technische universität
dortmund

Masterarbeit

A Domain-Specific Language for the Reduction of Hardware-Dependencies during the Development of Operating Systems

Karen Bieling
April 24, 2017

Supervisor:
Prof. Dr.-Ing. Olaf Spinczyk
M. Sc. Hendrik Borghorst

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de

Contents

1 Introduction

In the history of computers the requirements for efficiency in programming and computing has increased rapidly. The first step of simplifying programming in machine code was the invention of the *Assembler Code*. The first *Assembler Code* was written by the American Computer pioneer Nathaniel Rochester in the late 1940s [14]. It allows to write human readable instructions that are directly translated into the machine opcodes. Just a short time later it was not longer sufficient to run just one single program - the operating systems were invented to manage the processes on a computer. The first operating systems were written in *Assembler Code*. When the first high-level programming languages were developed the operating systems haven been written in these languages. Nevertheless there is still, 70 years after the first *Assembler Code* has been written, need to use *Assembler Code* in operating system code, especially at the start up. The operating system also has to deal with hardware-specific properties and optimization depending on the hardware architecture.

In the context of this master thesis a language has been implemented, which allows to separate the hardware description from hardware-dependent software. Here the focus is on *Assembler Code* as an important representative of hardware-dependent code. The basic idea is to find the hardware-independent intersection of assembler instructions, add the hardware information and finally generate the hardware-dependent code instead of implementing it. Figure 1.1 shows this procedure.

1.1 Motivation

Using operating systems on various hardware platforms cause the problem, that it has to be developed for various platforms. The first and obvious fact is the usage of *Assembler Code*. The different hardware designers use diverse instruction architectures. Some use more instructions, some less and others use different names for semantically identical instructions. That is why the *Assembler Code* has to be written for each supported platform. When supporting a new platform a precise knowledge about this code is necessary to reproduce it for the new hardware. This might be a problem, because the initial code was written a long time ago or the responsible programmer is not even available any longer. In this case the code might be too old for a proper use and has to be rewritten.

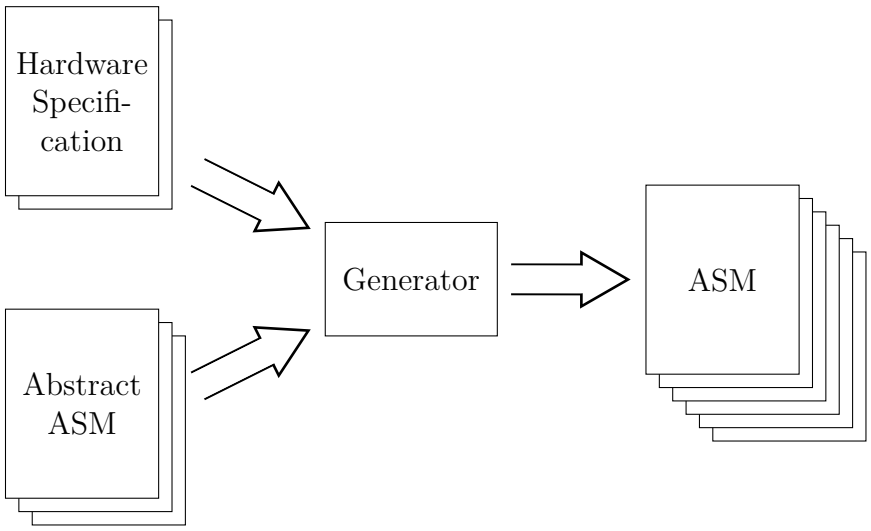

Figure 1.1: Basic idea of generating hardware-dependent code

When deciding to rewrite the whole code this often has to be done for all supported platforms so the programmer probably decides not to support old platforms anymore. Even then it is a lot of work. The first objectives that result from this problems are to **maintain sustainability**, provide **platform-independency** and **reduce complexity** by not just copy and paste *Assembler Code* but **reuse** the same code for all supported architectures. This will also **reduce the error rate**.

The other point why it is useful to eliminate hardware-dependency is in the nature of the architectures. A modern hardware consists of several processors, with various cores, memories and caches. The different elements might be connected with each other with buses of different speeds. The task of the operating system is, beside others, to organize the cooperation between those components, like doing the memory management and therewith optimize the hardware. The objective, especially in real-time operating systems is to reduce latency by reducing transfer ways of data. This might be done by using a cache or a memory which is directly connected to the current processor than using some other. Therefore an exact knowledge about the architecture the system is operating on is necessary. The derived objective for this task is to make it possible to describe an **Architecture Model** from which needed information for the operating system code are extracted.

1.2 Analysis

Before a concept for fulfilling the objectives can be developed, the tasks have to be determined. To split the hardware-specifications from the hardware-dependent code, a model of the hardware is needed. This model needs to contain frequently used values like addresses of different hardware components, cache-line sizes or the word size of the architecture to implement efficient loops (e.g. during data prefetching). More and detailed requirements on the hardware model are described in subsection 3.2.2.

The second aspect that makes the *Assembler Code* a hardware-dependent language is the usage of specific instruction sets. These differ among each other, nevertheless they are sometimes similar to each other. These similarities can be utilized to develop a language, which uses abstract instructions that can later be translated to concrete instructions of the corresponding architecture. Therefore it is necessary to define an instruction set for the different architectures.

The instruction sets are defined independent from the hardware models. Each hardware model uses one specific instruction set, thus it is resusable for later revisions of similar hardware platforms.

1.3 Concept

To fulfill the targets there are three main components necessary:

- Definition of an Architecture Model
- Definition of an Instruction Set
- Definition of *Abstract Assembler Code*

For each of these components a *domain-specific language* is developed. An introduction about *domain-specific language*s and tools for development, as well as an introduction about architecture models are given in chapter 2. The modeling of the three components is described in chapter 3, chapter 4 concerns the generation of *Assembler Code*. In chapter 5 the solution is evaluated and the probabilities of future work is discussed.
This master thesis focuses on generating *Assembler Code*. It is however possible to insert new code generators any time, e.G. generating C/C++ Code or even prose (maybe to write a documentation for the developed operating system).

2 Fundamentals

In this chapter some basic fundamentals are introduced. At first (section 2.1) two related projects which cover both main aspects of this work are presented. After that the used tools including the *domain-specific language* development framework is discussed (section 2.2). The last section (section 2.3) covers the testing environment (hardware and operating system) the project is tested in.

2.1 Related Work

The implemented project covers two aspects to maintain hardware independency. The first one is the development of a hardware model that provides on demand information about the hardware. The idea of modeling an architecture is already implemented with the *Linux Devicetree* (subsection 2.1.1) but there are still some drawbacks that the new architecture model tries to avoid. The next aspect is the automatic generation of *Assembler Code* which is also covered with the *QEMU Tiny Code Generator* (subsection 2.1.2). However its purpose is different.

2.1.1 Devicetree

The devicetree is a data structure to describe the hardware configuration of a system. It is used by the *Linux Kernel* to find devices and register them in the system [5]. Figure 2.1 shows an example from the Devicetree Specification [6]. It describes the computing, evaluation and devolepment platform MPC8572DS using Power Architecture technology [12]. Naturally it occurs as a binary and therefore non human-readable file. As this is only readable by the *Linux Kernel* it is obviously not useful to generate operating system code. For now it is also just usable for some selected platforms.

It is possible to make changes on the devicetree by decompiling it into the human readable code, modifying it and compiling it again. However, if there are any mistakes the system might get unbootable. The idea of the project implemented in this master thesis is to use the information from the hardware model to generate some operating system code. Therefore it is always possible to recheck the implementation and the decision about success or failure is made by the operating system implementation itself.

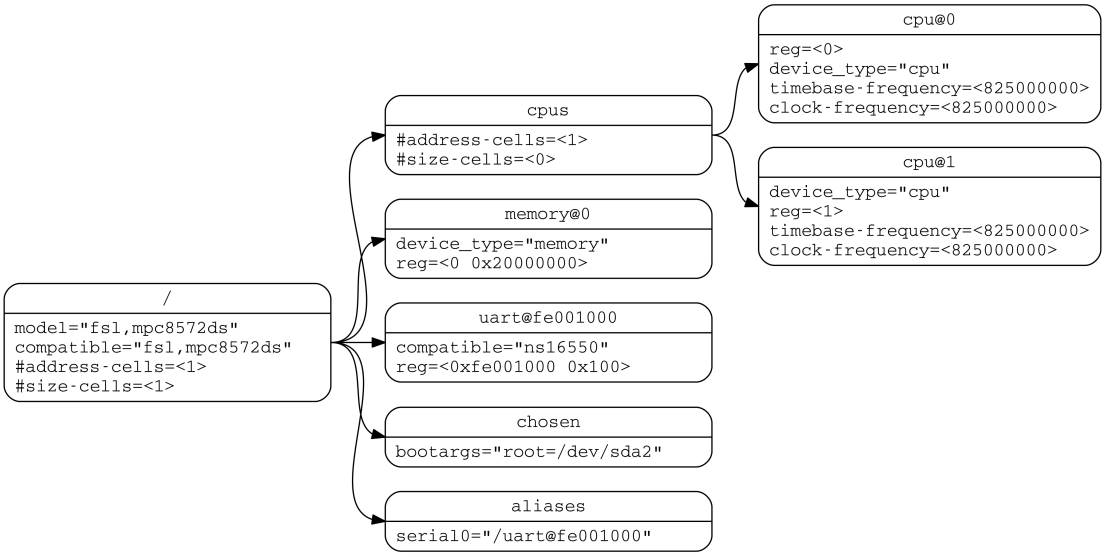

Figure 2.1: Example representation of a simple devicetree [6]

2.1.2 QEMU Tiny Code Generator

The QEMU Tiny Code Generator (short TCG) is part of the QEMU project - an open source machine emulator. To use the hardware specific *Assembler Code* for the platform which has to be emulated it is translated into *Assembler Code* for the platform on which the other is emulated. This is done in two steps: First the given CPU instructions from the emulated platform are encoded and used as an input for some given predefined operands. Then CPU instructions for another platform are generated.

To give an example, the Power PC instruction

```
addi r1,r1,1
```

causes the execution of the C-Function

```
tcg_gen_addi_tl(cpu_grp[rD(ctx->opcode)], cpu_gpr[rA(ctx->opcode)], simn);
```

which has been implemented in the TCG translation engine. Same colors indicate matching parts of the original instruction and the TCG instruction. The result of the function is

```
add $0x1,%ebx
```

which is the same instruction for x86 architectures (AT&T syntax). The idea of generating specific instructions from an abstract code is used in this project. While QEMU is using a given set of operands [7] that also just work for some architectures in this project all operands can be defined by the developer himself

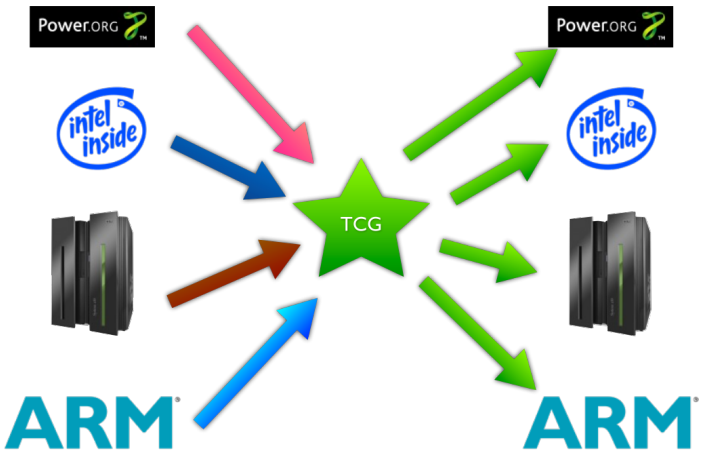

Figure 2.2: Alexander Graf: QEMU's recompilation engine [13]

for every platform he likes. The input is also defined by the user and not by other assembler code. There are some other features that are described in the chapter below like a dynamic register allocation, defining control structures and using loops.

