

**PDS16inEcplise**

|  |  |
| --- | --- |
|  | André Ramanlal |
|  | Tiago Oliveira |

|  |  |
| --- | --- |
| Orientadores | Tiago Miguel Braga da Silva Dias |
|  | Pedro Miguel Fernandes Sampaio |

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

Abril de 2016

**Instituto Superior de Engenharia de Lisboa**  
Licenciatura em Engenharia Informática e de Computadores

**PDS16inEcplise**

|  |  |
| --- | --- |
| 39204 | André Akshei Manoje Ramanlal |
| 40653 | Tiago José Vital Oliveira |
|  |  |

|  |  |
| --- | --- |
| Orientadores: | Tiago Miguel Braga da Silva Dias |
|  | Pedro Miguel Fernandes Sampaio |

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

Abril de 2016

# Resumo

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

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

# Índice

[Resumo v](#_Toc450399948)

[Índice vii](#_Toc450399949)

[Lista de Figuras ix](#_Toc450399950)

[Lista de Tabelas xi](#_Toc450399951)

[1 Introdução 1](#_Toc450399952)

[1.1 Enquadramento 1](#_Toc450399953)

[1.2 Motivação 3](#_Toc450399954)

[1.3 Objetivos 4](#_Toc450399955)

[2. PDS16 DSL – Linguagem de Domínio Especifico 5](#_Toc450399956)

[2.1 Introdução 5](#_Toc450399957)

[2.2 Modelo de programação (ISA) 6](#_Toc450399958)

[2.2.1 Mapa de memória 7](#_Toc450399959)

[2.2.2 Registos 7](#_Toc450399960)

[2.2.3 Instruções 8](#_Toc450399961)

[2.2.3.1 Processamento de Dados 9](#_Toc450399962)

[2.2.3.2 Controlo de Fluxo de Execução 10](#_Toc450399963)

[2.3 Assembler dasm 11](#_Toc450399964)

[2.3.1 Introdução 11](#_Toc450399965)

[2.3.2 Reserva e iniciação da memória 11](#_Toc450399966)

[2.3.3 Diretivas 11](#_Toc450399967)

[2.1 Diretivas 12](#_Toc450399968)

[3. Framework Xtext 14](#_Toc450399969)

[3.1 Introdução 14](#_Toc450399970)

[3.2 Arquitetura 15](#_Toc450399971)

[3.3 A gramática 16](#_Toc450399972)

[3.3.1 Regras da gramática 17](#_Toc450399973)

[3.3.2 Definição dos elementos do analisador de regras 18](#_Toc450399974)

[3.4 Integração com a plataforma Ecplise 19](#_Toc450399975)

[3.4.1 Configuração do plug-in 19](#_Toc450399977)

[3.4.2 Syntax Highlight 19](#_Toc450399978)

[3.4.2.1 Estilo do texto 19](#_Toc450399979)

[3.4.2.2 Highlight léxico 19](#_Toc450399980)

[3.4.2.3 Highligt semântico 19](#_Toc450399981)

[3.4.3 Gerador 19](#_Toc450399982)

[4. Progresso do Projeto 20](#_Toc450399983)

[Referências 23](#_Toc450399984)

# Lista de Figuras

[Figura 1 – Exemplo de um ciclo de desenvolvimento de um programa/aplicação. [1] 1](#_Toc450305764)

[Figura 2 - Exemplo de código de instruções 13](#_Toc450305765)

[Figura 3 - Classes geradas pela framework 13](#_Toc450305766)

[Figura 4 - Código exemplo da definição das regras 15](#_Toc450305767)

[Figura 5 - Código exemplo da definição regras terminais 15](file:///D:\Repositorio\PDS16ASM\Relatorio%20Beta\Relatorio%20Beta.docx#_Toc450305768)

[Figura 6 - Código da classe Pds16asmRuntimeModule 16](#_Toc450305769)

[Figura 7 - Exemplo de um validador 16](#_Toc450305770)

# Lista de Tabelas

[Tabela 1 - Diagrama de Gantt relativo à previsão da execução do trabalho. 19](#_Toc449994425)

# Introdução

## Enquadramento

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



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

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

Os Ambientes Integrados de Desenvolvimento (*IDEs*) são hoje em dia um enorme apoio no desenvolvimento destes programas, uma vez que não só disponibilizam diversas ferramentas para apoio à produção do código, e.g. um editor de texto, a geração automática de código ou o *refactoring*, como ainda possibilitam a interação com outras ferramentas e aplicações, como é o caso dos compiladores, *linkers*, *debuggers*, controladores de versão, etc.

