemblador externo (i.e. o DASM), a implementação do método acima referido consiste na invocar deste assemblador e na marcação no ficheiro fonte dos eventuais erros de assemblagem (Figura 19).

```
val String SYSTEM_ENV_DASM = "DASM_PATH"

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

    if(!existsEnd(resource)){
        val errors = new ArrayList<LinedError>()
        //add new error to mark
        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) {...}

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){...}
```

Figura 19 - Excerto de código da classe Pds16asmGenerator.

Na nossa implementação do método *doGenerate* (Figura 19) começámos por verificar a existência do elemento "end" no ficheiro fonte (representado pelo *token* "*.end*"), pois caso não exista, o ficheiro não será válido, e não é necessária a chamada ao assemblador.

Após esta verificação, invocamos o assemblador DASM (usando o *path* do programa em questão) através de um "*ProcessBuilder*", que é aclasse usada para criar processos do sistema operativo. Após ser feita esta chamada, é capturado o *output* retornado pelo processo em formato de *InputStream<String>*. Estes dados são processados para se obter os eventuais erros de compilação. Para tal, foi criada a classe "*DasmErrorParser*" que contém apenas um método estático que dado o *InputStream* recebido como parâmetro retorna uma lista de objetos do tipo *LinedError* que contém a descrição e a linha do(s) erro(s) no ficheiro fonte.

Como existe a possibilidade de interação com a interface do editor do Eclipse, a qual dispõe de um mecanismo de marcação no código, decidimos efetuar marcações dos erros retornados pelo assemblador. Assim, tendo a lista de erros acima mencionada, por cada erro é criada 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 [26].

3.3.4 Geração 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 o anexo “Criação do plug-in para o Eclipse”.

4 Conclusões

Embora já exista um assemblador e um *debugger* para suportar realização de programas em processadores que implementam a arquitetura PDS16, não existia, até este momento, um editor de texto que suportasse, de uma eficiente, o seu desenvolvimento usando a linguagem *assembly* PDS16. Neste projeto recorreu-se à *framework* Xtext para criar um *plug-in* que, conjugado com o ambiente de desenvolvimento Eclipse, permite disponibilizar aos programadores um editor de texto que está integrado com o assemblador DASM, criando-se assim uma ferramenta de trabalho que favorece a tarefa do programador. O *plug-in* realizado, denominado PDS16inEcplise, está disponível *online* na página do projeto [27] e poderá ser usado, entre outros, pelos alunos que frequentem a unidade curricular Arquitetura de Computadores do ISEL como uma ferramenta de auxílio na aprendizagem da arquitetura PDS16.

Com a realização deste projeto conseguimos produzir uma versão estável do *plug-in* PDS16inEcplise, atingindo todos os pontos obrigatórios propostos por nós na proposta do projeto. Não obstante, existem melhorias que podem ser realizadas nas funcionalidades já implementadas, como por exemplo, a forma como foi conseguida a integração com o assemblador DASM.

Existem ainda vários desafios interessantes que podem ser abordados no futuro, relacionados com a adição de mais características ao *plug-in*, como por exemplo:

- *Deploy* para outras plataformas, tais como o IntelliJ ou *Browser*;
- Adicionar a funcionalidade *help* para cada instrução da gramática;
- Criar um assemblador próprio recorrendo aos mecanismos disponibilizados pela *framework* Xtext;
- Incluir no *plug-in* as funcionalidades de uma ferramenta de *debug*, semelhantes às disponibilizadas pela aplicação PDDebugger v2.0 [28], para se poder fazer todo o ciclo de desenvolvimento de um programa dentro da mesma plataforma.

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," em *Arquitetura de Computadores – Textos de apoio às aulas teóricas*, Lisboa, 2011, pp. 13-1 – 13-27.

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

[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, "Estrutura Interna do PDS16," em *Arquitetura de Computadores – Textos de apoio às aulas teóricas*, Lisboa, 2011, pp. 14-1 - 14-14.