2.2 Tools

In this part a brief introduction about *domain-specific language*s is given (subsection 2.2.1) and the framework described (subsection 2.2.2).

2.2.1 *Domain-specific language*

"Domain-specific language (noun): a computer programming language of limited expressiveness focused on a particular domain."[9] This definition about *domain-specific language*s was made by Martin Fowler in his book of the same title.

A *domain-specific language* serves a particular domain. Mostly it is integrated as a small part in a larger system, like a part of operating system code. The language is focused on this particular part. It requires expertise, because it can just be used for a small number of tasks, but it reduces the complexity and therefore error-proneness because it is customized for the domain.

2.2.1.1 How a *domain-specific language* works

There are two different kinds of *domain-specific language*: Internal and external. While internal ones are embedded in a host language and use its principles, an

external *domain-specific language* has an own syntax and semantic. Examples for external *domain-specific language*s are regular expressions or SQL, examples for internal *domain-specific language*s are Lisp or the Ruby framework Rails. The *domain-specific language*s written as part of this master thesis are all external *domain-specific language*s.

Some source code written in the *domain-specific language* is parsed and populates a semantic model. It is possible to write an own parser or use an existing one provided by a framework. There are many possibilities to proceed with this model:

The model - and thereby the source code - can (and must) be verified. Even if it is syntactical correct (otherwise the model cannot be created) there might be semantic issues. These must be determined by the developer. The validation step is very important to avoid unexpected results or crashes during the further processing.

The model can be used by other models. This is useful if there are more than one languages which are dependent on each other. Circular dependencies should be avoided here, e.g. by a layered overall model used in this project (see section 3.1).

The model can generate new files. These can be all kinds of files which can be represented textual such as textfiles, source code for *general-purpose language*s or source code for the *domain-specific language* represented by the model.

2.2.2 Xtext Framework

"Xtext is a framework for development of programming languages and domain-specific language" [11] and is originated by *itemis* [17]. It is part of the Eclipse Modeling Framework and probably the most popular framework for developing *domain-specific language*s. *Xtext* creates a full infrastructure which contains a parser and other language elements. It is sufficient to create a grammar file and run a *Workflow* which is already basically defined by a template when a new dsl-project is created. It is also possible to create a Plug-In for Eclipse which already contains an *Editor* when building the project. That *Editor* can be customized by extending and implementing given classes written in *Xtend* (see subsection 2.2.3).

2.2.3 Xtend

"Xtend is a statically-typed programming language which translates to comprehensible Java source code." [15] It is a dialect of the Java language and is said to improve some aspects. Some examples are that *Xtend* goes without semicolons, simplify the getters and setters (`instance.getName()` becomes `instance.name` and `instance.setName("name")` becomes `instance.name = "name"`). An exhaustive list of improvements is given in the documentation [15]. In the *Xtext* framework *Xtend* is used to write the generator and validator and customize the

Listing 2.1: Example Grammar Rule from the Architecture DSL

```
EComponent:
  type = CType name = ID (':' parents = ParentList)?
  '{'
    properties += Property*
  '}'
;
```

editor (e.g. Content Assist and Outline). These files are immediately translated into *java* files.

2.2.4 EMF Metamodel and Xcore

As described above in subsubsection 2.2.1.1 a *domain-specific language* populates a model. To describe this model a metamodel is created from the given grammar. The EMF metamodel consists of two description files. The *ecore file* contains information about the language components (**Classifiers**) and their properties (**Structural Features**). From this metamodel java code is created [8]. The generator metamodel contains information about the imported models (**usedGenPackages**), Metainformation (e.g. **copyrightText**) and the information from the ecore model in a parent-child-relation. The listings below show the same language component from the architecture model in the ecore- (Listing 2.2) and generator (Listing 2.3) - Description File. The base component has been defined as a grammar rule (Listing 2.1). All names describing another grammar element (type, name, parents, properties) can be found as names of structural features in the ecore model (`<eStructuralFeatures name="..." .../>`). The generator model refers to these features (`<genFeatures ecoreFeature="..." .../>`). The types of the features are attributes of the ecore features called `xsi:type`. Primitive datatypes like the terminal rule 'ID' (terminal rules are always Strings) or the rule 'CType' which is an enum rule (enum rules are Strings as well) are called `ecore:EAttribute`. References to other rules are called `ecore:EReference`. The third attribute of the ecore features are called `eType`. If they refer to other implemented grammar rules the `eType` is this type (e.g. `eType="#//Property"`). Primitive types are already implemented in ecore (`ecore:EEnum` and `ecore:EDataType`).

To edit these metamodels, like adding properties or define some methods, it is useful to transform the metamodel into a textual, editable and clearly layouted one. The created metamodels can also be edited and have a textual representation (XML) but they are not created to be edited. The better solution is to transform it into an xcore-model. "Xcore is an extended concrete syntax for Ecore" [16]. The xcore model can be transformed to an ecore model any time, but this is

not necessary. The manifest file defines which model will be used to generate the *domain-specific language*. Listing 2.4 shows the transformed model code, extended by a method `getSize()` shown in red.

2.2.5 ANTLR Parser Generator

Because it is a lot of work to write a parser for a whole language (even if it is "just" a *domain-specific language*), the Xtext Framework contains a parser generator. Everytime the grammar of the language is changed the parser is automatically updated to these changes. This framework uses the ANTLR Parser Generator. ANTLR generates recursive decent parser for LL(k)-grammars. A tool which has become very handy to detect left-recursion and non-deterministic rules is the ANTRLR GUI Development Environment ANTLRWorks.

2.3 Testing Environment

To test the implementation of the project later it is integrated into the research operating system *CyPhOS*[4] which is then run on the ODROID-U3

The ODROID-U3 is a development platform which contains of a Samsung Exynos 4412 Prime Cortex-A9 Quad Core processor. The Cortex-A9 Processor [3] has four coherent cores using a 32 bit ARMv7 CPU, a 32 kB instruction cache and a 32 kB data cache. The cores share an 1 MB L2-Cache. The LPDDR2 RAM has a size of 2 GB.

The research operating system *CyPhOS* is a real-time operating system that is running on commercial off-the-shelf hardware. It uses an operating system controlled cache management, therefore it needs a detailed knowledge about the used hardware. The purpose of this work is to determine these information from a hardware model. These includes the configuration of caches and the bus connections between memory units as well as the access time on memory.

Listing 2.2: Ecore

```
<eClassifiers xsi:type="ecore:EClass" name="EComponent" eSuperTypes="#//Component
    #//BlockContent">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="type" eType="ecore:EEnum
        platform:/resource/edu.udo.cs.ess.libs/model/lib.ecore#//CType"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
        eType="ecore:EDataType
        platform:/resource/org.eclipse.emf.ecore/model/Ecore.ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="parents"
        eType="#//ParentList" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="properties" upperBound="-1"
        eType="#//Property" containment="true"/>
</eClassifiers>
```

Listing 2.3: Genmodel

```
<genClasses ecoreClass="Architecture.ecore#//EComponent">
    <genFeatures createChild="false" ecoreFeature="ecore:EAttribute
        Architecture.ecore#//EComponent/type"/>
    <genFeatures createChild="false" ecoreFeature="ecore:EAttribute
        Architecture.ecore#//EComponent/name"/>
    <genFeatures property="None" children="true" createChild="true"
        ecoreFeature="ecore:EReference Architecture.ecore#//EComponent/parents"/>
    <genFeatures property="None" children="true" createChild="true"
        ecoreFeature="ecore:EReference Architecture.ecore#//EComponent/properties"/>
</genClasses>
```

Listing 2.4: Xcore

```
class EComponent extends Component , BlockContent {
    CType ^type
    String name
    contains ParentList parents
    contains Property[] properties
    refers BlockInstance BContainer

    op long getSize() {
        //...
    }
}
```

3 Model

The defined *domain-specific language* actually consists of five individual languages resulting in five separate models that interact with each other. The structure of the overall architecture is shown in section 3.1 and described detailed in the section 3.1. Models within the same layers do not communicate with each other and only components on a lower level can refer to a component on a higher layer.

3.1 The three layers

The five separate models are part of a bigger overall model which consists of three layers. The top layer is constituted by a shared instance, called *Lib* which acts as an interface between the elements of the second layer. It mostly declares *Enums* that the underlying models can refer to, as adapters for different components. To give an example, the *ISA Description File* declares two types of symbols: *Symbols* and *Symbol Pointers*. The *Abstract Assembler Code* can then make use of these two types of symbols. Now for both language elements the method `getType():SymbolType` is implemented. `SymbolType` is an enum which is defined in the shared *Lib* containing the values `SYM` and `PTR`. If a symbol is used in the *Abstract Assembler Code*, the corresponding definition made in the *ISA Description File* can simply be found by comparing this `SymbolType`.

The second layer consists of the different submodels (**Architecture, ISA, Register** and **Abstract Assembler**). While the *Register Model* is always part of a certain *ISA*, the remaining submodels are independent from each other. The *ISA* provides information about the active *Register Set* to the *Generator*, but further processing is handled exclusively by the *Generator*. The *Architecture Model* on this second layer is not directly necessary to create *Assembler Code*. It provides values for certain variables that can be used by *Abstract Assembler Code*. In such cases, the *Generator* creates new *Abstract Assembler Code* where those variables are replaced by their values and the result is then directed back to the *Generator*. The *Generator* is the bottom layer of the entire system. It generates the *Assembler Code* from the upper models and is covered in detail in chapter 4.