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

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

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

## Motivação

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

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

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

## Objetivos

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

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

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

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

# PDS16 DSL – Linguagem de Domínio Especifico

## Introdução

O PDS16 trata-se de um processador a 16 bits que se segue a arquitetura de Von-Neumann que usa o mesmo espaço de memória tanto para código como para os dados. Este processador apresenta as seguintes características [4]:

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

Para além destas principais caraterísticas do processador, este também têm um mecanismo de interrupção que consiste na verificação de um pino ao fim de cada execução de uma instrução, e caso este esteja ativo é gerada uma chamada a uma rotina ISR (Interrupt Service Routine) que executará a ação pretendida por quem interrompeu. Mas um dos problemas que essa interrupção trás depois de executar código ISR é voltar a meter os registos nos estados originais e retornar o programa no estado incial antes da interrupção. Para que a interrupção corra bem é necessário preservar o estado de execução neste caso salvaguardar o valor corrente do registo PC no do registo Link. Em relação aos restantes registos ou fica pela responsabilidade da ISR usar uma estrutura de dados ou existe uma duplicação de vários registos do CPU e que são comutados no momento da interrupção, pois a arquitetura do PDS16 não suporta o stack.

A tabela seguinte, presente na documentação [8] , mostra a sintaxe das instruções que o processador PDS16 suporta, estando elas divididas em secções como cinco secções: Load, Store, Aritmétrica, Lógica e Jump.

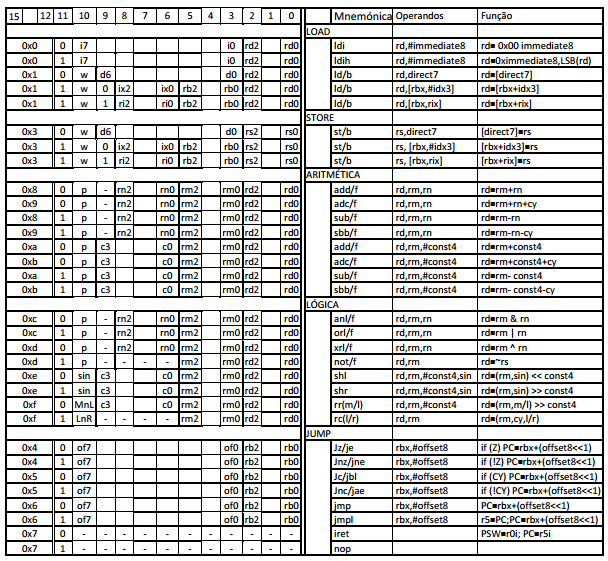


Figura 2 - Tabela do código e sintaxe das Instruções do PDS16

## Modelo de programação (ISA)

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

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

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

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

### Mapa de memória

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

### Registos

O Register file da estrutura do PDS16 é composto por 8 registos de R0 a R7 de 16 bits cada um. Os registos de R0 até ao R4 inclusive são registo usados para a manipulação de dados temporários sem recorrer a memória para armazenar. O registo R6, o PC guarda o endereço de memorio da instrução exatamente a seguir à que está a ser executada. Este registo é útil pois certas operações que usam o offset como o jump que somando ao endereço da instrução corrente resulta um salto para determinada instrução. O registo R5, o LINK, foi criado com o intuito de salvaguardar o valor corrente do registo R7, o PC, num caso de um salto de endereço do código que esta a ser corrido para outro endereço de código como é o caso de um jump ou uma interrupção. Com o registo LINK fica assegurado o retorno ao endereço exatamente a seguir ao da evocação da rotina. Por fim o registo R6, o PSW, serve de controlo de flags, pois cada bit dos 16 que o registo suporta esta determinado a uma flag. Conforme a seguinte imagem nem todos os bits estão ocupados com flags, estando apenas 6 bits: o Zero, Greater or Equal, Interrupt Enable, Carry/Borrow, Parity odd e o Bank Select.

