

**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 o assemblador DASM permite a assemblagem do código sem necessidade de sair do *IDE*, e também o processamento das pseudo instruções e diretivas suportados pelo assemblador. É também objetivo deste projeto criar uma documentação sobre a arquitetura do processador PDS16.

O desenvolvimento desta ferramenta 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 de uma gramática, e posteriormente analisadores da mesma para que fossem implementadas as funcionalidades acima referidas.

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

Índice

[Resumo v](#_Toc456799536)

[Índice vii](#_Toc456799537)

[Lista de Figuras ix](#_Toc456799538)

[Lista de Tabelas xi](#_Toc456799539)

[1 Introdução 2](#_Toc456799540)

[1.1 Enquadramento 2](#_Toc456799541)

[1.2 Motivação 3](#_Toc456799542)

[1.3 Objetivos 4](#_Toc456799543)

[1.4 Estrutura do documento 5](#_Toc456799544)

[2 Arquitetura PDS16 6](#_Toc456799545)

[2.1 Modelo de programação 6](#_Toc456799546)

[2.2 Registos 6](#_Toc456799547)

[2.3 Conjunto de instruções 9](#_Toc456799548)

[2.3.1 Processamento de dados 9](#_Toc456799549)

[2.3.2 Transferência de dados 10](#_Toc456799550)

[2.3.3 Controlo do fluxo de execução 11](#_Toc456799551)

[2.4 Subsistema de memória 12](#_Toc456799552)

[2.5 Exceções 12](#_Toc456799553)

[2.6 Assemblador DASM 13](#_Toc456799554)

[2.6.1 Escrita de programas 13](#_Toc456799555)

[2.6.2 Diretivas 14](#_Toc456799556)

[3 Framework Xtext 16](#_Toc456799557)

[3.1 Arquitetura 18](#_Toc456799558)

[3.1.1 Modeling Workflow Engine (MWE2) 18](#_Toc456799559)

[3.2 Gramática 19](#_Toc456799560)

[3.2.1 Regras da gramática 20](#_Toc456799561)

[3.2.2 Definição dos elementos do analisador de regras 23](#_Toc456799562)

[3.3 Integração com a plataforma Eclipse 24](#_Toc456799563)

[3.3.1 Syntax Highlight 24](#_Toc456799564)

[3.3.2 Outline 25](#_Toc456799565)

[3.3.3 Gerador 27](#_Toc456799566)

[3.3.4 Configuração do plug-in 28](#_Toc456799567)

[4 Conclusões 30](#_Toc456799568)

[Referências 31](#_Toc456799569)

[A.1 - Deploy do plug-in para o Eclipse 34](#_Toc456799570)

[A.2 - Instalação do Plug-in 39](#_Toc456799571)

Lista de Figuras

[Figura 1 – Exemplo do ciclo de desenvolvimento de um programa/aplicação. [1] 2](#_Toc456799572)

[Figura 2 –Estrutura interna do registo PSW. 8](#_Toc456799573)

[Figura 3 - Bancos de Registos PDS16 8](#_Toc456799574)

[Figura 4 – Excerto de código de uma gramática Xtext 19](#_Toc456799575)

[Figura 5 - Classes geradas pela *framework* 20](#_Toc456799576)

[Figura 6 - Código exemplo da definição das regras 21](#_Toc456799577)

[Figura 7 - Código exemplo da definição regras terminais 21](#_Toc456799578)

[Figura 8 - Código da classe Pds16asmRuntimeModule 22](#_Toc456799579)

[Figura 9 - Excerto da classe PDS16asmValueConcerter 22](#_Toc456799580)

[Figura 10 - Interface IValueConverter 22](file:///C:\Users\tiago\Desktop\Relatorio-Final-v1.1%20-%20PS.docx#_Toc456799581)

[Figura 11 - Exemplo de um validador 23](#_Toc456799582)

[Figura 12- Excerto de código de Pds16HighlightingConfiguration 24](#_Toc456799583)

[Figura 13 - Excerto de código de Pds16TokenAtributeIdMapper 25](#_Toc456799584)

[Figura 14 - Código da classe AbstractPds16asmUiModule 25](#_Toc456799585)

[Figura 15 - Excerto de código de Pds16asmOutlineTreeProvider 26](file:///C:\Users\tiago\Desktop\Relatorio-Final-v1.1%20-%20PS.docx#_Toc456799586)

[Figura 16 - Excerto de código de Pds16asmLabelProvider 27](file:///C:\Users\tiago\Desktop\Relatorio-Final-v1.1%20-%20PS.docx#_Toc456799587)

[Figura 17 - Excerto de código da classe Pds16asmGenerator 28](#_Toc456799588)

Lista de Tabelas

[Tabela 1 - Sintaxe das instruções PDS16. 9](#_Toc456799589)