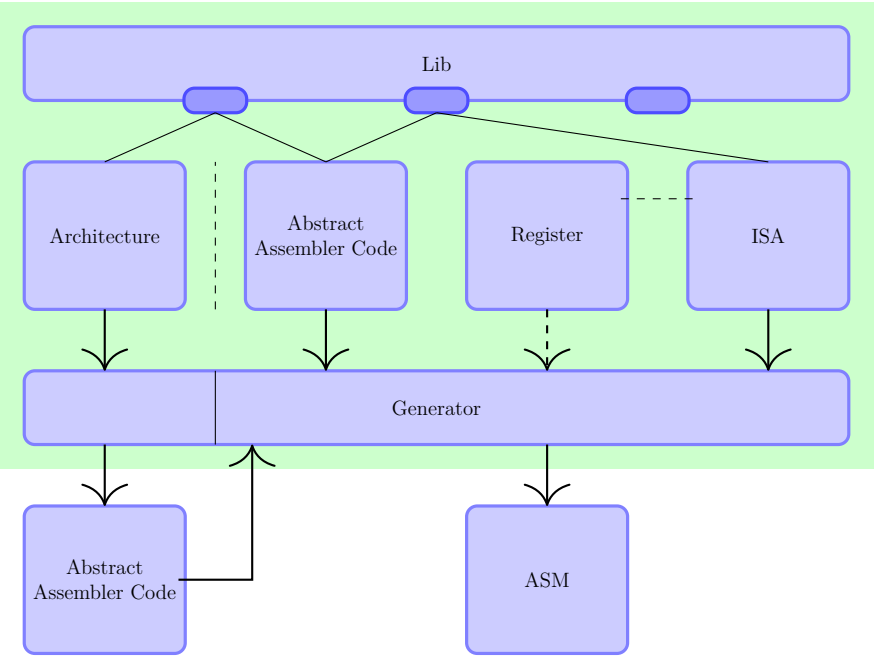

Figure 3.1: Architecture of the generation process using the models populated by the different DSLs

3.2 Architecture-Model

Describing the architecture is an important step towards the creation of architecture specific code. The objective is to create a model with which most common architectures can be described with and from which all relevant properties of the architecture can be extracted. First of all the requirements of such a model have to be determined. For that purpose several different architectures have been investigated, two of them are described in section 3.2.1.

3.2.1 Examples of Architectures

Two different architectures are shown in Figure 3.3 (Exynos4412) and Figure 3.4 (IBM x3850 X5). Figure 3.2 shows the *Cortex-A9* Processor, which is used with the *Exynos4412* and other ARM-Architectures. By analyzing the architectures several objectives that have to be met by the model can be determined. These objectives are described in detail in subsection 3.2.2.

3.2.2 Objectives

From these different architectures some important objectives can be developed: The first one is the **Reusability**. Both architectures contain components (like

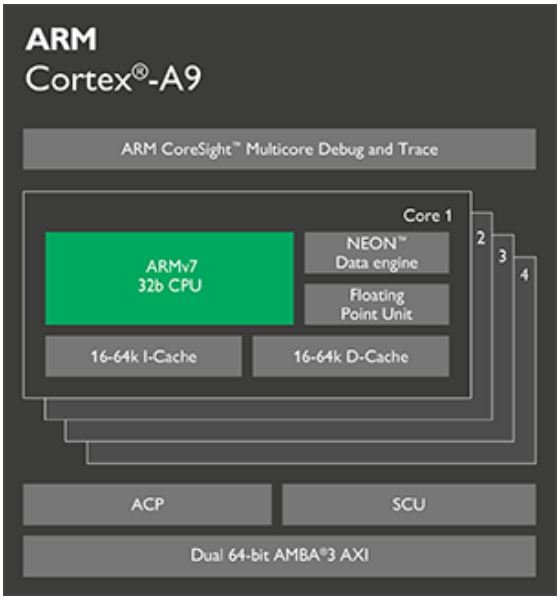

Figure 3.2: Cortex A9

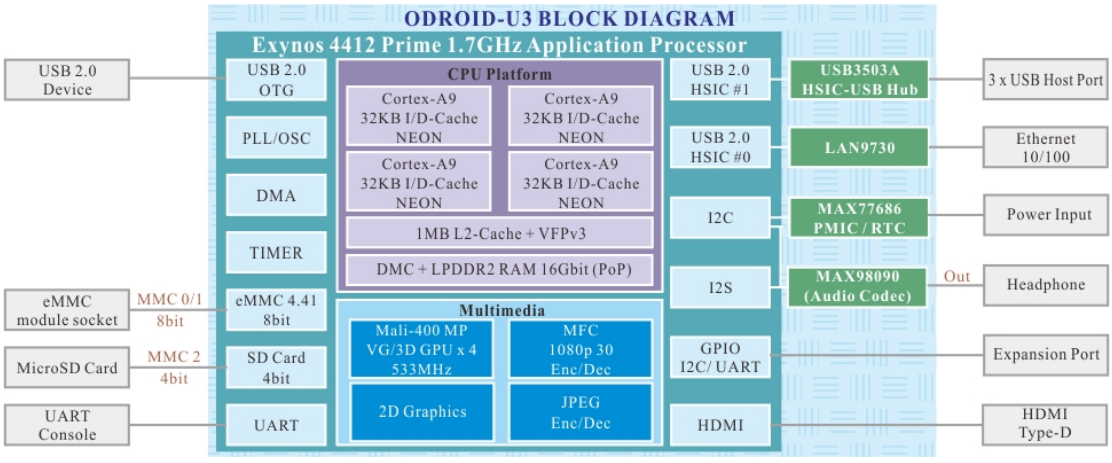

Figure 3.3: Exynos 4412 as the ODROID-U3 Application Processor

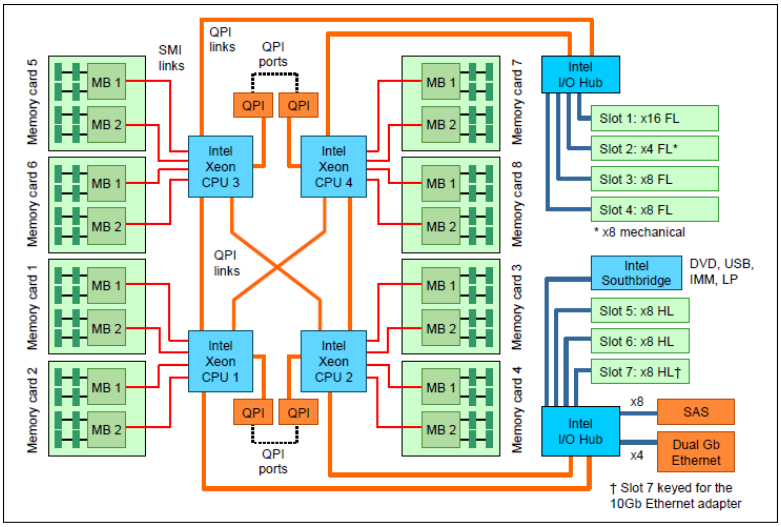

Figure 3-7 Block diagram for single-node x3850 X5

Figure 3.4: IBM x3850 X5

in

the *CPU*), that are used several times within one architecture as well as in other architectures. As these components can occur in different variants[1] these components have to be described with variable properties, which leads to the next objective. Components contain **Properties**. For now the project supports the following component-specific properties: *base-address* (the first address of the hardware component), *size* (from the base-address and the size the end-address of the address range can be calculated), *cacheline size* (for caches). The property *wordlength* is defined architecture-wise. Not all components have to and may not define all of these properties. To figure out which component define which property the components must be distinguished by a type, the **Component Type**. For now three different component types can be used: *Process Units (pu)*, *Memory Units (mu)* and *I/O-Devices (io)*. Memory Units can be either memories (like RAM) or Caches. They are not distinguished by their label, but by their properties: A Memory must not contain a cacheline size, the cache does not contain a base-address. If the base-address of the cache is accessed, the cache provides the one of the memory it belongs to. Therefore a **Hierarchy** between single components is necessary. Each component can be a subcomponent of another (e.g. L1 and L2-Caches). Either a single component or a **Group** of components can be such a subcomponent. To handle objectives like reusability or hiearchy, components have to be grouped. A group can be defined once and may contain variables. It is also possible to declare a group in a different architecture which is implemented by the current one. To give an example the Cortex-A9 Processor is described as an own architecture which is used by other architectures. It is a

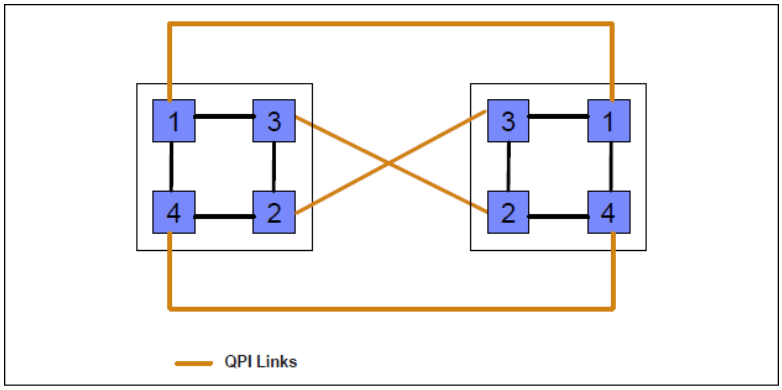

Figure 3-15 QPI links for a two-node x3850 X5

Figure 3.5: QPI links fo a two-node x3850 X5

quad-core processor with four identical cores. The caches of this core have variable sizes, therefore the core as a group has to be variable, too. Each architecture which uses the Cortex-A9 processor can now use these cores with a certain cache size.

The other way to connect components beside defining the hierarchy are **Links**. Figure 3.5 shows the two ways components can be linked. Either components in the same group can be linked (black lines) or components from different groups (brown lines).

3.2.3 Source Code

On top of the *Architecture Description File* there can be defined includes. This is necessary, if references to components of parent architectures are made. If the Eclipse plugin for the created *domain-specific language* is used references to other files can be made without the include. However the generation process is not part of this plugin, therefore the referenced resource must be made known to the current one (see section 4.2 for details). The include is just the filename without a path. Therefore it can be found within the whole project, but the filenames should be unique.

```
#include parent.arch
```

The next part of the source code is the architecture container. It must have a unique name and may inherit another architecture which must defined in one of the included files. Each architecture may, at least the most top architecture must define the word length of the architecture by adding # and the size to the name.

```
SubArch#4:Arch{ ··· }
```

The word length is fixed-sized characteristic of an architecture. If an architecture is marked **abstract** it does not need to define the word length, but all of the implementing architectures need to define the word length. Inside of this container the components are described. There are two types of components. The explicit components describe a real hardware component, which is described with a type (for now these types are *mu, pu* and *io* for memory units / caches, process units and i/o-devices) and a name. Like the containers these components can refer to another component. This defines a hierarchy between the components. The explicit components contain properties regarding to their types as described in Section 3.2.2. The block components define a group of explicit components and instances of other block components. They are used like constructors with a name and a set of parameters. As a difference to object orientated *general-purpose language*, they contain the object information directly within this constructor-like block component: The body contains other objects, explicit or as an instance. An instance of a component is created with the `new`-operator. Listing 3.1 shows two block components. The second one (`OtherComponent`) contains a cache (indicated by the keyword **mu** and the definition of the accessSize). The first block component `Component` also contains an explicit component (a memory component with a baseAddress) and an instance of the second component, delegating its own variable `Size b` to it.

Listing 3.1: Block Components

```
Component(Size a, Size b) {
  mu mem {
    size: a
    baseAddress: 0x1
  }

  new OtherComponent(b)
}

OtherComponent(Size a) {
  mu cache {
    size: a
    accessSize: 4
  }
}
```

When new components are instantiated, they get an increasing ID, starting from zero, by which they can be referred. This ID is needed if the components are linked. To link several components they need to be instantiated first. Each component can be linked with one or more other components, even with subcomponents. To link components within the same group, there are just links added after instantiating the needed components. The links are unidirectional and each component can link more than one other component.