[11] J. Paraiso, "Interrupções," em *Arquitetura de Computadores – Textos de apoio às aulas teóricas*, Lisboa, 2011, pp. 19-2 - 19-8.

[12] Wikipedia, "Intel HEX," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Intel_HEX. [Acedido em 21 7 2016].

[13] J. Paraíso, *PDS16 Quick Reference & SPD16 User Manual*, Lisboa, 2011.

[14] T. E. Foundation, "Eclipse Modeling Framework (EMF)," The Eclipse Foundation, [Online]. Available: https://eclipse.org/modeling/emf/. [Acedido em 13 7 2016].

[15] Model-driven Pretty Printer for Xtext, Prague, 2012.

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

[17] “Xtend Documentation,” [Online]. Available: https://www.eclipse.org/xtend/documentation/index.html. [Acedido em 13 7 2016].

[18] “Xtext Documentation - Configuration,” [Online]. Available: https://eclipse.org/Xtext/documentation/302_configuration.html. [Acedido em 21 7 2016].

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

[20] Wikipedia, “Plain Old Java Object,” [Online]. Available: https://en.wikipedia.org/wiki/Plain_Old_Java_Object. [Acedido em 15 7 2016].

[21] ANTLR / Terence Parr, “About The ANTLR Parser Generator,” 2014. [Online]. Available: http://www.antlr.org/about.html. [Acedido em 15 7 2016].

[22] Wikipedia, “Abstract syntax tree,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Abstract_syntax_tree. [Acedido em 19 7 2016].

[23] “Xtext Documentation - Eclipse Support,” [Online]. Available: https://www.eclipse.org/Xtext/documentation/310_eclipse_support.html. [Acedido em 21 7 2016].

[24] S. Hungerecker, SALTXT: An Xtext-based Extendable Temporal Logic, Lübeck, 2014.

[25] Google, “Google Guice,” [Online]. Available: https://github.com/google/guice. [Acedido em 15 7 2016].

[26] “Resource markers,” [Online]. Available: http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_markers.htm. [Acedido em 21 7 2016].

[27] “PDS16inEclipse,” [Online]. Available: http://tiagojvo.github.io/PDS16inEclipse/. [Acedido em 21 7 2016].

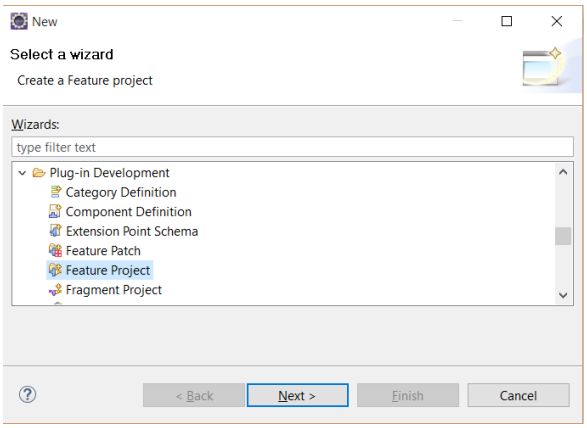
[28] “PDDebugger,” [Online]. Available: http://pwp.net.ipl.pt/cc.isel/ezeq/arquitetura/sistemas_didaticos/pds16/ferramentas/PDDebugger_v_2_0.rar. [Acedido em 23 7 2016].

