

**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. É também objetivo deste projeto criar uma documentação sobre a arquitetura do processador PDS16.

O desenvolvimento da ferramenta PDS16inEclipse é baseado na *framework* Xtext tendo como plataforma alvo o Ambiente Integrado de Desenvolvimento (*IDE*) Ecplise. 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

[Resumo v](#_Toc456867419)

[Índice vii](#_Toc456867420)

[Lista de Figuras ix](#_Toc456867421)

[Lista de Tabelas xi](#_Toc456867422)

[1 Introdução 2](#_Toc456867423)

[1.1 Enquadramento 2](#_Toc456867424)

[1.2 Motivação 3](#_Toc456867425)

[1.3 Objetivos 4](#_Toc456867426)

[1.4 Estrutura do documento 5](#_Toc456867427)

[2 Arquitetura PDS16 6](#_Toc456867428)

[2.1 Registos 6](#_Toc456867429)

[2.2 Conjunto de instruções 9](#_Toc456867430)

[2.2.1 Processamento de dados 10](#_Toc456867431)

[2.2.2 Transferência de dados 10](#_Toc456867432)

[2.2.3 Controlo do fluxo de execução 11](#_Toc456867433)

[2.3 Subsistema de memória 12](#_Toc456867434)

[2.4 Exceções 12](#_Toc456867435)

[2.5 Assemblador DASM 13](#_Toc456867436)

[2.5.1 Escrita de programas 14](#_Toc456867437)

[2.5.2 Diretivas 14](#_Toc456867438)

[3 Framework Xtext 16](#_Toc456867439)

[3.1 Arquitetura 18](#_Toc456867440)

[3.1.1 Modeling Workflow Engine (MWE2) 18](#_Toc456867441)

[3.2 Gramática 19](#_Toc456867442)

[3.2.1 Regras da gramática 20](#_Toc456867443)

[3.2.2 Definição dos elementos do analisador de regras 23](#_Toc456867444)

[3.3 Integração com a plataforma Eclipse 24](#_Toc456867445)

[3.3.1 Syntax Highlight 24](#_Toc456867446)

[3.3.2 Outline 25](#_Toc456867447)

[3.3.3 Gerador 27](#_Toc456867448)

[3.3.4 Configuração do plug-in 28](#_Toc456867449)

[4 Conclusões 30](#_Toc456867450)

[Referências 31](#_Toc456867451)

[A.1 - Deploy do plug-in para o Eclipse 34](#_Toc456867452)

[A.2 - Instalação do Plug-in 39](#_Toc456867453)

Lista de Figuras

[Figura 1 – Exemplo do ciclo de desenvolvimento de um programa/aplicação. [1] 2](#_Toc456867454)

[Figura 2 - Bancos de Registos PDS16 6](#_Toc456867455)

[Figura 3 –Estrutura interna do registo PSW. 7](#_Toc456867456)

[Figura 4 – Excerto de código de uma gramática Xtext 19](#_Toc456867457)

[Figura 5 - Classes geradas pela *framework* 20](#_Toc456867458)

[Figura 6 - Código exemplo da definição das regras 21](#_Toc456867459)

[Figura 7 - Código exemplo da definição regras terminais 21](#_Toc456867460)

[Figura 8 - Código da classe Pds16asmRuntimeModule 22](#_Toc456867461)

[Figura 9 - Excerto da classe PDS16asmValueConcerter 22](#_Toc456867462)

[Figura 10 - Interface IValueConverter 22](file:///C:\Users\tiago\Desktop\proj\repo\Relatorio%20Final\Relatorio-Final-v1.2.docx#_Toc456867463)

[Figura 11 - Exemplo de um validador 23](#_Toc456867464)

[Figura 12- Excerto de código de Pds16HighlightingConfiguration 24](#_Toc456867465)

[Figura 13 - Excerto de código de Pds16TokenAtributeIdMapper 25](#_Toc456867466)

[Figura 14 - Código da classe AbstractPds16asmUiModule 25](#_Toc456867467)

[Figura 15 - Excerto de código de Pds16asmOutlineTreeProvider 26](file:///C:\Users\tiago\Desktop\proj\repo\Relatorio%20Final\Relatorio-Final-v1.2.docx#_Toc456867468)

[Figura 16 - Excerto de código de Pds16asmLabelProvider 27](file:///C:\Users\tiago\Desktop\proj\repo\Relatorio%20Final\Relatorio-Final-v1.2.docx#_Toc456867469)

[Figura 17 - Excerto de código da classe Pds16asmGenerator 28](#_Toc456867470)

Lista de Tabelas

[Tabela 1 - Sintaxe das instruções PDS16. 9](#_Toc456867471)

[Tabela 2 - Palavras-chave da sintaxe PDS16. 9](#_Toc456867472)

[Tabela 3 - Elementos da sintaxe gramatical Xtext 20](#_Toc456867473)

1. Introdução
   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]](#footnote-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. 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* 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*.