Listing 3.2: Link different blocks

```
1  architecture:#4 {
2      Socket() {
3          new Core() //Core:0
4          new Core() //Core:1
5      }
6
7      Core() {
8          //...
9      }
10
11     new Socket() { //Socket:0
12         link(Core:0,[Socket:1.Core:1])
13         link(Core:1,[Socket:1.Core:0])
14     }
15
16     new Socket() { //Socket:1
17         link(Core:0,[Socket:0.Core:1])
18         link(Core:1,[Socket:0.Core:0])
19     }
20 }
```

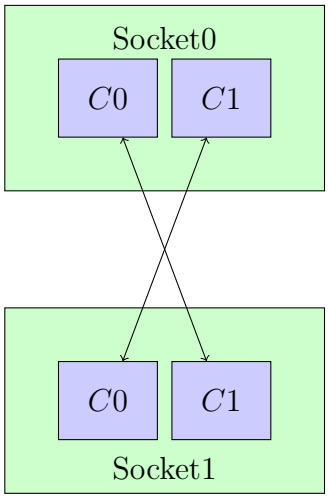

Figure 3.6: Implementation (left) and graphical representation (right) of a simple two socket system

```
link(comp:0,[comp:1,comp:2,comp:3])
```

To link to subcomponents of other groups the link is defined inside the instance `new comp( ) { link(...)}`. The links refer to instances defined in the declaration of the component. To link to subcomponents of other group the referenced scope is the container of the instance. This might be the declaration of another group or the architecture itself. The subcomponent is found by defining a link to its container and the subcomponent appended to it with a dot. An example of a simple two socket system is shown in Listing 3.2. The architecture consists of two sockets with two cores each. These cores are wired cross-wise. The socket as a hardware component is declared in line 2. They consist of two instances of the hardware component `Core` which is a blackbox component (line 7) in this example. The sockets are instantiated two times (line 11 and line 16). Within these instances the links are defined. Each core (`Core:0` and `Core:1`) of the first Socket is connected with the opposite core (`Core:1` and `Core:0`) of the other Socket (`Socket:1`). The cores of the second Socket are connected with the cores of the first socket in the same manner.

3.2.4 Validation

In this section it will be shown, that the objectives of Section 3.2.2 are fulfilled with the *domain-specific language* implemented to model the hardware-architecture. The examples (Section 3.2.1) have been implemented. To show the **Reusability** two architectures that both contain a *CortexA9 Processor* have been implemented in the *Architecture Description File*. The Cortex Architecture is an abstract one, it is not possible to use it on its own. It consists of an Process Unit and two L1 Caches. When iterating over the caches of each implementing architecture, these architectures also contain these L1-Caches.
For each component there can be defined some **Properties**. By implementing the validator for the *domain-specific language* it can be made sure, that each hardware component of a certain type can just define some specific properties (e.g a cache must define a cacheline-size, but must not define the base address which is inherit by the corresponding memory).
In most cases the **Component Types** are directly defined. Memories and Caches (both labeled with *mu*) are differentiated by their properties. Iterating over the different types of components show that each component is correctly associated with its type.
The correctness of the **Hierarchy** of components can easily verified by calling the methods `getChildren()` and `getAllParents()` (this returns a list of all parents and their parents up to a root element)
It can be shown, that all components of a **Group** can be found in the hardware model with their correct variables.
It is possible to create **Links** within the same group as well as between components of different groups. For now the links have no further use. (See section 5.3.2).

3.3 Registers

Registers are one of the most important components of the hardware. They are used to store values. They have different sizes, different purposes and are usually accessed as a whole and sometimes the high or low half. As a special case of the modeling the registers always belong to an *Instruction Set*.

3.3.1 Examples

There are two important examples of *Register Sets* that will be considered here: **ARM-Registers** [2] and **x86-Registers** [10]. The classic ARM Register Set contains 16 32 bits wide registers (R0 - R15). The last three registers are special registers (stack pointer, link register and program counter). The Register Set can be extended by other registers like floating point registers (single and double).

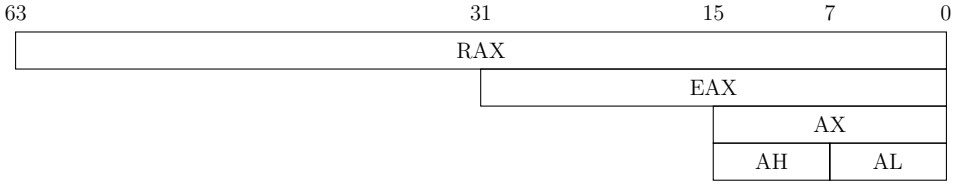

Figure 3.7: The Accumulator Registers of the x86 register set

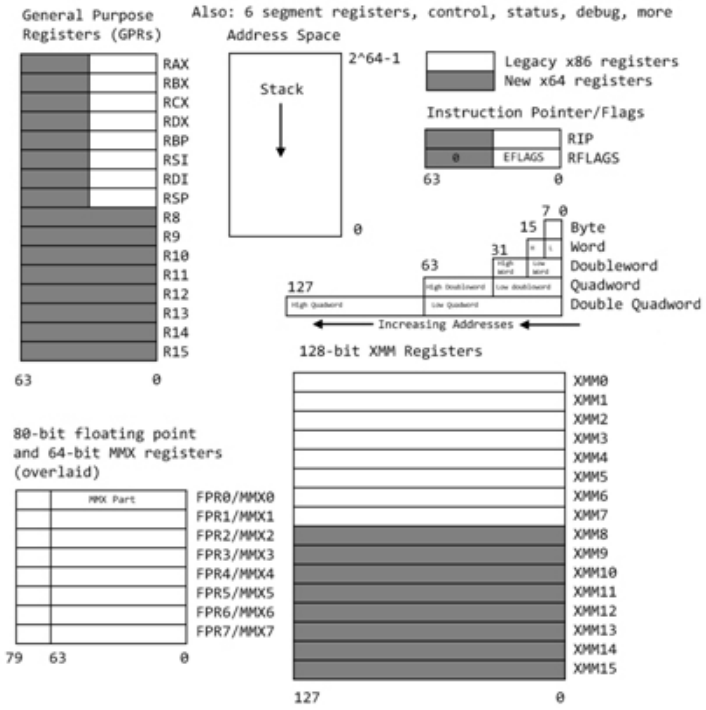

Figure 3.8: Register Set x86 [10]

All registers can be accessed by aliases when using *Assembler Code*. To give an example the stack pointer can either be accessed by `sp` or `r13`. The names are case insensitive.

The x86 Register Set (extended to 64 bits) is shown in Figure 3.8. There are also general purpose registers (top left) and floating point registers (bottom left). The registers of the 64 bit extension are marked in gray.

A property of the x86-Architecture is the downwards compatibility. It is always possible to access the lower parts of the register. Figure 3.7 shows an example for the x86 Accumulator Registers. `RAX` is the 64bit-extension of the `EAX` (32 bits). Its lower part can be accessed by `AX` (16 bit). This contains the higher part (`AH`) and the lower part (`AL`), each is 8 bits wide.

3.3.2 Objectives

The two main objectives for generating code which make it necessary to have a model of the registers are on one hand to verify the use of registers, on the other hand to allocate and release registers automatically. Therefore it is necessary to differ the purpose of the registers and the size. In case of the ARM-Registers the aliases are an important point: The registers can be accessed by a valid alias (e.g. stack pointer **R13** alias **SP**). Another purpose of the aliases is to handle the switch of CPU mode. **R13** is valid for USR mode and SYS mode, but not for the other modes. To reduce this variety these registers should also be accessed by **R13** but automatically substituted by e.g. **R13_fiq**.
Since there can be a large number of similar registers (like **R0-R15**) it should be possible to declare these registers as a single group.

3.3.3 *Register Description File* Source Code

A register Set in the *Register Description File* always starts with its name and has its contents embraced by curly brackets

```
registerSetName { ··· }
```

The register defined inside are grouped by their purposes (**gp** for general purpose, **sp** for special purpose, **flags**, **float** for floating point registers) and the size in bits of the registers in this group.

```
purpose:64 { ··· }
```

Registers can be defined as single registers, or as a group by using wildcards and a range. If wildcards are used, this group also needs a name to access them. For example

```
general:R%[0..15]
```

defines the first 16 registers R0-R15 of the ARM-Register Set. Each register consists of a higher and a lower half, which might be or might not be accessed. This is done as followed:

```
Register(high,low)
```

The high or low part may be empty. Even both parts, but there is no need to add the parentheses then anyway. As a high or low part there may be accessed either an explicit range of registers (the range must match the parent register range), a group of registers (the size of those registers must be half of the parent register size), or a single register of half size). Registers that are defined directly as a part of other registers may also contain a high or low part. Registers can be accessed

Listing 3.3: Floating Point Registers in ARM

```
Reg {
    float:32 {
        single:S%[0..31]
    }

    float:64 {
        double:D%[0..15](float:32.single.even,float:32.single.odd)
    }
}
```

by their purpose, their size and their name or group name. The following example shows the access of x86 Stack Point Register (32 bit) as lower part of the 64 bit Stack Point Register.

```
RSP(,gp:32.ESP)
```

It is also possible to access just a part of a register group by adding even or odd. This is needful for registers like the ARM floating point registers. As shown in Listing 3.3 there are single (32 bit) and double (64 bit) floating point registers. Two consecutive single floating point registers constitute one double floating point register. Therefore the higher part of the double register is even, the lower part the next even register. The double registers are called `D`, the single registers `S`. As a result each register Dn consists of the higher half $S(2n)$ and the lower half $S(2n+1)$. This can be realized with the usage of even and odd.

3.4 ISA

The *ISA Description File* is one of the core elements of the generating process. In this file the abstract instructions are defined with their corresponding architecture-specific instruction.

3.4.1 Objectives

In this description file the user defines abstract instructions. The way these instructions are called should be the same for each architecture. Defining function-like code is an easy understandable way of doing this. The function head looks the same for each architecture, the content differs. The content might be explicit code (`ADD r1, r1, r2`) or contain other architecture functions to realize more complicated structures. As an example the *x86* instructions allow much more addition-instructions, like directly adding a memory address to a register or add

a constant to a memory address location in one instruction. To realize such behavior in the *ARM* instructions, two or more instructions are needed, like load, add and store. To later use the abstract assembler code on every *ISA*, every *ISA Description File* should implement all instructions the other do without limiting the possibilities of any architectures. Therefore it must be possible to implement instructions that consist of two or more other instructions.

The next point to realize is the handling of operands. An example is to load the content of a memory address location into a register. The memory address can be of different natures: Direct addressing with a hexadecimal value (`0x...`) or the address of a register (e.g. `[R1]`). Inserting offsets or register updates can even change the amount of operands.

The third objective the description file will cover is the definition of control structures, like simple loops.