A.1. Criação do *plug-in* para o Eclipse

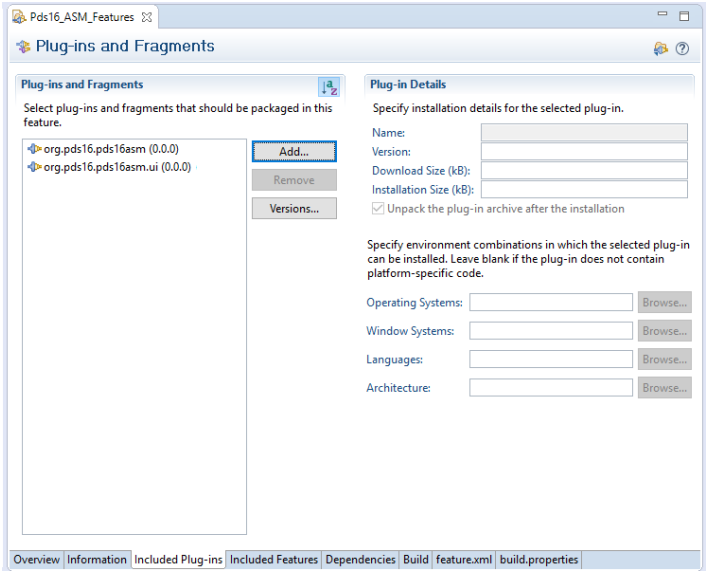
Após o desenvolvimento do editor de texto para a linguagem Assembly PDS16, usando a *framework* Xtext, decidimos publicar o *software* para poder ser instalado noutras máquinas. Como o software tem que ser acoplado com um IDE, neste caso o Eclipse, criámos um *plug-in* com o nome PDS16inEcplise que adicionará as novas funcionalidades ao mesmo. Este não só contém o *software* desenvolvido como também as dependências do mesmo. No processo de *deploy* foi tido em conta o controlo de versões do *plug-in*, podendo este ser atualizado manualmente pelo utilizador quando for lançada uma nova versão.

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

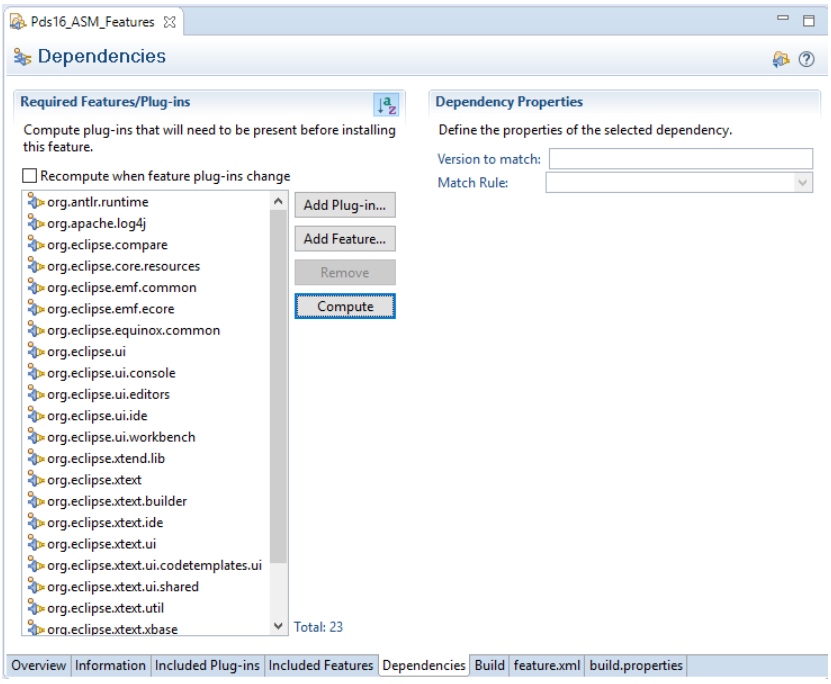
1. Criar um "*Feature Project*" no Eclipse.