## 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 foiconseguido 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 e Internet Explorer.

* 1. Estrutura do documento

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

* Capitulo 1 – é feito o enquadramento do trabalho, em que âmbito se insere, e os objetivos definidos;
* Capitulo 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;
* Capitulo 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;
* Capitulo 4 – é feito um resumo do que poderá ser melhorado no futuro e são apresentadas sugestões para a continuação do desenvolvimento da ferramenta.

1. 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.

* 1. Registos

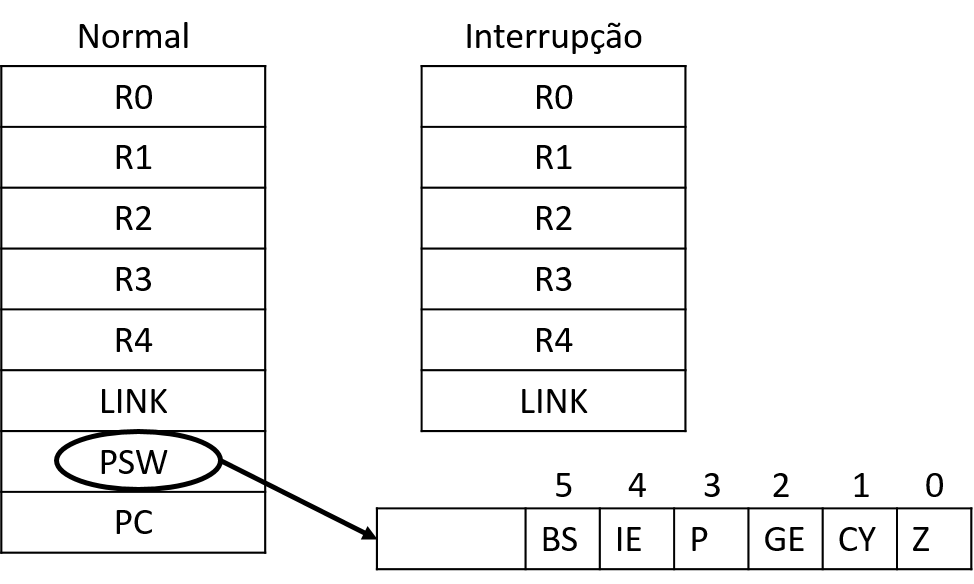
A arquitetura PDS16 inclui dois bancos de registos, ilustrados na Figura 2, que visam suportar, de uma forma eficiente, o funcionamento nos seus dois modos de operação: o modo normal e o modo de interrupção.

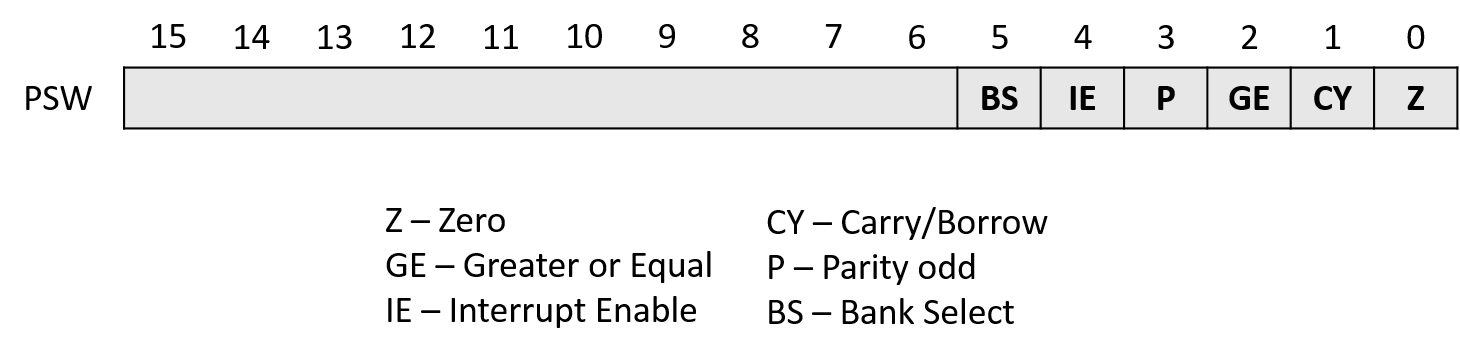
Figura 2 - Bancos de Registos PDS16

O banco de registos acessível 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 memoria 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 outros os outros dois registos especiais do processador. O registo R6, não obstante também poder ser usado como operador 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.