3.4.2 Parts of the ISA

These objectives lead to a structure that divides the description file into different parts: The register definition (Section 3.4.2.1), that structures the declared registers in a way the instruction set architecture description file can use them. The definitions (Section 3.4.2.2) that allow to define parameter types more detailed. The instructions (Section 3.4.2.3), that define the instructions themselves and the control structure part (Section 3.4.2.1).

3.4.2.1 Registers

At the head of the *ISA Description File* there is a referenced to the used register file. As the other description files the *ISA Description File* is of the structure `name { //content }`. Here, one register description file has to be included:

```
isa_name use register_file.rg { ··· }
```

This position of the include points out that one and only one *Register Description File* is part of this ISA. The registers that can be used a defined in are special *register block*. There the registers defined in the *Register Description File* can be grouped for own special purposes so that they can be accessed in a most possible hardware independent way. Two examples are shown in the listings 3.4 and 3.5. In both cases the registers can now be accessed with `<reg>` but contain the architecture dependent registers. The keyword **any** can be replaced by **even**, **odd** or a **range(a..b)** to access just a part of the registers if it is used for any purpose.

Listing 3.4: Isa ARM Register Definition

```
#register {
    <reg_f> any float:32
    <reg_g> any gp:32
    <special> any sp:32
    <reg> {
        <reg_f>; <reg_g>
    }
}
```

Listing 3.5: Isa x86 Register Definition

```
#register {
    <reg32> any gp:32
    <reg16> any gp:16
    <reg8> any gp:8
    <reg> {
        <reg8>; <reg16>; <reg32>
    }
}
```

3.4.2.2 Definitions

In the definition section of the *ISA Description File* `#define { }` used parameters can be defined in detail. An important example is the usage of the memory address. It is not necessary, but common that the memory address consists of two or even more operands, e.g. a register and an immediate value, e.g. to add an offset to an address. At this point it is also possible to add additional characters, like squared brackets. If the memory parameter is defined as followed

```
<mem> { [<reg>,#<imm>] }
```

and a register and a constant is passed where a memory variable is expected the memory variable in the instruction will be distributed by the expression defined above. The load instruction

```
load(<reg> r, <mem> m) {LDR r, m}
```

called with two registers and a constant will create

```
LDR r, [m0,#m1]
```

This procedure works in most cases, but there are some limitations: Listing 3.6 shows a scenario where it is not possible to decide which of the addition instructions will be used. The first step of the code generation regarding to the *ISA Description File* is to populate new instructions from such that use operands declared in the definition section, that can directly be called from the *Abstract Assembler Code* with the right parameters. The details are described in subsection 3.4.3. In this scenario the first addition instruction will remain the same, because register operands are final, that means they cannot be replaced by any rules of the definition section. The second addition instruction will populate a new instruction `addition(<reg> r, <reg> m0)`. This new instruction uses the same operands as the first one, but creates different content. This new instruction is shown in the box next to the original code. To avoid using the wrong instruction the instruction in the *Abstract Assembler Code* can explicitly be called with a memory operand as described in section 3.5.

Listing 3.6: Scenario for non-deterministic instruction definitions

```
#define {
    <mem> {
        [<reg>];
        //...
    }
}

#instructions {
    addition(<reg> r1, <reg> r2) {
        ADD r1, r1, r2
    }
    addition(<reg> r, <mem> m) {
        r2 = new <reg>
        load(r2,m)
        addition(r2,r1)
        store(r2,m)
    }
}
```

```
addition(<reg> r, <reg> m0) {
  r2 = new <reg>
  LDR r2,[m0]
  ADD r2,r2,r1
  STR r2,[m0]
}
```

Listing 3.7: Defining Addition for the ARM Instruction Set

```
1 addition(<mem> m, <reg> r) {
2     r2 = new <reg_g>
3     load(r2, m)
4     addition(r2, r)
5     store(r2, m)
6 }
```

3.4.2.3 Instructions

In this part the instructions themselves are defined. They are like functions consisting of the head, which should be the same in every *ISA Description File* used by a project, and a body. The body can be an explicit assembler instruction where just the parameters are replaced, or a group of one or more other instructions. These instructions must refer to explicit ones, so the maximum depth is two. This limitation is made to avoid loops and other unexpected behavior and maintain the clearness of the code.

An example why it must be possible to group two or more instructions was given in section 3.4.1. Listing 3.7 shows an example for implementing the addition instruction. In line 2 a new register is defined. This is an internal register. A free register (in this case a general purpose register) will be allocated to store temporary data and will be released afterwards.

There are some situations when it might be useful to just wrap one instruc-

Listing 3.8: Pattern of a Control Structure

```
1 #control_structures
2     name(<...> par1, <...> par2, ...) {
3         //pre-instruction
4         CONTENT
5         //post-instruction
6     }
7 }
```

tion into another. As an example *x86* does not support Store and Load like *ARM* does, the corresponding instruction here is **Move**. An *x86* programmer uses `move(<reg>,<mem>)` which wraps `store(<reg>,<mem>)` in the *ARM ISA Description File*, while the *ARM* programmer uses `store(<reg>,<mem>)`) which wraps `move(<reg>,<mem>)` in the *x86 ISA Description File*. To get a full hardware independency the defined instruction sets should be equal. Let A be the set of instructions used for the *ARM* Architecture and X be the set of instructions used for the *x86* Architecture the optimal instruction sets fulfills

$$A = A \cap X = X$$

However each single instruction set can contain special architecture specific instructions that are just used for internal purposes, e.g. in control structures or inside other instructions.

3.4.2.4 Control Structures

There are procedures that are used in a similar way at different parts of the *Assembler Code*. An example is a loop structure: The start and the end value are loaded into registers, a jump mark is set, some instructions are executed, the value in the start register is increased with a certain step and until a break condition is reached the execution steps back to the jump mark. The *x86* instructions already implement a loop instruction that can be used in a limited way but in the control structure part it is possible to define such a procedure just in the way it is needed by using variable stepsizes and conditions. Listing 3.8 shows the basic-pattern for defining a control structure. They are similar to ordinary instructions, but contain the keyword `CONTENT`. Different to the ordinary instructions the control structures in the *Abstract Assembler Code* have some content (see subsection 3.5.2). The keyword `CONTENT` is later replaced by this content.

3.4.2.5 Symbols

The last definitions that are made in the *ISA Description File* are Symbols. The structure is very strict, there are two types of symbols that have to be defined: The

Listing 3.9: Isa ARM Register Definition

```
#symbols {
    symbol { "@"}
    symbolPtr {"=@"}
}
```

Listing 3.10: Isa x86 Register Definition

```
#symbols {
    symbol { "SYM␣(@)" }
    symbolPtr { "$␣SYM␣(@)" }
}
```

symbol itself defines the syntax of the access to a variable. The symbol pointer defines the syntax of the access to the address of such a variable. The listings 3.9 and 3.10 show possible implementation for the ARM and x86 ISA. Whenever a variable is used as a symbol it will be replaced by the content of the symbol, the @ is a placeholder for the variable name.

3.4.3 Implementation Details

To use the model that is populated by the *ISA Description File*, there is some preprocessing necessary. The first step is to extend the implemented instructions to such instructions, that can be called with real parameters. In the implementation of the instructions there might be parameters, that cannot be called, but are defined in the section *Definitions*. Those parameters have to be resolved before the instructions can be used. The extension of an instruction is be done in several steps:
First of all it has to be determined if the instruction is final (the instruction can directly translate given parameters to operands) or needs to be extended. This is the case if it uses parameters, that are defined in the section *Definitions*. The parameter in this section is defined with one or more rules. For each of these rules a new instruction is created. An instruction definition consists of two parts: The head, which contains the parameter, and the body, which contains the instructions executed when called. Both parts need an update, because the type, and amount of parameters might differ. In the body some extra characters that might be defined in the rules have to be inserted. The procedure is quite simple: In the head of the instruction the parameter which needs an update is removed and replaced by the one or more parameters defined in the current rule. To avoid duplicate identifiers the position number of the new parameter in the rule is added. The following example shows the replacement of parameters in the ARM Instruction

`load(<reg> r, <mem> m) {...}`

which is defined in the `#instruction` section, with the `<mem>`-rule

`[<reg>,<con>]`

and the `<con>`-rule

```
#<imm>
```

which are both defined in the `#definition` section, which results in the Instruction

```
load(<reg> r, <reg> m0, <imm> m10) {...}
```

At second step the contents of the instruction have to be replaced. The most easy case is, that the content is the creation of a new register. There are no rules applied on this register, therefore this instruction remains the same. The second case is that the content is a direct instruction, e.g.

```
LDR r, m
```

Here, the parameter that needs to be updated is replaced by the content of the rule.

```
LDR r, [<reg>,#<imm>]
```

Each reference in the rule to another rule is replaced by the name of the parameter (including the counter).

```
LDR r, [m0,#m10]
```

The third and most complicated case is the replacement of an instruction call. To simplify the task and to avoid circular dependencies this call must always refer to a direct instruction. The parameters need to be delegated to this direct instruction. By using the extension method for this direct instruction, a matching candidate can easily be found.
When finishing the instruction extension all instruction bodies just contain of direct instructions.

3.4.4 Validation

To validate the *ISA Description File domain-specific language*, basic ARM and x86 instructions are implemented. By translating the *ISA Description File* to a new one with extended instructions instead of using the instructions directly there are two advantages: A visual check of the correctness of the extension is possible and there is no need to do the extension everytime code is generated. The new *ISA Description File* can be used if it is newer than the original file.

3.5 Abstract Assembler Code

The *Abstract Assembler Code* is the code from which the hardware-specific *Assembler Code* is generated. Here the defined instructions or control structures from the *ISA Description File* are called with some specific parameters.

3.5.1 Objectives

The *Abstract Assembler Code* must fulfill the same expressiveness as *Assembler Code*. The basic idea is to translate the *Abstract Assembler Code* directly to hardware-specific *Assembler Code*. Every *Assembler Code*, independent from the architecture it belongs to, consists of a sequence of instructions to execute. A sequence of instructions can be accessed by referencing an entry point. This reference is entered by (conditional) jumps within the *Assembler Code*, or from external code.

The objectives of this *domain-specific language* are quite clear: It must be possible to call the defined instructs with some parameters. These parameters can be of different kind:

- constants (mostly Integers or Hexadecimal Values, but also Strings)
- free variables used for automatic register allocation
- existing registers
- symbols
- pointer on symbols
- variables referring hardware specific values

The name of this container is later used as the entry point of the explicit *Assembler Code*.