[Tabela 2 - Palavras-chave da sintaxe PDS16. 9](#_Toc456799590)

[Tabela 3 - Elementos da sintaxe gramatical Xtext 20](#_Toc456799591)

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 elaboração do algoritmo para a sua solução, o programador começa a implementar o programa usando uma dada linguagem, obtendo-se assim um ou vários ficheiros fonte. De seguida, estes são traduzidos para a linguagem entendida pela máquina recorrendo a um compilador ou *assembler*, que primeiramente verificam as regras sintáticas da linguagem e de seguida geram um ficheiro objeto correspondente a cada ficheiro fonte. O *linker* efetua a ligação entre os diversos ficheiros objeto que compõem o programa e as bibliotecas utilizadas, que correspondem a ficheiros partilháveis que podem conter código, dados e recursos em qualquer combinação. Deste último processo resulta um ficheiro com a descrição do algoritmo codificado pelos programadores em linguagem máquina localizável em 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 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*, integração com ferramentas externas 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 suportam 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”. Esta arquitetura a 16 *bits* adota a mesma filosofia das máquinas do tipo *Reduced Instruction Set Computer* (RISC), disponibilizando o seu modelo de programação ao programador 8 registos de 16 *bits* e cerca de 40 instruções distintas, organizadas em três classes: 6 instruções para controlo do fluxo de execução, 18 instruções de processamento de dados e 12 instruções de transferência de dados. O espaço de memória útil, que é partilhado não só para o armazenamento do código e dos dados dos programas mas também para a integração com periféricos, é endereçável ao *byte* e tem uma dimensão total de 64 kB.

Atualmente, o desenvolvimento de programas para esta arquitetura pode ser feito utilizando a própria linguagem máquina ou *assembly*. A tradução do código *assembly* para linguagem máquina é realizada recorrendo à aplicação dasm [7], que consiste num *assembler* de linha de comandos que apenas pode ser executado em sistemas compatíveis com o sistema operativo Windows da Microsoft. Desta forma, o ciclo de geração de um programa passa por codificá-lo em linguagem *assembly* utilizando um editor de texto 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 um *plug-in* para suportar o desenvolvimento de programas para o processador PDS16 usando a sua linguagem *assembly*. Este é essencialmente um editor de texto que integra as seguintes ferramentas e funcionalidades:

* Verificação da semântica e sintaxe em tempo de escrita de código, de modo a que o programador possa ser alertado para eventuais erros na utilização da linguagem mais cedo e dessa forma otimizar a sua produtividade;
* *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*, trata-se de uma janela onde são assinalados pontos importantes do código, como *labels* e algumas diretivas, para que o programador consiga navegar rapidamente entre 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, atendendo à sua maior utilização na produção de programas e aplicações no domínio dos sistemas embebidos [8], onde se insere a utilização da arquitetura PDS16 no ISEL, bem como pelo facto de esta plataforma ter vindo a ser utilizada pelos 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, aquando da frequência das unidades curriculares de programação dos primeiros semestres.

Esta ferramenta foidesenvolvida sob a forma de um *plug-in* para a arquitetura PDS16 utilizado a *framework* Xtext  [9], que é uma *framework* genérica para o desenvolvimento de linguagens específicas de domínio (*DSL*). Para além da sua grande atualidade, a *framework* Xtext  apresenta ainda a grande vantagem de, com base numa mesma descrição de uma DSL, permitir gerar *plug-ins* para outras plataformas*.* Assim, partindo como base deste nosso trabalho, é possível criar *plug-ins* para a plataforma IntelliJ e para os vários *browsers* como Google Chrome, 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 dada uma visão pormenorizada sobre a arquitetura do processador Pds16, a sua linguagem especifica de domínio e também uma visão geral acerca do assemblador DASM (assemblador da linguagem Assembly Pds16);
* Capitulo 3 – é dada uma visão geral sobre a *framework* Xtext, e explicado com base em exemplos do nosso projeto o processo de criação de um *plug-in* e eventuais ferramentas;
* Capitulo 4 – é feito um resumo do que poderá ser melhorado de futuro neste projeto, e também sugestões de continuação de desenvolvimento da ferramenta.