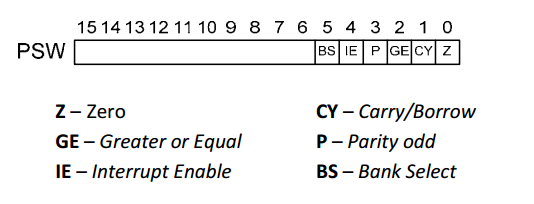


Figura 3 – Flags do registo PSW

### Instruções

Como referido acima as instruções dividem se em três tipos: acesso a memória de dados, processamento de dados e controlo fluxo de execução, apresentadas nos pontos seguintes.

### Acesso a memória de dados

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

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

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

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

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

ld rx, label\_name

Esta instrução coloca no registo “rx” o contudo da posição de memória indicado pelo símbolo “label\_name. Neste caso o acesso é direto pois não são efetuados quaisquer cálculos para definir a posição de memória requerida.

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

ldb rx, [ry, #const]

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

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

ldb rx, [ry, rz]

### Processamento de Dados

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

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

Exemplo de operação envolvendo 2 registos como operandos fonte:

add rx, ry, rz

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

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

add rx, ry, #const

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

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

ldi rx, #const

ldih ry, #const

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

### Controlo de Fluxo de Execução

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

Exemplo de instrução de salto incondicional:

jmp LAB1

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

Exemplo de instrução salto condicional:

jz rx,#const

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

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

## Assembler dasm

### Introdução

### Reserva e iniciação da memória

### Diretivas

Para além das instruções assembly PDS16, o *assembler* dasm reconhece e processa um outro conjunto de comandos [8] que visam não só facilitar a organização do código e dos dados dos programas em memória, mas também a utilização de símbolos para representação de endereços e constantes.

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

* “.DATA” – zona de dados inicializados;
* “.BSS” – zona de dados não inicializados;
* “.TEXT” –zona de código.

Para além destas, permite ainda que o programador defina o seu próprio tipo de secção através da expressão “.SECTION section\_name”, substituindo section\_name pelo nome da secção desejada.

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

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

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

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

# Framework Xtext

## Introdução

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

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

Decidimos utiliza-la para a realização de um plug-in para uma linguagem de *assembly* PDS16, não havendo nenhum editor de texto para a mesma.

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

## Arquitetura

A framework Xtext esta desenvolvida com base no ambiente de desenvolvimento Ecplise e tem como base a linguagem de programação Java. Para desenvolver uma linguagem priemeiro temos de definir a sintaxe da linguagem. E como já referido a linguagem é definida através de regras que podem referenciar outras regras ou palavras-chaves. Por cada regra definida é criada uma clases com os métodos de *get* e *set* conforme a definição da regra, mas qualquer regra poderá depender de outra regra. Para isso a geração automática das classes cria também a dependência das classes com as outras. Como por exemplo nas seguintes regras da figura 2:

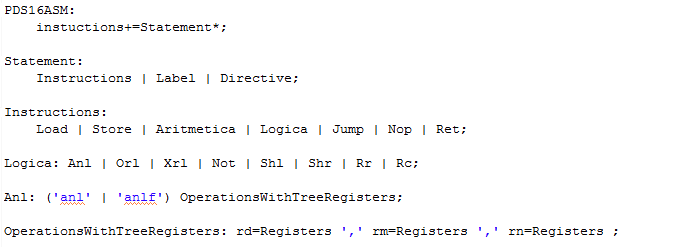


Figura 4 - Exemplo de código de instruções

Na figura 2 podemos ver que a regra *OperationsWithTreeRegisters* depende de *Anl* e que por sua vez depende de *Logica* que esta depende *Instructions* e assim consecutivamente ate chegar a regra *PDS16ASM*.