3.5.2 Source Code

The structure of the Abstract Assembler Code is simple. There are containers

```
container_name { ··· }
```

which contain instructions

```
instr_name (par1, par2, ...)
```

and control structures

```
cs_name (par1, par2, ...)  { // some instructions }
```

To differ symbols from other parameters they are marked with a dot (`.symbol`) or a double dot (`..symbol_pointer`)

Listing 3.11 shows abstract profiling code that measures the time for executing a load instruction which loads the content of each memory address within the address range of a hardware component into a general purpose register. Listing 3.12

Listing 3.11: Profiling in *Abstract Assembler Code*

```
1  measure_mem {
2    load(registerA, .BENCHMARK_RESULT_ARRAY_START)
3    for (i, baseAddress, endAddress, "lo", accessSize)
4    {
5      addition(registerB, registerA, wordLength)
6      readCCStart(registerX)
7      load(registerC,i)
8      readCCEnd(registerB)
9      addition(registerB, registerB, wordLength)
10     addition(registerB, registerB, wordLength)
11   }
12 }
```

Listing 3.12: Generated Profiling Code in ARM Assembler

```
1  measure_mem_RAM_DCache:
2    LDR R3, BENCHMARK_RESULT_ARRAY_START
3    LDR R11, #0x40000000
4    LDR R6, #0x40040000
5    0:
6    CMP R6,R11
7    LDR R5,[R3,#4]
8    ADD R4,R4,R5
9    MRC p15,0,R2,c9,c13,0
10   DSB
11   STR R2,[R3]
12   LDR R8,[R11]
13   MRC p15,0,R10,c9,c13,0
14   DSB
15   STR R10,[R4]
16   LDR R7,[R3,#4]
17   ADD R3,R3,R7
18   LDR R12,[R3,#4]
19   ADD R3,R3,R12
20   ADD R11,R11,#256
21   BLO 0b
```

Figure 3.9: Abstract and Generated Code for the ARM profiling code

shows the generated *Assembler Code* for a Data Cache of an ARM Architecture. Even this small example shows the effectiveness of the *Abstract Assembler Code*: The *Assembler Code* is nearly as twice as long, corresponding registers are hard to see and recurring variables like wordLength are just defined once (in the architecture description).

3.5.3 Specific features

Even though the *Abstract Assembler Code* should be directly translated into *Assembler Code* one simplification is made. Frequently recurring structures (e.g. for loops) are defined as control structures within the instruction set, therefore a two-step translation is made: A resolution of the control structure to its containing instructions with given parameters and the generation of *Assembler Code* itself.

4 Code Generation

The final goal of this work is to generate *Assembler Code* that is used as Operating System Code. It is also possible to generate any *Assembler Code* with hardware specific components (e.g. profiling). For this purpose, the *domain-specific language* framework provides a generator which can easily create any text file. The input of this file is generated from the description files declared before. In a special generator file (see Section 4.1) a setup for which code should be generated is done.

4.1 The Generator File

The generator file is a configuration file which can be (for now) of two forms: *XML* and *JSON*. To add new configuration types easily this has been implemented intern as a new *domain-specific language*. It is possible to add new types just by modifying the grammar file. The grammar elements

```
XArchitecture returns GenArchitecture:{ ··· }
```

and

```
JArchitecture returns GenArchitecture:{ ··· }
```

create the Meta-Model Class `GenArchitecture`. There is no need to define somewhere which of these configuration types is used, the generator-configuration can just be implemented in the favorite style. The structure of the configuration is shown in Figure 4.1. Attributes of components are shown in squared brackets, the asterisk indicate that there are more than one component possible. Using an architecture for code generation is optional.
The detailed usage of the architecture model is described in Section 4.2.3 All files in this configuration files are only referenced by their names. There are no relative of absolute paths. Therefore every filename must be unique.

4.2 Preprocessing

Before the code is actually generated there is some preprocessing necessary. First of all, all required resources must be loaded. When starting the generator outside

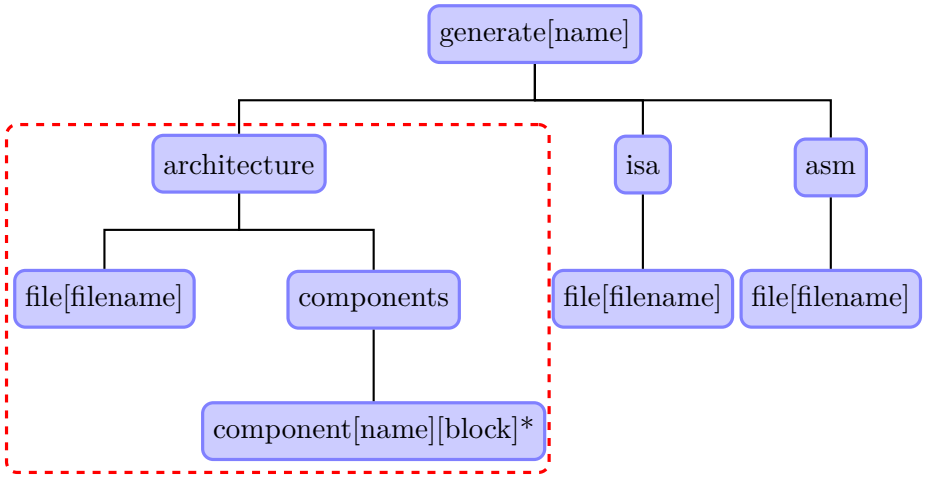

Figure 4.1: Configurate the Generator

from Eclipse, the generator does not know about any resource than the generator file itself. To every resource belongs a *resource-set* which carries information about which resources are known to the current one. This is necessary to resolve cross-references between different resources. The generator needs to know about the *Architecture Description File*, the *ISA Description File* and the *Abstract Assembler Code*. An *Architecture* might include other *Architecture Description Files* and the *ISA Description File* needs to know about the *Register Description File*. After loading all resources, they need to be validated. Every *domain-specific language* needs to provide its own resource. All *domain-specific language* have to implement the interface defined in the *Libinstance* to create a resource from a given path and validate this resource. The problem of this procedure is that a resource can only be validated, after all its subresources have been loaded. Because this resource may be corrupt there should be as less access on its contents as possible before validating. The only operation made on a resource before this validation is to find the filenames of its possible subresources to load and validate them. The validation of the resource is then done as soon as all subresources have been loaded and validated. It is therefore useful not to create too deep hierarchies. The framework provides three methods for preprocessing, processing and postprocessing: `beforeGenerate()`, `doGenerate()`, `afterGenerate()`, but only the first two are implemented now. Possible postprocessing might be the deletion of no longer needed temporary files or a further processing of results created by the execution of generated assembler code (for future work see section 5.3).

4.2.1 ISA-Preprocessing

The first operation which is done in the preprocessor is the creation of a new *ISA Description File*. It is possible to do the ISA preprocessing without creating a new file, but there are two advantages to do so: First there is no need to run the ISA preprocessing everytime the generator is started and second the generated file can be used again until it is older than the original ISA resource in which case it has to be regenerated. This saves up to 40% of preprocessor runtime (See subsection 5.2.1). The details of the ISA preprocessing have been described in Section 3.4.3.

4.2.2 Register Allocation

To have an overview which registers are free for use and which contain important information, the predefined registers need to be allocated. First the registers are ordered in the groups the *ISA Description File* uses. They are stored in a map with the register itself as the key and a list of the groups as its value. The `AllocatedRegister`-Class provides several methods to access registers by their name, their group or their alias. The next step is to allocate directly used registers. They are directly used, if the *Abstract Assembler Code* uses the name of a register or its alias. For each allocated register now or later a new instance of a `InternalRegister` is created and stored in a list of the `AllocatedRegister`-Class. The `InternalRegister` contains the name, the alias or aliases and the scope in which this register is used. As a scope the hashcode of the containing object is used. It prevents from wasting registers so they can be allocated in different contexts. All other registers, which are not used explicitly are allocated on demand while generating the code.

4.2.3 Using the Architecture Model

If the generated code is dependent on a hardware architecture the configuration component *architecture* must be defined. It is now possible to use parameters defined by the architecture description (e.g. wordlength) within the *Abstract Assembler Code*. Further usage is the iteration over hardware components. Therefore the *components* are defined within the configuration. It contains a name, which is one of **cache, mu, pu** or **io**. If using them a concrete *Assembler Code* block which must be inside the *Abstract Assembler Code* file used in the configuration is executed for each component of the given type. For each instance of a component of this type a new temporary *Abstract Assembler Code* file is generated which contains the same code of the origin *Abstract Assembler Code* file with concrete values for possible variables. With this technique special *Assembler Code* which refers to a concrete hardware component can easily be generated. This is espe-

cially then useful, if similar code (e.g. which just differs in address ranges) for a large amount of different hardware components is needed. The new name for this block is of the structure

`orginalName_[muName_]componentName`.

Special types require special processing. Caches always belong to a memory unit, therefore a new block is generated for each memory unit. All blocks are stored in the same file. The generated files (for all types) can be found at

`tmp\architectureName\type\orignalName_componentName.aS`.

Each of these files creates an own *Assembler Code* file of the same name.

4.3 Processing

After the preprocessing is done, the *Assembler Code* can be generated. All instructions in the *Abstract Assembler Code* (also the new created) need to be replaced with the real instructions. The class `ParameterMapper` provides some methods to determine a matching instruction from a group of candidates and finally build the instruction by replacing parameters and allocate registers. Candidates are at first all instructions provided by the updated *ISA Description File* with the same name as the calling one. Then all candidates with non-matching amount and type of parameters are eliminated. It is possible that more than one candidate is left. For now two rules of choosing the right one is implemented. The first one prefers the candidate with less replacement steps. Replacements using rules with final operands are referred to those which are using rules that need replacements again. The second rule prefers the candidate with less replacements. Here instructions with less non-final operands are preferred to those with (originally) more non-final operands. Currently the first one is used. After finding one candidate, the final instruction is built. First of all all new created registers within the instruction must be resolved: The next free register of the right type is found in the register list and is allocated. The register name is replaced in the final instruction. After finishing the processing of the instruction it is released again. For each parameter the corresponding parameter in the instruction code must be replaced by the given value. If this value is a symbol, it is also necessary that the symbol-code defined in the *ISA Description File* is used. If the value is a register which has not been allocated before, because it creates a new register, it is also necessary to allocate a new and free register. The allocation process is the same as in the preprocessing, the variable name is used as an alias so the same register is always used for the same variable. Finally the register names are replaced with the given values.

Figure 4.2 shows the interaction between the single components, the preprocessor and the generator. Preprocessor actions (extract the resources from the

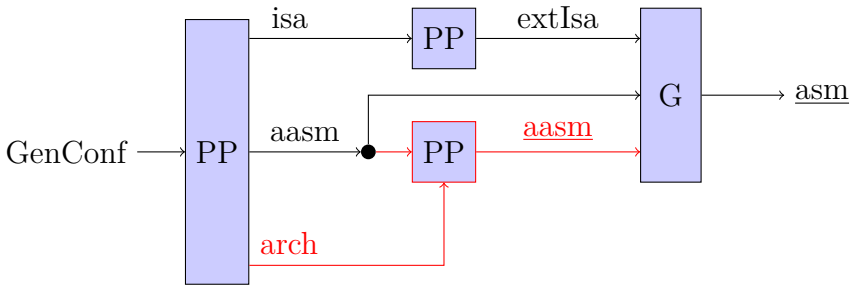

Figure 4.2: Interaction between the components and the generator / preprocessor

Generator Configuration File, resolve the instructions and include the architecture) are marked with `PP`, the generation of the final *Assembler Code* (asm) is marked with `G`. Underlined names mean that more than one resource can be created. The red path is optional. In the first step of preprocessing (first PP block on the left) the *ISA Description File* and *Abstract Assembler Code* file (and optionally the *Architecture Description File*) that are defined in the *Generator Configuration File* are loaded. The *ISA Description File* is directly processed and passed to the generator. The *Abstract Assembler Code* is also directly passed to the generator. If there is an architecture defined in the generator the corresponding *Abstract Assembler Code* files are generated by the preprocessing process visualized by the red PP block and passed to the generator. For each of the *Abstract Assembler Code* files one hardware dependent *Assembler Code* file is generated.