1. Arquitetura PDS16

O PDS16 trata-se de um processador a 16 *bits* que adota o modelo de *Von-Neumann*, ou seja, que utiliza o mesmo espaço de memória tanto para código como para dados. Segue a filosofia de RISC, onde um conjunto regular de poucas instruções de tamanho fixo oferecem uma grande performance no que toca ao tempo de execução do código (ciclos máquina). Este processador apresenta as seguintes características [6]:

* Arquitetura LOAD/STORE baseada no modelo de *Von Neumman*;
* ISA, instruções de tamanho fixo que ocupam uma única palavra de memória;
* Banco de registos (*Register File*) a 16 *bits* – onde os registos de R0 a R5 se encontram em dois bancos de registos diferentes, e R6 e R7 são registos de uso específico;
* Possibilidade de acesso à palavra (*word*) e ao *byte*;
* Suporte a rotinas;
* Interrupção externa.
  1. Modelo de programação

Nesta arquitetura, para interação do programador com a máquina e elaboração de programas, são disponibilizados:

* Set de instruções - transferência de dados, processamento de dados e controlo de execução;
* Conjunto de registos – que podem ser por exemplo para armazenamento temporário de dados para execução de instruções;
* Espaço de memória partilhado – este pode ser utilizado para o armazenamento de dados ou interação com periféricos
* Mecanismo de interrupção – útil para a notificação de eventos externos.
  1. Registos

Esta arquitetura inclui dois bancos de registos que visam suportar, de uma forma eficiente, o funcionamento no modo normal e atendimento de interrupções. Assim, ao ser atendida uma interrupção, é alterado o banco de registos em utilização de modo a salvaguardar o estado do programa em modo normal. Ao terminar a rotina de interrupção o banco de registos é novamente trocado, continuando com o normal funcionamento do programa.

O banco de registos do modo normal disponibiliza ao programador 8 registos de 16 *bits*, ser utilizados não só para guardar os operandos das instruções *assembly* que compõem os programas, mas também os resultados produzidos pela sua execução. 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 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 *Linker Register*.

Os registos R6 e R7 são outros dois registos especiais do processador. O registo R7 corresponde ao *Program Counter* (PC), guardando o endereço de memória da próxima instrução a ser executada. O registo R6 guarda o estado das 6 *flags* do processador, motivo pelo qual também é denominado de *Processor Staus Word* (PSW):

* Zero (Z): Caso se encontre ativa (i.g. valor lógico 1) significa que o código da operação realizada pela ALU corresponde ao valor zero;
* *Carry/Borrow* (CY): Esta *flag* apresentanda o valor lógico 1 quando a instrução anterior gerou *carry/borrow*, baseado no teste de menor/maior ou igual, considerando que os operandos da ALU pertencem ao conjunto de números naturais;
* *Greater or Equal* (GE): Esta *flag* apresentanda o valor lógico 1 quando ao realizar uma subtração o diminuendo é maior ou igual ao diminuidor, considerando que os operandos da ALU pertencem ao conjunto de números relativos;
* *Parity* (P): Esta *flag* fica ativa quando o valor resultante de uma operação lógica ou aritmética contenha um número de *bits* com valor lógico 1 em quantidade ímpar.
* *Interrupt Enable* (IE): Quando esta *flag* possui valor lógico 1 o mecanismo de interrupção está ativa, podendo a normal execução de um programa ser interrompida por uma ação externa, o que consiste num de exceção.
* *Bank Selct* (BS): 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 lógico é 0 enquanto se o seu valor lógico for 1 é o banco de registos do modo interrupção que está em utilização.

É ainda de mencionar que algumas destas *flags* são utilizados para controlo de fluxo de código, ou seja, é analisado os operandos e/ou resultado das operações da ALU e dependendo tomando com base nesta analise a decisão da próxima instrução a ser executada. São estas as *flags* Z, CY, GE e P.

O posicionamento das diferentes *flags* nos 16 *bits* que compõe este registo é ilustrado na Figura 2.

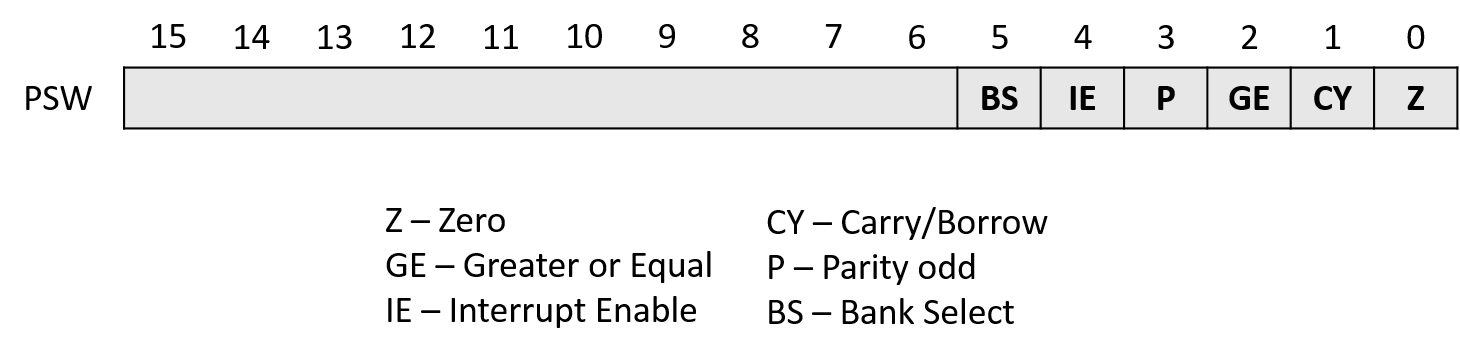


Figura 2 –Estrutura interna do registo PSW.