* 1. 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 | 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 | Descrição |
| 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.

* + 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 do PDS16:

LDI  R0, #0xFF

LDIH R0, #0xFF

* + 1. 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 do PDS16 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.

* + 1. 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.

* 1. 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).

* 1. Exceções

Um mecanismo de exceção visa o tratamento de eventos assíncronos ou inesperados 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.

Para garantir indivisibilidade entre as várias ações necessárias à realização do retorno da interrupção foi criada a instrução IRET. Esta instrução copia em simultâneo o valor do registo LR e R0 do banco de interrupção para o registo PC e PSW do banco normal, respetivamente. Se admitirmos que o registo R0 do banco de interrupção mantém o valor de PSW do banco normal no momento da interrupção, então é reposto o estado do programa interrompido e a permissão às interrupções, sendo também restabelecido o banco de registos normal como selecionado para a execução da aplicação.

* 1. 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], uni modelar, 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.

Sendo o DASM um assemblador didático e uni modular, não suporta o desenvolvimento de aplicações usando múltiplos ficheiros fontes, fazendo-se a localização em memória das instruções, variáveis e constantes de forma estática, estabelecida no ficheiro fonte. 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 memória. Caso existam erros de compilação, os mesmos são assinalados na respetiva instrução com uma mensagem identificadora do seu tipo e da possível causa.

* + 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 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:** 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.
  + 1. 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. 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” – 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.

1. Framework Xtext

Xtext é uma *framework* 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 continuidade com a Eclipse Modeling Framework [14] (EMF) que permite a conversão de código numa dada linguagem (neste caso a gramática) para um modelo que poderá ser posteriormente transformado noutro modelo, ou serializado para outra linguagem. [15] O motivo pelo qual é necessário associar este modelo ao código, é a necessidade de ter uma *meta-data* à qual referir na implementação das regras de dada linguagem.

Com *a* *framework* Xtext é possível definir toda a sintaxe e semântica de uma linguagem tendo assim como resultado uma infraestrutura que poderá incluir *parser*, *linker*, *typechecker*, compilador bem como suporte à edição. [16] É possível obter um editor utilizando uma plataforma, como é o caso do ambiente de desenvolvimento Eclipse ou do IntelliJ IDEA, e também através de um *browser*.

O primeiro passo para o desenvolvimento de um *plug-in* utilizando esta *framework*, é a instalação do *plug-in* da *framework* Xtext no IDE de desenvolvimento, neste caso o Eclipse, e a criação um novo projeto do tipo “*Xtext Project*”. De seguida é necessário a definição de uma gramática (onde é descrita a sintaxe da linguagem) e de todas as funcionalidades a implementar, como o caso do *highlighting*, validadores, integração com um compilador, etc. Dadas estas implementações o último passo será a publicação da ferramenta, através da geração de um *plug-in* que integrará todas as funcionalidades acima descritas.

* 1. Arquitetura

Xtext é uma *framework* Eclipse desenvolvida com base na linguagem de programação Java.

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

Sendo assim, aquando da criação de um projeto Xtext, este encontrar-se-á divido em vários packages com diferentes responsabilidades, sendo que os mais importantes são:

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

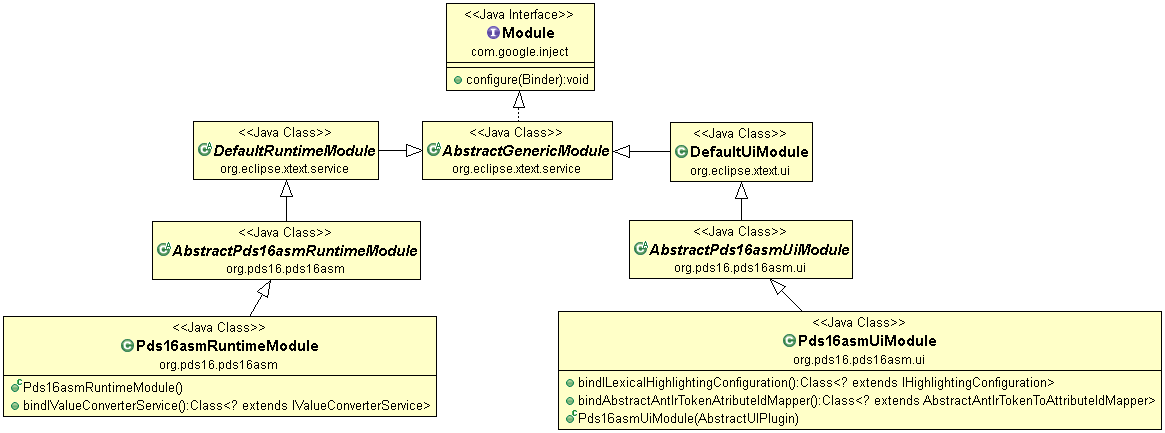
Cada um destes *packages* contém um módulo responsável por indicar que componentes deverão ser utilizadas para cada tarefa. Por exemplo, para uma linguagem denominada de “*MyDsl*”, irá existir um “*MyDslRuntimeModule*” e um “*MyDslUiModule*”, sendo que cada um deles contém métodos para a associação de cada um dos componentes acima mencionados. Estes 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 á organização de Módulos

Dado este modo implementação, ponto principal de um projeto na *framework* Xtext é a sua gramática (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 a pelo *Modeling Workflow Engine* (MWE2).

* + 1. Modeling Workflow Engine (MWE2)

Existe a possibilidade de, através da configuração do *Modeling Workflow Engine* [18], configurar a forma de como o código é gerado, como por exemplo definir se a linguagem é ou não *case-sensitive*. O MWE2 é baseado no modelo *plain old java object* (POJO [19]), 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]

O ficheiro de configuração contém uma componente denominada de *Generator*, neste caso a *class XtextGenerator* que é o *entry point* para a geração do *plug-in* da linguagem. Este tipo de componente é constituído por fragmentos que por sua vez 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 [20], geradas pelo Xtext: uma para produção do "*parser*" onde uma *Abstract Syntax Tree* [21](AST)é produzida, outra que é utilizada para o processamento dos eventos do editor do *Eclipse*. [23] A sintaxe do ficheiro de gramatica Xtext é igual à do ANTLR, à exceção de que para a verificação semântica, o ANTLR contém código Java embebido na descrição da gramática, e o Xtext faz uso de injeção de dependências para referir que tipos devem ser gerados para essa mesma verificação. [22] Estas dependências são resolvidas utilizando a *framework* Google Guice [23], 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]

* 1. Gramática

Com o estudo das instruções do PDS16 e das diretivas (e mnemónicas) do DASM (através a documentação [13], [6] e [7]), foi possível definir uma gramática tendo em conta as possíveis formas de escrever uma dada instrução desta linguagem.

Apesar de ser possível converter um dado código num modelo, tem que ser definido um *template* representativo desse mesmo modelo, que deverá ser seguido para que seja criado um modelo coerente entre as várias instruções. No caso de um projeto Xtext, a gramática toma o papel de *template* de código.

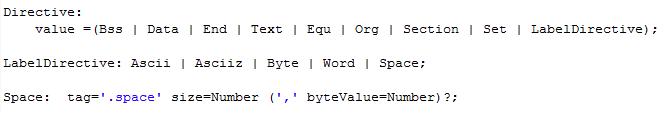
Assim a gramática é definida através de regras que podem referenciar outras regras ou palavras-chave. Por cada regra definida é gerada uma classe modelo, com métodos e atributos dependendo da definição da regra. Na geração das classes modelo, será adicionada a dependência entre elas como por exemplo nas seguintes regras da Figura 4:

Figura 5 – Excerto de código de uma gramática Xtext

Na Figura 4 podemos apontar que a regra “*Space*” depende de “*LabelDirective*”, o que irá traduzir-se numa dependência entre elas. 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.

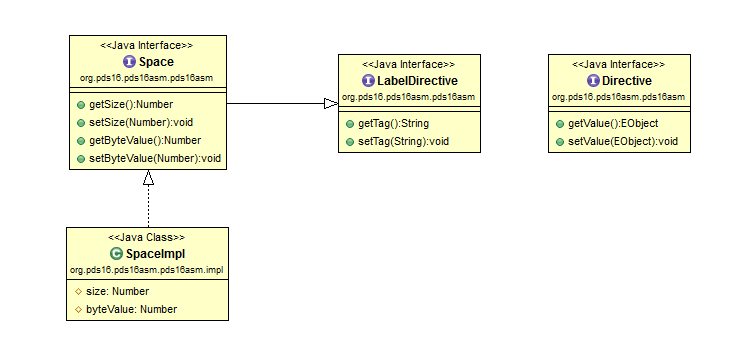
Essa dependência é tratada pelo Xtext gerando automaticamente classes em Java quando o *Modeling Workflow Engine 2* (MWE2 [18]) é executado, resolvendo essa dependência pela extensão entre classes, criando assim uma hierarquia entre as regras de uma DSL (Figura 5).