4.4 Examples

To show the process of an instruction through the generation a minimal example has been implemented. The setup is an architecture with one RAM and one L1 cache, 16 registers (R0-R15) with their aliases R_alias0 - R_alias15, and an addition and a load instruction. There are only the registers with an even number used by the *ISA Description File*.

4.4.1 Creating Code without including the Architecture

In the first scenario the architecture is not used, the only instruction called is `add(<reg> r, <mem> m)`, where `<mem>` is defined as `[<reg>,#<imm>]`. The procedure (graphical representation of the preprocessing in Figure 4.3) is the following

1. load and validate necessary resources
2. extend *ISA Description File* (see subsubsection 4.4.1.1)

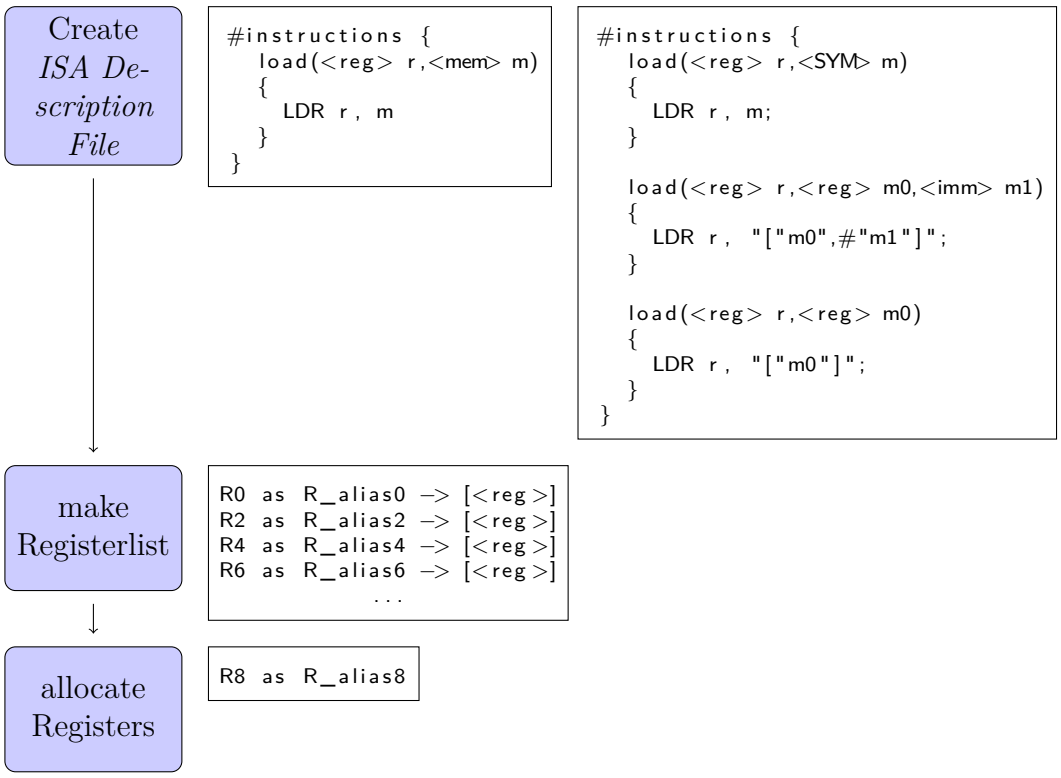

Figure 4.3: Flowchart of the preprocessing of a minimal platform.

3. group registers as defined in *ISA Description File* (see Figure 4.3 middle):
   In this example there is only one group with all 32bit-wide general purpose registers with an even number (R0, R2, ..., R14).

4. allocate directly used register (see Figure 4.3 bottom):
   The only directly accessed register (`R8 as R_alias8`is allocated as described in Section 4.2.2.

5. end of preprocessing

When processing the code a new *Assembler Code*-File with the name given in the generator is created. The *Abstract Assembler Code* is the following:

```
example { load(r, r_alias8, 0) }
```

From all instructions defined in the new *ISA Description File* those with the name `load` are possible candidates.

1. eliminate all candidates with a wrong amount of parameters
   One candidate left: `load(<reg> r, <reg> m0, <imm> m1)`

2. check type of parameter: All parameters (two registers and one immediate value) are matching
3. build the assembler instruction. For each parameter in the instruction there must be a corresponding parameter when calling the instruction
   a) non-allocated, non-defined register
   The variable in the final instruction is replaced with the recently allocated register. Allocated registers can differ from generation to generation
   b) allocated register
   The register is used with a predefined alias and replaced with the name of its real register
   c) immediate value
   remains the same

With the following the steps above the *Abstract Assembler Code*

```
example { load(r, r_alias8, 0) }
```

and the instruction definition

```
load(<reg> r, <reg> m0, <imm> m1) { LDR r, "["m0",#"m1"]" }
```

with the following replacements

$$\begin{array}{rcl} r & \Rightarrow & \text{some random free register, e.g. } R2 \\ m0 & \Rightarrow & R8 \\ m1 & \Rightarrow & 0 \end{array}$$

finally creates the *Assembler Code*

```
example:
    LDR R2,[R8,#0].
```

4.4.1.1 extend *ISA Description File*

When creating the extended *ISA Description File* there are two modifications made.

1. Insert symbols:
   The definition of symbols are necessary for the processing. If they are not defined they are declared to just provide the variable itself by inserting the symbols block

```
#symbols {
  symbol { "@" }
  symbolPtr { "@" }
}
```

2. Include resolved instructions:
   For every rule (in this example 2) in the definition of `<mem>` a new instruction is created. (See details in Section 3.4.3). The original instruction is kept with changing the `<mem>` to `<SYM>`: It is always possible to use the instruction with a symbol instead. The original and extended instructions are shown in the top middle and top right block of Figure 4.3.

3. Delete the definition section. There is no further use for it

4.4.2 Creating Code including the Architecture

To include the Architecture into the Code Generation the following modifications are made: The Architecture-Configuration is added to the *Generator Configuration File*.

```
<architecture>
    <file>platform.arch</file>
    <components>
        <component name = "cache" block = "example"/>
    </components>
</architecture>
```

Second the constant "0" in the *Abstract Assembler Code* is replaced by the variable "size".

```
example { load(r, r_alias8, size) }
```

The *Architecture Description File* `platform.arch` is the following:

```
Platform#4 {
    mu L1:RAM {
        size:32K
        acessSize:8
    }

    mu RAM {
        baseAddress:0x1
        size:2G
    }
}
```

When starting the generator the preprocessing is done as in the former example, just the generation of the extended *ISA Description File* is skipped because there were no changes in the original *ISA Description File*. Now for all components defined in the generation there are new temporary files with *Abstract Assembler Code* generated as described in Section 4.2.3. In this example there is only one file created, because there is only one cache **L1** defined in this minimal architecture. The name of the new file is *example_L1.aS*, the generated code-block (just one for one memory belonging to the cache) is now `example_RAM_L1`. Instead of *size*,

the size of the cache is inserted. It is 32 kB which results in a size of 262.144 bit. In the generation process itself there is first *Assembler Code* for these new files created. The code contains specific architecture-specific variables, otherwise there would also be the standard-code of the former example created.

The modified instruction now results into the *Assembler Code*

```
example_RAM_L1:
    LDR R2,[R8,#262144].
```

5 Conclusion

In this chapter it will be shown that the previous work fulfills the set targets. Therefore the Architecture Models of two different platforms are implemented and integrated into the operating system *CyPhOS*. It will be evaluated how much effort is required to integrate the system on the one hand and how much effort of coding is saved on the other hand. It will also be evaluated how much the additional runtime which is used to generate the code influences the total runtime of compiling the whole operating-system code. In the end of this chapter and this thesis some possibilities of future work are worked out.

5.1 Including the code into an environment

The testing environment is described detailed in section 2.3. The project itself consist of two parts: The *Generator* on one hand and the *Architecture Description File*, *ISA Description File*, *Register Description File* and *Abstract Assembler Code* on the other hand. The *Generator* is packed into an executable jar-File, the editors which the framework creates from the other *domain-specific language*s are added to an update-site and can be used as an Eclipse Plugin. While using the editors the *xtext-nature* has to be enabled for the project. The Description Files can be put anywhere in the project. There are two generator files, one for the *Exynos4412* architecture and one for the *PowerEdge R820 Rack-Server*. In both configurations the output name is `profiling`. This profiling code is executed for all **caches**, the corresponding block in the *Abstract Assembler Code* `profiling.aS` is called **measure_mem**. In this piece of *Assembler Code* a whole cache line is measured across the whole address range of the cache or the RAM. The *Generator*s are executed from the *Makefile* with both *Generator* configuration files as its arguments. The name of the generator Figure 5.1 shows the hierarchical structure of the output files.

The generated files have to be integrated into the build infrastructure of the used system. This system needs to know the entry points of the generated *Assembler Code* and execute it on the corresponding CPU. This takes minimal effort depending on the system in which to integrate.

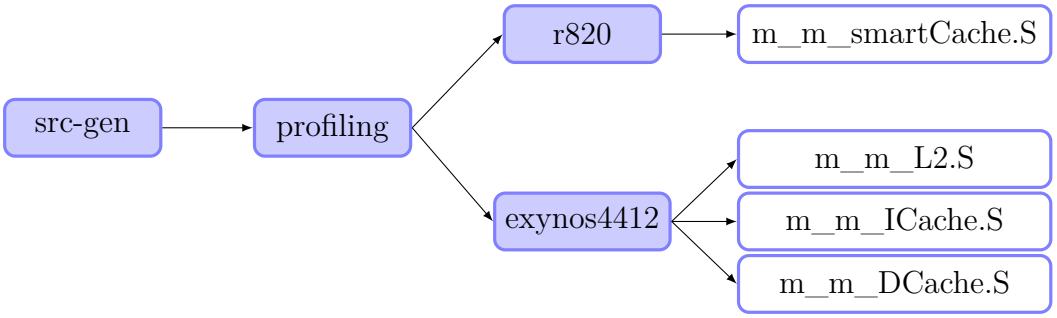

Figure 5.1: Hierarchy of the Output Files

5.2 Evaluation of target objectives

5.2.1 Runtime

The runtime of the project is analyzed under three different aspects:

1. External vs. Internal Code Generation
   The external code generation is a stand-alone solution. The generator is packed into an executable *jar* archive that is executed external. For the internal code generation the same code is executed from the Eclipse Launcher.