Apesar de existirem dois bancos de registos, não existe uma duplicação de todos os registos, Figura 3.

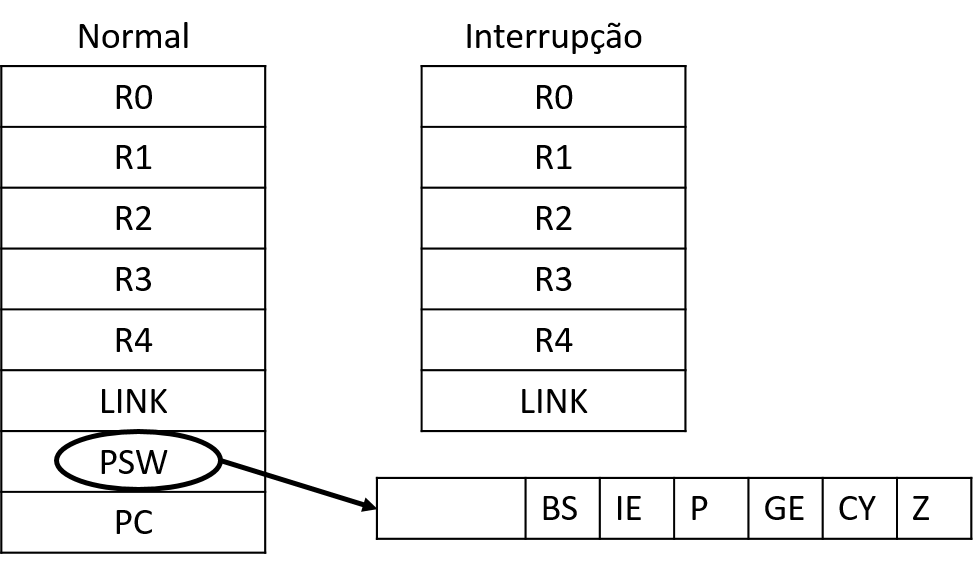


Figura 3 - Bancos de Registos PDS16

O banco de interrupção não é acessível em modo normal, sendo reservado para o tratamento da exceção de interrupção. Neste cenário, com o BS ativo, o banco de registos em utilização será o banco de interrupção, sendo que este não contém uma duplicação de todos os registos disponíveis no banco de registos normal. Neste banco, existem apenas 6 registos, sendo que o registo LINK é reservado para a cópia do registo PC do banco normal, sendo que poderá ser utilizado, mas tendo em conta que o seu valor será colocado no registo PC do banco normal, no final da rotina. O registo R0 também tem um comportamento semelhante ao anterior, mas neste caso salvaguarda o valor do registo PSW do banco de registos normal, sendo que deverá ser preservado no final da rotina. Não existe a necessidade de os registos PSW e PC serem duplicados uma vez que o seu valor já se encontra salvaguardado pelo mecanismo acima mencionado, e não faria sentido estes dois registo serem duplicados, pois tratam-se de registos de uso específico. Os restantes registos de R1 a R4, são registos de uso normal, não tendo qualquer comportamento excecional.

* 1. Conjunto de instruções

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

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

Tabela 1 - Sintaxe das instruções PDS16.

|  |  |
| --- | --- |
| Palavras-chave | 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 Unidade Lógica e Aritmética (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 do PDS16, 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 (*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 ou 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 do PDS16 (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 do PDS16, 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 2 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 1 instrução de salto incondicional e 4 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 2 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 2 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

Como este processador segue a arquitetura de *Von-Neumann*, é usado apenas uma memória para código e dados de 32K\*16 em que os *bytes* são organizados seguindo o esquema *big-endian* (bytes por ordem crescente do seu "peso numérico" em endereços sucessivos da memória). Este espaço de memória é partilhado não só entre código e dados, mas também com o acesso a periféricos, ou seja, definindo uma zona de memória em que os acessos à mesma, irão ser refletidos numa ação no periférico, e não com o intuito de manipular o estado da memória.

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

* 1. Exceções

Um mecanismo de exceção trata-se de manipular eventos inesperados que ocorrem durante a execução de um programa que têm impacto, direta ou indiretamente nessa mesma execução. Neste caso, o processador suporta 2 mecanismos de exceção: Interrupção Externa e *Hard Reset*.