Figura 6 - Classes geradas pela *framework*

Na Figura 5 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 implementação (“*SpaceImpl*”), estas propriedades traduzir-se-ão em campos, acedidos através dos *getters* e *setters* implementados de acordo com a declaração em “*Space*”.

* + 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.

|  |  |
| --- | --- |
| 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 |

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 tratam-se de operadores que auxiliam a construção de regras de modo conseguir implementar definições específicas de cada uma delas. A Tabela 3 contém alguns exemplos destes elementos:

Tabela 3 - Elementos da sintaxe gramatical Xtext

Dadas estas definições de elementos, é possível começar a definição de regras sendo que existem dois tipos de regras de gramática em Xtext: *Parser Rules* e *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]](#footnote-2)mas sim uma árvore de regras terminais e não terminais, as deniminadas *parse tree* ou *node model*.

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 6 são apresentadas algumas *parser rules* definidas para o projeto.

Figura 7 - Código exemplo da definição das regras

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

Por outro lado, na regra “*Label”* podemos verificar que na sua definição existe uma palavras-chaves (‘:’), 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 – ‘$’.

*Terminal Rules,* também denominada por *token rule*s ou *lexer rules*, tratam-se de um tipo de regra que representada por *tokens*, usualmente definidos por expressões regulares.

Uma regra terminal é definida pela palavra-chave “*terminal*”, seguida do nome da regra (em letras maiúsculas por convenção) e do caracter ‘:’, e da expressão regular que a define. A Figura 7 contém a definição de duas regras terminais implementadas no projeto.

Figura 8 - Código exemplo da definição regras terminais

O primeiro terminal, “*ID*”, começa com um caracter de ‘a’ a ‘z’ (maiúsculo ou minúsculo) ou pelo caracter ‘\_’ seguindo de nenhum ou mais caracteres (devido ao elemento de sintaxe ‘\*’), incluindo números.

Um terminal retorna sempre um tipo, que por definição se trata de uma *String*, sendo possível manipula-lo para o tipo específico pretendido. O terminal “*HEX*” trata-se da definição de um número 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 na definição da mesma indicar o tipo que se pretende retornar, através 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 é o redefinir o método “*bindIValueConverter*” na classe que representa o *RunTimeModule* do projeto em questão, neste caso “*Pds16RunTimeModule*”, Figura 8. Este método retorna a classe responsável pela conversão dos tipos de retorno das regras definidas na gramática.



Figura 9 - Código da classe Pds16asmRuntimeModule

A classe “*Pds16asmValueConverter*” implementa a interface “*IValeuConverterService*”, onde através de anotação de métodos (para injeção de dependência, como descrito nas secções anteriores), são definidas as regras em que se pretende converter o tipo de retorno, e qual a classe responsável pela conversão, Figura 9.

Figura 10 - Excerto da classe PDS16asmValueConcerter

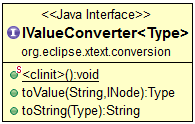
Como presente na figura, a anotação “*@ValueConverter(rule=“HEX”)*”, indica que o método por ela anotado, retornará um conversor para o tipo de retorno (neste caso para *Integer*) da regra com o nome *“HEX”*, sendo que se trata de uma instância da classe “*HEXValueConverter*”, que por sua vez terá de implementar a interface “*IValueConverter*” (Figura 10)*.*

Figura - Interface IValueConverter

* + 1. Definição dos elementos do analisador de regras

Como em todos as linguagens, existe a necessidade de validar regras de semântica. Estas validações não são possíveis apenas através da definição de uma gramática (*par*ser *rules* e *terminal rules*), logo têm que ser verificadas no ato de escrita de código.

A *framework* disponibiliza um mecanismo de validação que satisfaz essa necessidade, permitindo assim analisar o conteúdo de uma regra e indicar ao utilizador caso exista um erro. Estas verificações são feitas na classe responsável por validações, no nosso caso “*Pds16asmValidator*”, que é gerada pela *framework*. O mecanismo passa pelo mesmo mencionado nas secções acima, onde é feita injeção de dependência através de anotação de métodos, que neste caso se trata da anotação “*Check*” sobre um método que recebe como parâmetro uma instância da classe representativa da regra a analisar.