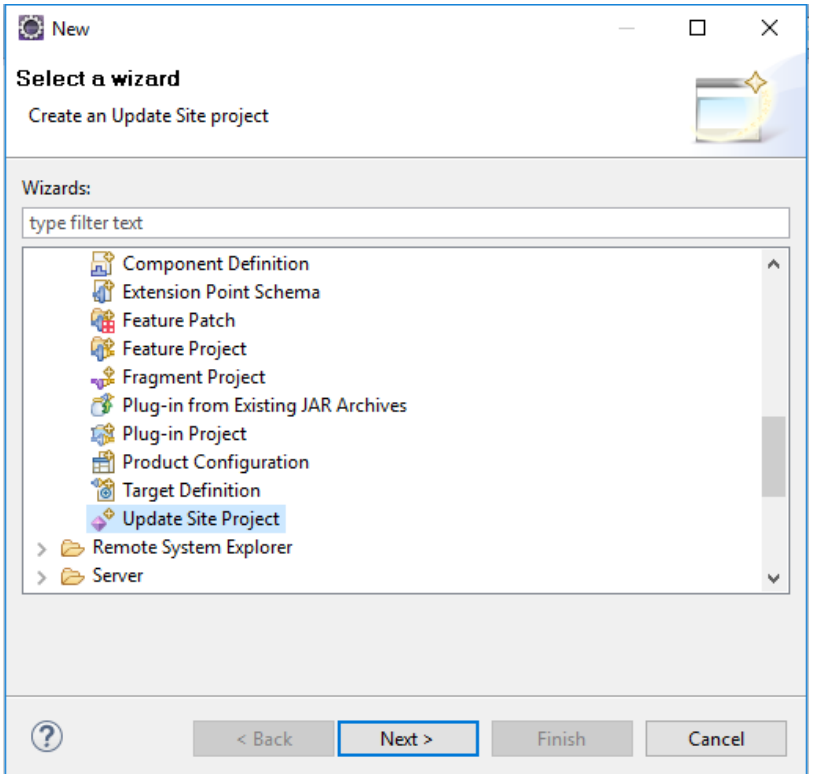
2. Abrir o ficheiro "*feauture.xml*" no projeto "*Feature*" criado anteriormente e abrir a *tab* "*plug-in*". Nessa *tab*, selecionar o botão "*Add*" e adicionar os respetivos projetos. Neste caso foram adicionados os 2 projetos correspondentes ao *software* em desenvolvimento.



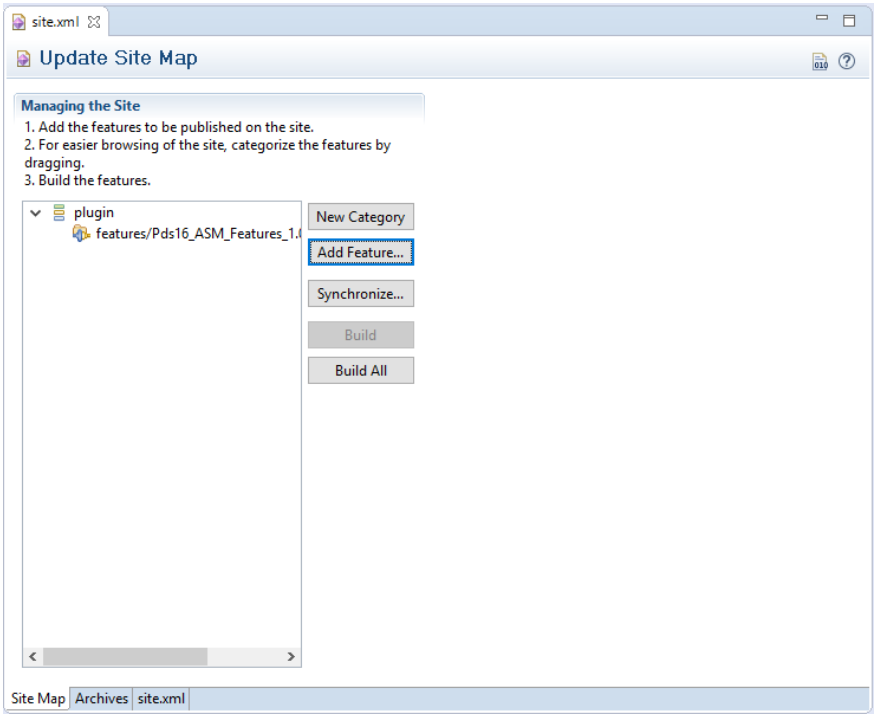
3. Na *tab* "*Dependencies*", selecionar "*Compute*" para incluir automaticamente todas as bibliotecas dos quais os projetos do passo anterior são dependentes.