O mecanismo de interrupção externa é disponibilizado através de uma entrada IRQ (*Interrupt Request*), e consiste na verificação do pino IRQ ao fim de cada execução de uma instrução, e caso este esteja ativo (*active-low*) é gerada uma chamada a uma rotina ISR (*Interrupt Service Routine*) que executará a ação pretendida por quem interrompeu. Esta verificação só é efetuada caso a *flag* “IE” se encontre ativa, a qual deve ser ativa pelo utilizador caso pretenda atender interrupções externas. Uma das dificuldades que essa interrupção trás, depois de executar código ISR, é voltar a colocar os registos nos estados originais e retornar o programa no estado inicial antes da interrupção (registo PC). Visto que a arquitetura do PDS16 não suporta o uso de um *Stack*, o modo de resolução implementado pelo sistema, trata-se de utilizar um segundo banco de registos (através da *flag* “BS”). Aquando de uma interrupção, o CPU realiza as seguintes ações em simultâneo [10]:

* Copio o valor do registo PSW para o registo R0 do banco de interrupção;
* Coloca o valor lógico 1 no *bit* BS do registo PSW, tornando desta torna ativo o banco de interrupção;
* Coloca o valor lógico 0 no *bit* IE do registo PSW, inibindo assim as interrupções;
* Copia o valor do registo PC para o registo LINK do banco de interrupção;
* Coloca o valor 2 no registo PC, vetorizando desta forma o processamento para o ponto de entrada da interrupção.

Por outro lado o mecanismo de *Hard Reset* é assinalado manualmente pelo utilizador através de um botão durante a execução de um programa. Como o próprio nome indica, este mecanismo leva a que o processador volte ao estado inicial, interrompendo qualquer execução existente. Isto é garantido pois a implementação do mecanismo passa por carregar o valor 0 para o registo PSW, (levando a que seja selecionado o banco de registos 0 e ao não atendimento de interrupções externas, *falg* “IE” a 0) e também para o PC (levando a que seja passada a execução novamente para a primeira posição de memória, *boot*).

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

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

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

* + 1. Escrita de programas

Quando o utilizador escreve um ficheiro deve ter em conta que o assemblador DASM lê o ficheiro por ordem *top down*, e que cada símbolo contém o endereço da instrução depois do mesmo. Cada instrução pode ser dividida em 4 campos ordenados, seguindo a seguinte forma:

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

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

Para além das instruções assembly PDS16, o assemblador DASM reconhece e processa um outro conjunto de comandos [11]. Estes comandos visam não só facilitar a organização em memória do código e dos dados dos programas, mas também a utilização de símbolos para representação de valores, e.g. endereços e constantes.

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

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

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

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

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

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

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

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 [12] (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. [13] 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. É possível obter um editor utilizando uma plataforma, como é o caso do ambiente de desenvolvimento Eclipse [3] ou do IntelliJ IDEA [4], e também através de um *browser*, por exemplo o *Chrome*.

Para a elaboração deste projeto foi utilizado como recurso o livro “*Implementing Domain-Specific Languages with Xtext and Xtend*” [14], que serve como suporte aos programadores que pretendem criar uma DSL utilizando a *framework* Xtext, oferecendo uma abordagem passo a passo, acompanhada de exemplos simples, das funcionalidades a implementar.

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. [15] 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*.

O ponto principal de um projeto na *framework* Xtext é a sua gramática (ficheiro com extensão *“.xtext”*), onde é definida toda a sintaxe da linguagem (*tokens*, expressões regulares, etc.). Através da gramática 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* [16], 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 [17]), 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. [13]

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. [13] Para a geração do código, são utilizadas duas gramáticas ANTLR [18], geradas pelo Xtext: uma para produção do "*parser*" onde uma *Abstract Syntax Tree* [19](AST)é produzida, outra que é utilizada para o processamento dos eventos do editor do *Eclipse*. A sintaxe do ficheiro de gramatica Xtext é igual à do ANTLR, à exceção de que para a verificação semântica, o ANTLR contem 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. [20] Estas dependências são resolvidas utilizando a *framework* Google Guice [21], 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. [13]

* 1. Gramática

Com o estudo das instruções do PDS16 e das diretivas (e mnemónicas) do DASM (através a documentação [11], [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.

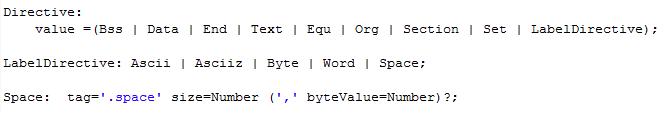
Assim a gramática é definida através de regras que podem referenciar outras regras ou palavras-chave. Por cada regra definida é gerada uma classe modelo, com métodos e atributos dependendo da definição da regra. Na geração das classes modelo, será adicionada a dependência entre elas como por exemplo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

Figura 4 – 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.