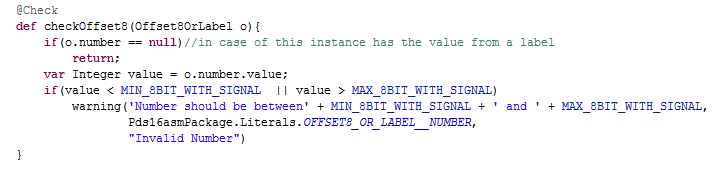
No caso do nosso no projeto verificamos os limites dos números presentes nas regras, por exemplo, uma regra que contenha um número representativo de um *offset* a8 *bits* com sinal, será verificado se o número inserido pelo utilizador se insere nos limites permitidos, emitindo um *warning* caso contrário (Figura 11).

Figura 12 - Exemplo de um validador

* 1. Integração com a plataforma Eclipse

A *framework* Xtext disponibiliza uma 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*. No nosso caso, geramos um *plug-in* compatível com o Eclipse, permitindo assim ao utilizador desenvolver programas em *assembly* de PDS16.

* + 1. 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 em cinco tipos de estilos: diretivas, nome 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 contém 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 no exemplo da Figura 12.

Figura 13- 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 13.

Figura 14 - Excerto de código de Pds16TokenAtributeIdMapper

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.

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 15.

Figura 15 - Código da classe AbstractPds16asmUiModule

* + 1. Outline

O *Outline* é uma funcionalidade que permite o programador navegar facilmente entre o seu código. Trata-se de uma janela que dispões a estrutura definida de um ficheiro que esteja aberto na área de edição, listando assim os elementos que o ficheiro contém. Essa lista de elementos permite assim ter um atalho para uma certa zona de código conforme o elemento definido. Ao selecionar um elemento da janela do *outline* irá ser selecionado no editor de texto o elemento correspondente. Estes elementos podem ser configurados e podem variar conforme a linguagem de programação. No nosso caso apenas definimos que apenas alguns dos elementos do *assembly* PDS16 devem constar na lista do *outline*, sendo estes os seguintes: *labels*, e algumas diretivas como o *bss*, *data*, *end*, *text*, *equ*, *org*, *set* e *section*. Limitamos 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 diferentes e selecionar *labels* (que são associadas normalmente a instruções importante ou variáveis em memória).

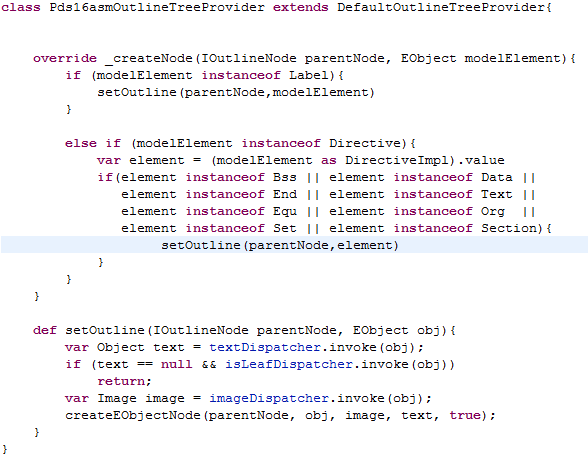
Para definir os elementos que queremos que estejam presentes no *outline* temos que filtra-los. Para esse efeito usamos a classe gerada *Pds16asmOutlineTreeProvider* que estende de *DefaultOutlineTreeProvider* onde é feito *override* ao método “\_*createNode*” como se pode verificar na Figura 15.

Figura - Excerto de código de Pds16asmOutlineTreeProvider

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 adiciona-lo ao nó já presente no *outline,* “*parentNode*”. Para termos um *outline* personalizado, tivemos que analisar o “*modelElement*” de modo a pudermos rejeitar a criação de um novo nó caso não pertença ao conjunto pretendido, neste caso, 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*”).