Essa dependência toda é tratada pelo Xtext gerando automaticamente classes em Java quando o MWE2 *workflow* é corrido, resolvendo essa dependência pela extensão a classe da que depende criando assim uma hierarquia entre as regras de uma DSL, como o exemplo da figura seguinte.

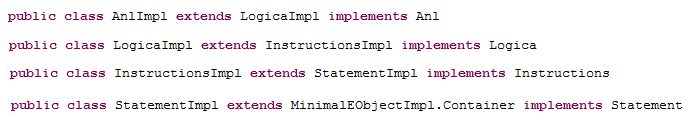


Figura 5 - Classes geradas pela framework

Esta geração automática de um objeto que representa uma determinada regra não é feito ao acaso. Com um objeto desses é possível obter em runtime os valores atribuídos a uma determinada etiqueta da regra permitindo assim implementar as outras funcionalidades que a framework oferece como o *highlighting*, validação, *parser*, compilador entre outros.

Certas ferramentas da biblioteca Xtext como o *parser* ou o validador podem ser implementados, mas numa linguagem específica criada com base na linguagem Java, o Xtend. A linguagem de programação Xtend está totalmente integrada com a linguagem Java obtendo assim todos os recursos e suporto que o Java têm como as bibliotecas, e oferendo outras funcionalidades como o *type inference,* métodos de extensão*,* expressões lambdase *multi-line template expressions*. Todos os aspetos da DSL implementado em Xtext podem ser implementados em Xtend em vez do Java pois é mais fácil de usar e permite escrever código de leitura legível.

Após definir a gramática no ficheiro com a extensão “.xtext”, é necessário compilar e neste caso corrermos o ficheiro Modeling Workflow Engine 2, (MWE2), como fosse o *entry point* da aplicação que gerará todas as classes necessárias para podermos correr depois o projeto como forma de um plug-in gerado.

## A gramática

Para definir uma linguagem de programação, temos de estudar a sua gramatica e ter em atenção as possíveis formas de escrever uma determinada regra da sintaxe. Como o nosso projeto visa criar um editor de texto para o processador PDS16, nesse sentido estudamos o *assembly* do PDS16 através a documentação [8], [4] e [5].

A gramatica no xtext é definida por *parser rules* ou por *key words. Parser Rules* são regras que definem uma sequência de outras regras conjugando com palavras-cahves. Mas esta regra é manipulada como um tipo para a criação de EObjects que formula a semântica do modelo.

### Regras da gramática

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



Figura 6 - Código exemplo da definição das regras

*Statement* é uma regra que na sua definição é uma das referências para outra regra. Neste caso na regra “*Label”* podemos ver que a sua definição já contem palavras-chaves como “:” e um identificador “*labelName*” que é o tipo ID considerado um terminal. “*Ret*” e “*Nop”* são apenas é constituídas por palavras-chave, não dependendo de nenhuma outra regra. A regra “*Jump*” que é mais complexa pode ser definida por uma destas palavras-chaves, seguida pela regra “*OperationWithOffset*”.

Regra terminal (*Terminal Rule*) trata-se de um tipo de regra que é definida por uma sequência de caracteres (*token)* também denominada por *token rule* ou *lexer rule*. Um terminal pode retornar um tipo que por definição se trata de uma *String: ecore::EString*. Mas é possível converter o tipo de retorno para um tipo especifico desde que seja uma instancia de *ecore::EDataType*. Para isso é necessário criar uma classe que implemente a interface “*IValueConverter*” e criar o respetivo converter de *String* para o tipo pretendido.

Figura 7 - Código exemplo da definição regras terminais

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



Figura 8 - Código da classe Pds16asmRuntimeModule

Ao redefinir o método “*bindIValueConverter*” estamos a associar o nosso próprio *IValeuConverterService* ao *Pds16asmRuntimeModule*, neste caso *Pds16asmValueConverter*. Este implementa a interface *IValeuConverterService*, onde através da anotação de método “*@ValueConverter(rule=”RULE\_NAME”)*”, se definem todos os *IValeuConverter’s* suportados pela classe.