4. Criar um "*Update Site Project*"



5. Neste último passo é necessário adicionar o projeto "*Feature*" criado anteriormente ao projeto "*Update Site*". Para tal, deve abrir-se o ficheiro "*site.xml*" e no *tab* "*Site Map*" selecionar "*Add Feature*" e de seguida selecionar o projeto "*Feature*" criado. Depois, deve-se selecionar o 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 Eclipse através do *link* do website alojado no Github.

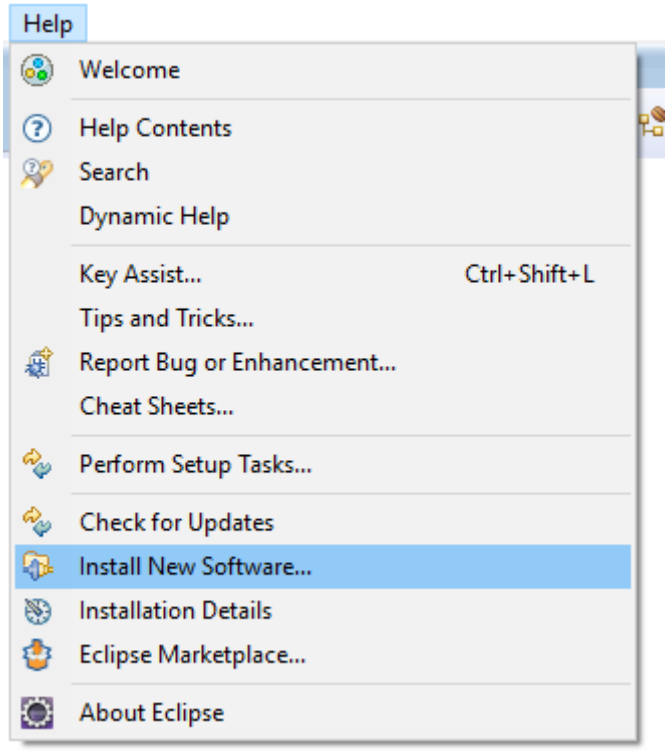
A.2. Instalação do *plug-in*

Para fazer o uso correto do editor de texto é necessário instalar o *plug-in* PDS16inEcplise e definir uma variável de ambiente com a *path* do assemblador DASM, que pode ser obtido a partir do seguinte endereço URL:
http://pwp.net.ipl.pt/cc.isel/ezeq/arquitetura/sistemas_didaticos/pds16/ferramentas/dasm.exe

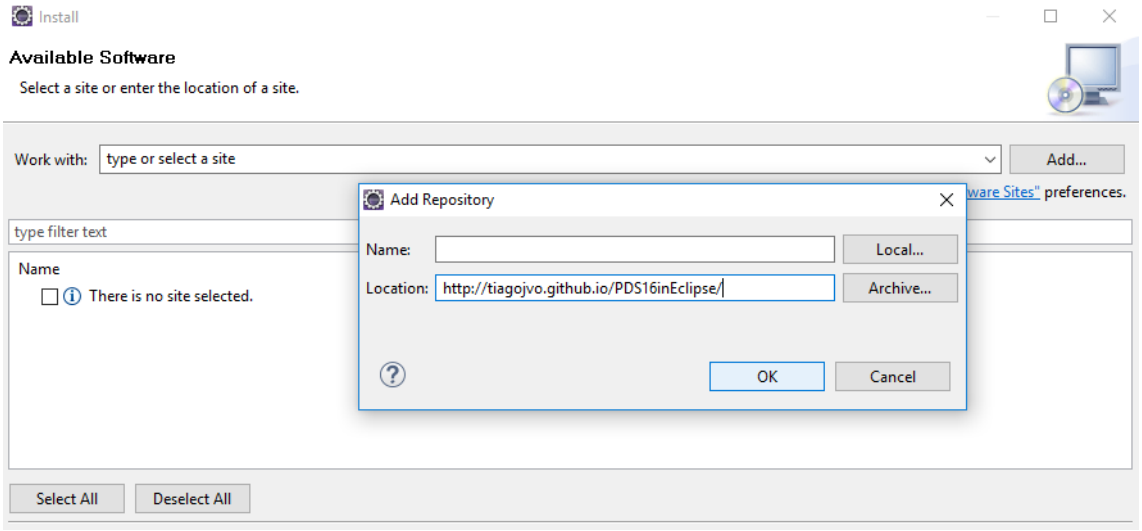
O *plug-in* PDS16inEcplise pode ser instalado no *IDE* Ecplise de duas maneiras distintas: usando o apontador http://tiagojvo.github.io/PDS16inEclipse/ ou descarregando o ficheiro ZIP disponível também a partir deste apontador.