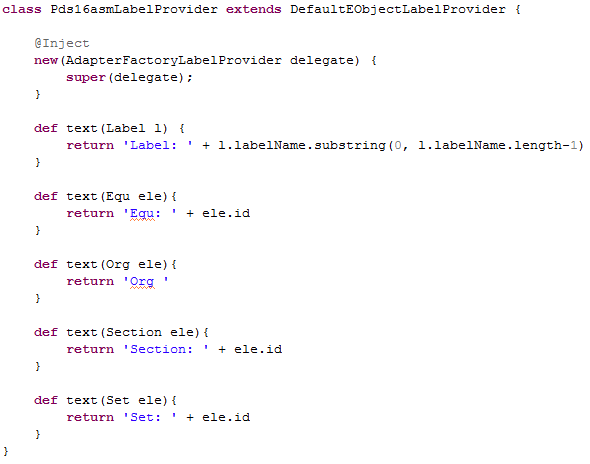
No nosso caso quisemos especificar com mais detalhe o nome que exibido em cada elemento do *Outline* e para isso usamos a classe *Pds16asmLabelProvider* (gerada pela framework) que estende de *DefaultEObjectLabelProvider*, onde para cada tipo de regra da gramatica suportada pelo nosso *outline* é calculado o nome a apresentar no elemento final.

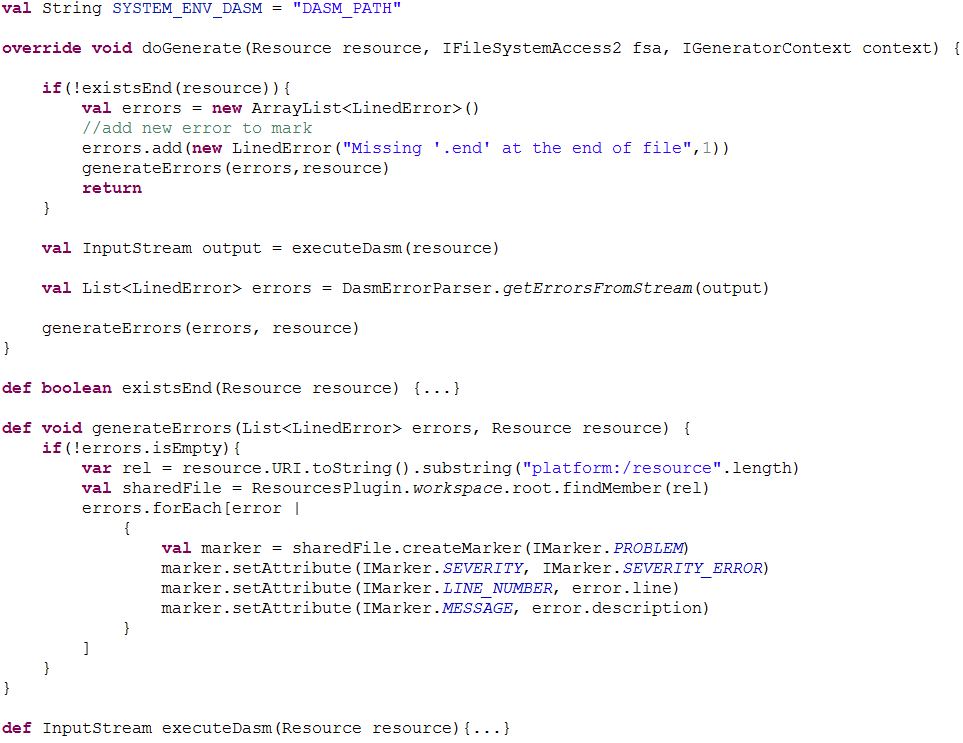
Figura - Excerto de código de Pds16asmLabelProvider

Cada método “*text*” definido presente na Figura 17 recebe como parâmetro um objeto que representa o elemento, e através dele conseguimos aceder a propriedades específicas do objeto que ajudam na construção da *label* que aparecerá na janela de *outline*. Devido ao facto de cada um destes métodos ter como parâmetro um objeto de um diferente tipo, permite que sejam evocados através do “*textDispatcher*” (presente na Figura 16) através de reflexão, dependendo do tipo de objeto a analisar. [23]

* + 1. Gerador

A *framework* disponibiliza a opção de criar um compilador, mas na implementação deste 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 contém apenas a definição de um método, *doGenerate*. Este método é chamado automaticamente, por definição, ao guardar um ficheiro que já tenha sido validado e analisado, ou seja que não contenha qualquer erro de validação.

Figura 18 - Excerto de código da classe Pds16asmGenerator

Na nossa implementação do método *doGenerate* (Figura 17) começámos por verificar a existência do elemento “End” no ficheiro (representado pelo *token* “*.end*”), 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.

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]

* + 1. 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 “A.1 - Geração do *plug-in* para o Ecplise”.

1. Conclusões

Embora exista um assemblador e um *debugger* criado especificamente para o processador PDS16, não existia até este momento um editor de texto que suportasse a linguagem de programação deste processador. Com este projeto criamos um *plug-in* que conjugado com um ambiente de desenvolvimento permite ter ao programador um editor de texto que está integrado com o assemblador DASM, criando assim uma ferramenta de trabalho que favorece o programador.