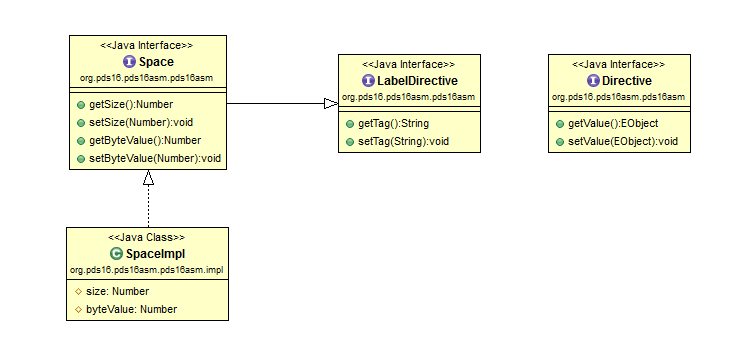
Essa dependência é tratada pelo Xtext gerando automaticamente classes em Java quando o *Modeling Workflow Engine 2* (MWE2 [16]) é executado, resolvendo essa dependência pela extensão entre classes, criando assim uma hierarquia entre as regras de uma DSL (Figura 5).

Figura 5 - 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 6 - Código exemplo da definição das regras

Tendo como exemplo a nossa implementação da gramática, “*Statement*” é uma regra que na sua definição apenas contem 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 7 - 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 8 - 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 9 - Excerto da classe PDS16asmValueConcerter

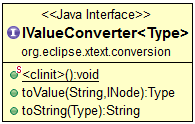
Como presente na figura, a anotação “*@ValueConverter(rule=“HEX”)*”, indica que o método por ela anotado, retornará um conversor para o tipo de retorno (neste caso para *Integer*) da regra com o nome *“HEX”*, sendo que se trata de uma instância da classe “*HEXValueConverter*”, que por sua vez terá de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

Figura 10 - 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.

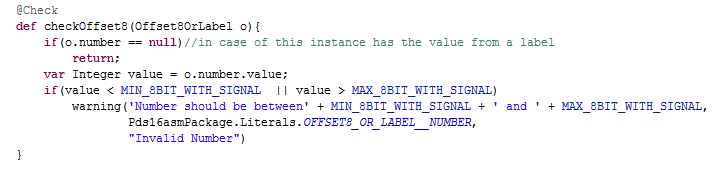
No caso do nosso no projeto verificamos os limites dos números presentes nas regras, por exemplo, uma regra que contenha um número representativo de um *offset* a8 *bits* com sinal, será verificado se o número inserido pelo utilizador se insere nos limites permitidos, emitindo um *warning* caso contrário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

Figura 11 - 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 12- 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 13 - 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 14.

Figura 14 - Código da classe AbstractPds16asmUiModule

Esta classe encontra-se no *package* responsável pela parte gráfica do projeto, neste caso do Eclipse, sendo que que nesta classe devem ser registadas eventuais alterações às configurações por definição por parte da *framework*.

* + 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).

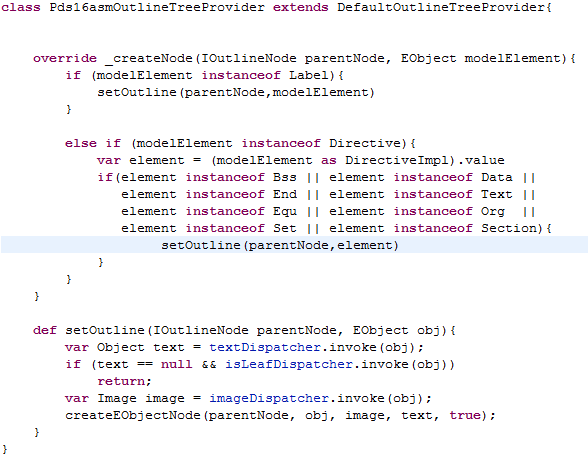
Para definir os elementos que queremos que estejam presentes no *outline* temos que filtra-los. Para esse efeito usamos a classe gerada *Pds16asmOutlineTreeProvider* que estende de *DefaultOutlineTreeProvider* onde é feito *override* ao método “\_*createNode*” como se pode verificar na Figura 15.

Figura 15 - 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 à 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*”).