Para instalar o *plug-in*, seja qual for a sua fonte, é necessário seguir os seguintes passos:

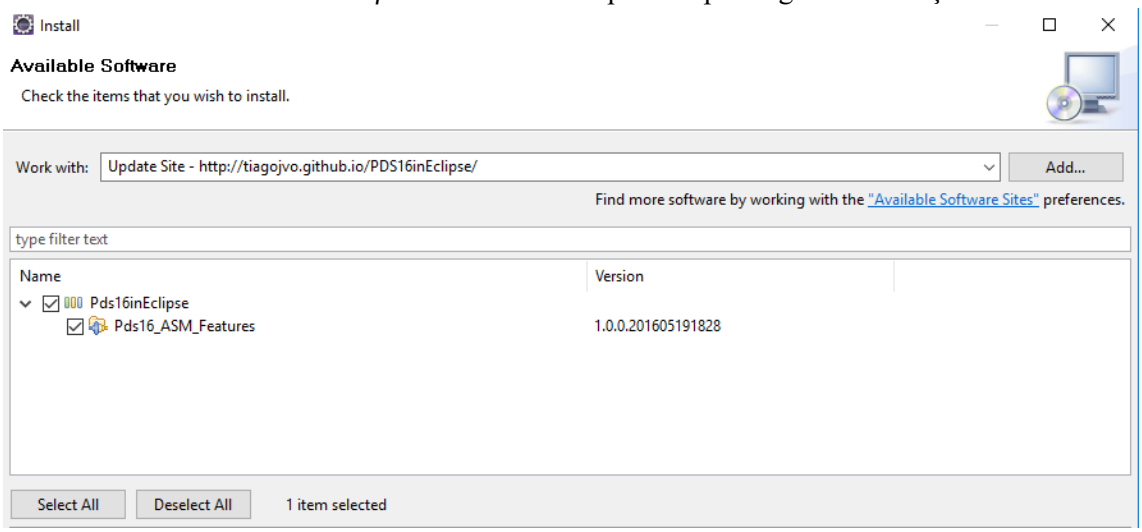
1. Definir uma variável de ambiente do sistema operativo Windows com o nome "DASM_PATH" com a respetiva *path* do assemblador, reiniciando de seguida o sistema operativo para que esta fique disponível.
2. Efetuar os seguintes passos no *IDE* Eclipse:
    a. Selecionar a tab "*Help*" -> "*Install New Software*";



    b. Selecionar "*Add*" e no campo "*Location*" colocar o endereço web do *plug-in* PDS16inEcplise ou, em alternativa, descompactar o ficheiro *ZIP* e selecionar o ficheiro "*contente.jar*" presente na raiz da pasta descompactada.



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



Após estes passos, para utilizar o *plug-in* PDS16inEcplise basta seguir os seguintes passos no *IDE* Eclipse:

1. Criar um novo projeto do tipo *Java Project*;
2. No projeto criado, adicionar um novo ficheiro dando-lhe a extensão ".*asm*".