Existem vários desafios interessantes de serem abordados no futuro, relacionado com a continuação de implementação de mais características deste projeto PDS16inEclipse como por exemplo:

* *Deploy* para outras plataformas como o IntelliJ ou *Browser*;
* Adicionar a funcionalidade *help* em cada instrução da gramática;
* Criar o próprio assemblador DASM através da *framework* Xtext;
* Incluir as funcionalidades da ferramenta de *debug* já existente no *plug-in*;

Para além dos pontos referidos existem melhorias que podem ser realizadas nas funcionalidades já implementadas no projeto, como por exemplo a forma como esta a ser feita a integração do assemblador DASM com o *plug-in*.

Com a realização deste projeto conseguimos obter uma versão estável atingindo todos os pontos obrigatórios propostos por nós na proposta do projeto. O *plug-in* realizado está disponível *online* na página do projeto [27] e poderá ser usado pelos alunos da unidade curricular de Arquitetura de Computadores como uma ferramenta de auxílio na aprendizagem da arquitetura do processador PDS16.

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] | “MWE2 Documentation,” [Online]. Available: https://eclipse.org/Xtext/documentation/306\_mwe2.html. [Acedido em 10 6 2016]. |
| [19] | Wikipedia, “Plain Old Java Object,” [Online]. Available: https://en.wikipedia.org/wiki/Plain\_Old\_Java\_Object. [Acedido em 15 7 2016]. |
| [20] | ANTLR / Terence Parr, “About The ANTLR Parser Generator,” 2014. [Online]. Available: http://www.antlr.org/about.html. [Acedido em 15 7 2016]. |
| [21] | Wikipedia, “Abstract syntax tree,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Abstract\_syntax\_tree. [Acedido em 19 7 2016]. |
| [22] | S. Hungerecker, SALTXT: An Xtext-based Extendable Temporal Logic, Lübeck, 2014. |
| [23] | Google, “Google Guice,” [Online]. Available: https://github.com/google/guice. [Acedido em 15 7 2016]. |
| [24] | “PDS16inEclipse,” [Online]. Available: http://tiagojvo.github.io/PDS16inEclipse/. |

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.

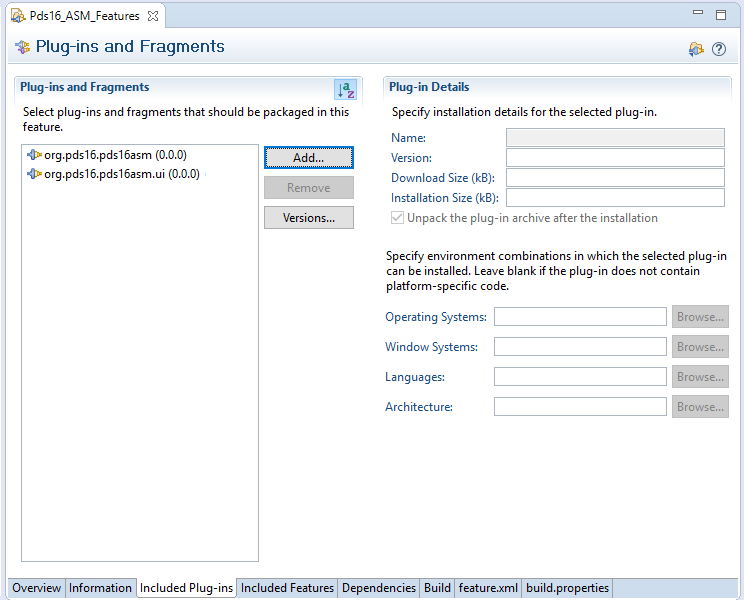
Como o software tem que ser acoplado com um IDE neste caso o Eclipse, criámos um *plug-in* que adicionará as novas funcionalidades ao mesmo. Este não só contém o *software* desenvolvido como também as dependências do mesmo. No processo de *deploy* foi tido em conta o controlo de versões do *plug-in*,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

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

1. Criar um “*Feature Project*” no Eclipse.



1. 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 dois projetos correspondentes ao *software* em desenvolvimento.



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



1. Criar um “*Update Site Project*”



1. 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 Eclipse 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:
   1. Clicar na tab “Help” -> “Install New Software”;



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

1. Programas que ajudam adicionar novas funcionalidades aos *plug-ins*. [↑](#footnote-ref-1)
2. Sequência de caracteres. [↑](#footnote-ref-2)