### Definição dos elementos do analisador de regras

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

Figura 9 - Exemplo de um validador

## Integração com a plataforma Ecplise



### Configuração do plug-in

### Syntax Highlight

### Estilo do texto

### Highlight léxico

### Highligt semântico

### Gerador

A própria *framework* disponibiliza a opção de criar um compilador, mas nesta etapa do projeto decidimos usar um compilado externo, o DASM [5]. Para isso, é feita uma chamada ao compilador externo através do “*ProcessBuilder*”, passando-lhe como parâmetro a localização do ficheiro fonte, ASM, capturando o output após a compilação. Com esse output é então analisado e determinando se o ficheiro fonte tem erros. Caso contenha erros, estes serão assinalados no ficheiro fonte com os *Markers* na respetiva linha e com a respetiva mensagem vinda do compilador.

# Progresso do Projeto

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

* **Estudo do *Assembly* PDS16:** Estudo da linguagem com base na documentação de Arquitetura de Computadores, capítulos 13 [1] e 15 [2].
* **Estudo da *Framework* Xtext:** Estudo da *framework* com base na documentação disponibilizada na Web.
* **Elaboração Proposta do Projeto:** Foi elaborada a proposta do projeto depois do estudo do *assembly* PDS16 e da *framework* que vai servir de suporte ao projeto, tendo sido realizada uma proposta de calendarização com os prazos a cumprir.
* **Implementação do ASM PDS16:** Foi definida a sintaxe gramatical da linguagem utilizando a *framework* Xtext [3], criando também validadores para certos aspetos da linguagem que ajudam ao utilizador informando os erros.
* **Gerador (Utilizando PDS16):** Para compilar o ficheiro foi invocado o compilador dasm, passando um ficheiro fonte como entrada e recebendo o resultado da compilação como saída.Esta informação é utilizada para verificação da existência de erros de compilação e, caso existam, assinalá-los no ficheiro fonte com a mensagem de erro produzida pelo *assembler*.

Face ao exposto, à data atual prevemos cumprir a calendarização inicialmente definida que se apresenta na Tabela **1**.



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

# Referências

|  |  |
| --- | --- |
| [1] | “IDE Ecplise,” [Online]. Available: http://www.eclipse.org. |
| [2] | “Intellij, IDE,” [Online]. Available: https://www.jetbrains.com/idea/. |
| [3] | O. White, “IDEs vs. Build Tools: How Eclipse, IntelliJ IDEA & NetBeans users work with Maven, Ant, SBT & Gradle,” 2014. [Online]. Available: http://zeroturnaround.com/rebellabs/ides-vs-build-tools-how-eclipse-intellij-idea-netbeans-users-work-with-maven-ant-sbt-gradle/. [Acedido em 25 03 2016]. |
| [4] | J. Paraíso, “PDS16. Arquitetura de Computadores – Textos de apoio às aulas teóricas (págs. 13-1 – 13-27),” Lisboa, 2011. |
| [5] | J. Paraíso, “Desenvolvimento de Aplicações. Arquitetura de Computadores – Textos de apoio às aulas teóricas (págs. 15-2 – 15-5),” Lisboa, 2011. |
| [6] | C. Ajluni, “Eclipse Takes a Stand for Embedded Systems Developers,” [Online]. Available: http://www.embeddedintel.com/search\_results.php?article=142. [Acedido em 30 03 2016]. |
| [7] | “Xtext 2.5 Documentation, Eclipse Foundation,” 2013. [Online]. Available: http://www.eclipse.org/Xtext/documentation/2.5.0/Xtext%20Documentation.pdf. [Acedido em 05 02 2016]. |
| [8] | J. Paraíso, “QuickRef\_V2,” [Online]. Available: http://pwp.net.ipl.pt/cc.isel/ezeq/arquitetura/sistemas\_didaticos/pds16/hardware/QuickRef\_V2.pdf. |
| [9] | T. Dias, “Elaboração de Ficheiros Executáveis,” 2013. [Online]. Available: https://adeetc.thothapp.com/classes/SE1/1314i/LI51D-LT51D-MI1D/resources/2334. [Acedido em 27 03 2016]. |

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