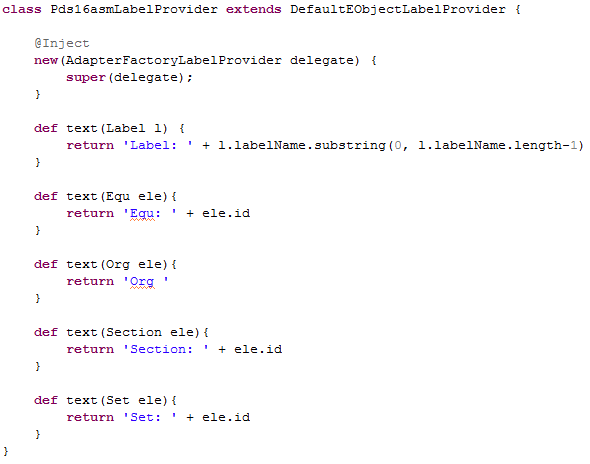
No nosso caso quisemos especificar com mais detalhe o nome que exibido em cada elemento do *Outline* e para isso usamos a classe *Pds16asmLabelProvider* (gerada pela framework) que estende de *DefaultEObjectLabelProvider*, onde para cada tipo de regra da gramatica suportada pelo nosso *outline* é calculado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

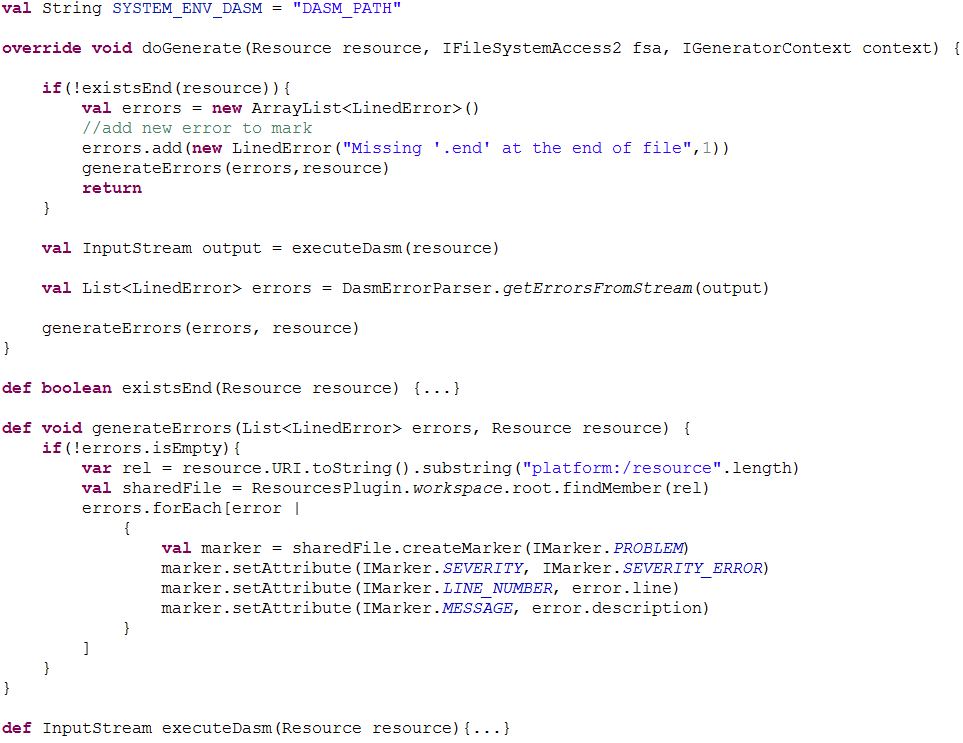
Figura 16 - Excerto de código de Pds16asmLabelProvider

Cada método “*text*” definido presente na Figura 16 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 aparecera na janela de *outline*. Estes métodos são evocados através do objeto “*textDispatcher*” presente no método da Figura 15.

* + 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 17 - 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.

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

* + 1. Configuração do *plug-in*

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

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

De seguida foi criado um projeto do tipo *Update Site* para conseguirmos criar e disponibilizar o *plug-in* de modo a poder ser instalado remotamente, alojando-o numa página web. Neste projeto tivemos apenas de referenciar o *feature project* criado anteriormente e efetuar a operação *build all*, que gera todos os ficheiros necessários para a instalação do mesmo.

No processo de *deploy* tivemos em conta o controlo de versões do *plug-in*, podendo este ser atualizado pelo utilizador quando for lançado uma nova versão do software.

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

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 <http://tiagojvo.github.io/PDS16inEclipse/> 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. Arquitetura de Computadores – Textos de apoio às aulas teóricas (págs. 13-1 – 13-27),” Lisboa, 2011. |
| [7] | J. Paraíso, “Desenvolvimento de Aplicações. Arquitetura de Computadores – Textos de apoio às aulas teóricas (págs. 15-2 – 15-5),” Lisboa, 2011. |
| [8] | C. Ajluni, “Eclipse Takes a Stand for Embedded Systems Developers,” [Online]. Available: http://www.embeddedintel.com/search\_results.php?article=142. [Acedido em 30 03 2016]. |
| [9] | “Xtext 2.5 Documentation, Eclipse Foundation,” 2013. [Online]. Available: http://www.eclipse.org/Xtext/documentation/2.5.0/Xtext%20Documentation.pdf. [Acedido em 05 02 2016]. |
| [10] | J. Paraiso, “Interrupções. Arquitetura de Computadores – Textos de apoio às aulas teóricas (págs. 19-2 – 19-8),” 2011. |
| [11] | J. Paraíso, “QuickRef\_V2,” [Online]. Available: http://pwp.net.ipl.pt/cc.isel/ezeq/arquitetura/sistemas\_didaticos/pds16/hardware/QuickRef\_V2.pdf. |
| [12] | T. E. Foundation, “Eclipse Modeling Framework (EMF),” The Eclipse Foundation, [Online]. Available: https://eclipse.org/modeling/emf/. [Acedido em 13 7 2016]. |
| [13] | Model-driven Pretty Printer for Xtext, Prague, 2012. |
| [14] | L. Bettini, Implementing Domain-Specific, Packt Publishing, 2013. |
| [15] | “Xtend Documentation,” [Online]. Available: https://www.eclipse.org/xtend/documentation/index.html. [Acedido em 13 7 2016]. |
| [16] | “MWE2 Documentation,” [Online]. Available: https://eclipse.org/Xtext/documentation/306\_mwe2.html. [Acedido em 10 6 2016]. |
| [17] | Wikipedia, “Plain Old Java Object,” [Online]. Available: https://en.wikipedia.org/wiki/Plain\_Old\_Java\_Object. [Acedido em 15 7 2016]. |
| [18] | ANTLR / Terence Parr, “About The ANTLR Parser Generator,” 2014. [Online]. Available: http://www.antlr.org/about.html. [Acedido em 15 7 2016]. |
| [19] | Wikipedia, “Abstract syntax tree,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Abstract\_syntax\_tree. [Acedido em 19 7 2016]. |
| [20] | S. Hungerecker, SALTXT: An Xtext-based Extendable Temporal Logic, Lübeck, 2014. |
| [21] | Google, “Google Guice,” [Online]. Available: https://github.com/google/guice. [Acedido em 15 7 2016]. |
| [22] | “PDS16inEclipse,” [Online]. Available: http://tiagojvo.github.io/PDS16inEclipse/. |

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

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

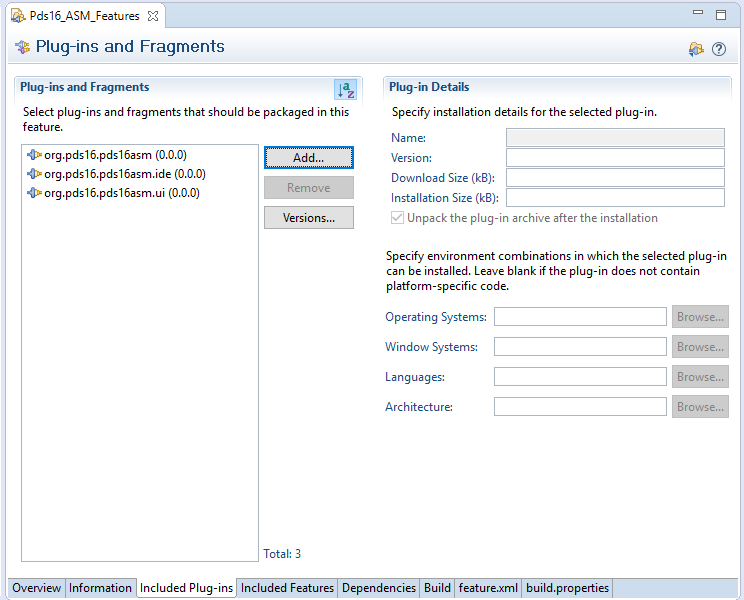
Como o software têm que ser acoplado com um IDE neste caso o Eclipse, criámos um *plug-in* que adicionará as novas funcionalidades ao IDE. Este não só contém o software desenvolvido como também as dependências do mesmo. No processo de *deploy* tivemos em conta o controlo de versões do *plug-in*, podendo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

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

1. Criar um “Feature Project” no eclipse.



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 três 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 Ecplise através do *link* do website alojado no *Github*.

A.2 - Instalação do Plug-in

Para fazer o correto uso do editor de texto é necessário instalar o *plug-in* e definir uma variável de ambiente com a path do assemblador DASM (<http://pwp.net.ipl.pt/cc.isel/ezeq/arquitetura/sistemas_didaticos/pds16/ferramentas/dasm.exe>)

O *plug-in* pode ser instalado no IDE Ecplise de duas maneiras, fazendo download do ficheiro ZIP ou instalar usando este link: <http://tiagojvo.github.io/PDS16inEclipse/>.

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

1. Definir uma variável de ambiente com o nome "DASM\_PATH" com a respetiva *path* do assemblador, reiniciando de seguida o Windows para que esta fique disponível.
2. Efetuar os seguintes passos no IDE Eclipse:
   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)