2. Dedicated vs. Embedded Generator Code
   The dedicated generator code is an Eclipse project which just contains the files of the *domain-specific language*s. In the embedded generator these files are part of a bigger project, the Operating System Code of *CyPhOS*. This scenario will test how much time the recursive resource search (section 4.1) will cost on large projects.

3. Reusing the precreated ISA File (subsection 4.2.1)
   Reusing the extended *ISA Description File* saves time. It is analyzed how much time it saves.

To measure the runtime on the aspects above, each aspect is combined with each other and analyzed for the generation of *ARM*-specific and *x86*-specific *Assembler Code*. Therefore there are $2^4 = 16$ different scenarios. It is measured how much time is used to **load** the used resources, to do the **preprocessing** and the **code generation** itself.

Table 5.1, Table 5.2, Table 5.3 show the results of the different measurements. Each aspect (internal, external, dedicated, embedded, reuse isa and create isa) is part of eight scenarios. The values in the table are the average times in milliseconds. Calculated values (Σ and saves) are calculated from non rounded values, therefore they might differ from calculating with rounded values.

	intern	extern	saves
load	890.45	8942.19	90.04%
pp	1186.78	2071.21	42.70%
gen	457.14	5636.16	91.89%
Σ	2534.36	16649.56	84.79%

Table 5.1: intern vs. extern

	ded.	emb.	saves
load	4953.34	4879.30	-1.52%
pp	1625.00	1632.99	0.49%
gen	3075.65	3017.65	-1.92%
Σ	9653.99	9529.94	-1.30%

Table 5.2: dedicated vs. embedded

	reuse	create	saves
load	4873.85	4958.78	1.71%
pp	1466.02	1791.96	18.19%
gen	3056.78	3036.51	-0.67%
Σ	9396.66	9787.26	3.99%

Table 5.3: reusing isa file

From Table 5.1 one can see it is obviously better to keep the generator integrated in Eclipse. This has to be considered when creating a plugin (subsection 5.3.3).

Against the assumption it takes longer to find files in a large project (see Table 5.2, row **load**), the dedicated generation is a bit slower. There is no difference where in the file hierarchy of the os code the *domain-specific language* files are. However the differences between the runtime of the dedicated and embedded scenario are so little that they can be said the same under consideration of a certain variance.

As expected the reuse of the extended *ISA Description File* (Table 5.3) saves runtime: 4% in average, 18% during the preprocessing. Running the *x86* code generator embedded in the operating system code intern from Eclipse saves 15% of total runtime and more than 39% during the preprocessing.

5.2.2 Complexity

A main target of the project is to reduce the complexity. It is hard to compare the costs of writing an *ISA Description File* to writing *Assembler Code*. To simplify the calculation of complexity some simple assumptions are made:

- all tasks implemented by *Assembler Code* (e.g. clearing the bss segment on startup or the profiling code (Listing 3.11 and Listing 3.12) are of same complexity
- implementing an *ISA Description File* for an architecture is as k as complex as implementing one task in *Assembler Code*.
- implementing a task in *Assembler Code* is as complex as implementing it in *Abstract Assembler Code*.

Let n be the number of different architectures and m be the number of tasks which have to be generated, then the complexity of writing *Assembler Code* for each of these architectures is $n+m$. If the code generation is used instead, the complexity is $k \cdot n + m$. The complexity of code generation is smaller than the complexity of handwritten *Assembler Code* if

$$k \cdot n + m < n \cdot m$$

For a constant number of architectures the code generation is profitable if

$$m > \frac{k \cdot n}{n - 1}$$

For a constant number of tasks the code generation is profitable if

$$n > \frac{m}{m - k}$$

It must apply that $m > k$ to achieve feasible results here, however this is obvious.

5.3 Future Work

Working with *domain-specific language*s offers a large variety of possibilities to implement. These are the expressiveness of the *domain-specific language* itself, the processing of the generated models and the IDE for the *domain-specific language*. In this section possible future work on each of these three aspects are discussed.

5.3.1 Extending the *domain-specific language*

Because the *domain-specific language* is set on top of the populated model, an extension of it means an extension of this model.

For now the *Architecture Description File* is able to differ between three types of hardware components: Caches, Memory Units and Processor Units. More component types (e.g. i/o) can be implemented in the future. (For now it is possible to use the keyword **io** but with no effect). The *Generator Configuration File* has to be extended for a more efficient code generation. For now just one file of *Abstract Assembler Code* can be processed at once. New types of configuration files can also be simply added with no big effort.

The expressiveness of the *Abstract Assembler Code* is very limited. It is not possible to use loops to execute an instruction or a group of instructions several times. Neither it is possible to use arithmetic options. These operations can become quite complicated to implement. It should be reflected on building the *Abstract Assembler Code* as an internal *domain-specific language* to use existing

language elements, but even then it is not common to use language elements like loops and conditions. The danger is to turn the *domain-specific language* to a *general-purpose language* and it is not necessary to create more expressiveness than the *Assembler Code* itself. The *Abstract Assembler Code* is directly translated to *Assembler Code* and in principle it is possible to translate it directly into machine code. (For debugging purposes this option is set aside.)

5.3.2 Extending the model

As described in the previous section, the extension of a *domain-specific language* already means the extension of the model. This section covers the aspects which can not be described by the grammar which is designed for the language like adding new methods to the metamodel and extending the generator.

A big aspect which can be described in the architecture model precisely but are not considered any further now are the links and buses between components. The research operating system *CyPhOS* needs detailed knowledge about these connections between components, but it is currently not possible to use these information while generating code.

To use these information it is probably not any more sufficient to generate some *Assembler Code*. When extending the generator it will become possible to generate any source code and transfer the hardware model into a C++ class hierarchy.

5.3.3 Creating an Eclipse Plugin

The framework *Xtext* (subsection 2.2.2) already supports the basic implementation of an Eclipse Plugin. The correct representation of the outline for the editors is mostly implemented by now, but there are still a lot of possibilities for features that can be implemented. Some are provided by the framework already such as the outline (structure as well as labeling), code validation (the syntax validation has been created by the grammar, the semantic validation is partly implemented but not exhaustive now) and a proposal provider. Other features can be implemented with all known tools from the Eclipse Plugin Development Environment. Sensible features will be wizards for new files or a group of files and a debugger. As described in former chapters the path of an abstract instruction to an assembler instruction is long, winded and it is possible to implement non deterministic instruction sets.

5.3.4 Performance

The runtime measurements show that it is necessary to integrate the generation process into the Eclipse environment. Different changes are not necessary. There are surly some parts of the code that can be optimized but this will only speed

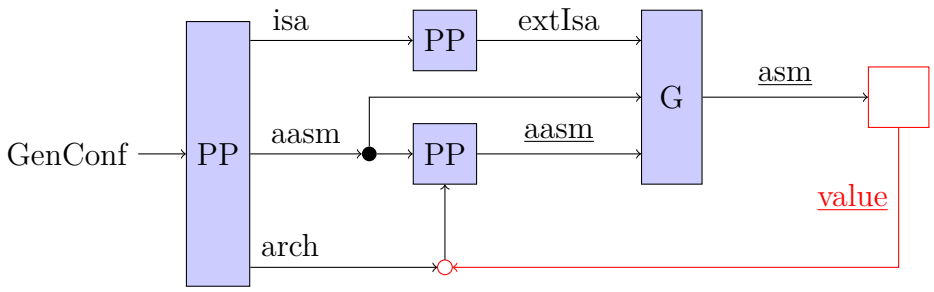

Figure 5.2: Backpropagation of operation results

up the generation process for some milliseconds. The code generation is not used anywhere at runtime, but within the compiling process of operating system code. The average time for the code generation (internal) is about 2.5 seconds on a 32 core machine (Intel Xeon CPU E5-4640 at 2.4 GHz clock speed). As a comparison the build of a minimal configured Linux Kernel takes about 1 minute on the same machine. A save of some milliseconds in the generation process will not change much about it.

5.3.5 Further aspects of future work

To avoid the usage of instructions that have not been defined in the instruction set (which is hard to validate, because the *ISA Description File* and the *Abstract Assembler Code* are on the same layer and do not know each other) this validation has to be made on the *Generator* level. A simple validation is already made, but it only respects the keywords of the instruction, not the parameters. The missing instructions are neither marked in the *Abstract Assembler Code* editor. As a possible feature a list of subsets of instructions which are defined in every *ISA Description File* can be created to find out which instructions are safe to use in the *Abstract Assembler Code*.

The research operating *CyPhOS* needs information about the access time on memory. For this purpose profiling code has been implemented in the *Abstract Assembler Code*. In the future, the results of such measurements have to be automatically fed back into the hardware model. A possible solution (Figure 5.2) is added to the generator interaction graph of Figure 4.2. When executing the *Assembler Code* (e.g. time measuring), the resulting values (measured time) is inserted to the architecture model and can be used for further processing.

Bibliography

[1] ARM. *Cortex - A9. Revision r2p2*. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f/DDI0388F_cortex_a9_r2p2_trm.pdf#page=25.

[2] *ARM: Introduction to ARM: Registers*. Mar. 2012. URL: http://www.davespace.co.uk/arm/introduction-to-arm/registers.html.

[3] ARM Ltd. *Cortex-A9 Processor*. URL: https://www.arm.com/products/processors/cortex-a/cortex-a9.php.

[4] H. Borghorst and O. Spinczyk. "Increasing the Predictability of Modern COTS Hardware through Cache-Aware OS-Design". In: *Proceedings of the 11th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '15)*. (Lund, Sweden). July 2015. URL: http://www.mpi-sws.org/~bbb/events/ospert15/pdf/ospert15-p41.pdf.

[5] *Device Tree - What it is*. Dec. 2016. URL: http://elinux.org/Device_Tree_What_It_Is.

[6] *Devicetree Specification*. May 2016. URL: http://www.devicetree.org/specifications-pdf.

[7] *Documentation/TCG/frontend-ops*. Oct. 2016. URL: http://wiki.qemu-project.org/Documentation/TCG/frontend-ops.

[8] *EMF Tutorial*. June 2016. URL: http://eclipsesource.com/blogs/tutorials/emf-tutorial/.

[9] M. Fowler and R. Parsons. *Domain specific languages*. Upper Saddle River, NJ: Addison-Wesley, 2011.

[10] *Introduction to x64 Assembly*. Mar. 2012. URL: https://software.intel.com/en-us/articles/introduction-to-x64-assembly.

[11] *Language Engineering For Everyone*. URL: http://www.eclipse.org/Xtext/.

[12] *MPC8572 Development System User´s Guide*. Jan. 2009. URL: http://www.nxp.com/assets/documents/data/en/user-guides/MPC8572DSUG.pdf.

[13] *QEMU's recompilation engine*. Slides no longer available, talk on https://chemnitzer.linux-tage.de/2012/vortraege/1062. 2012. URL: https://dl.dropboxusercontent.com/u/8976842/TCG.pdf.

[14] N. Rochester. "The 701 Project as Seen by its Chief Architect". In: *Annals of the History of Computing* 5.2 (Apr. 1983). URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4640454.

[15] *Xtend -Documentation.* URL: http://www.eclipse.org/xtend/documentation/index.html.

[16] *Xtend -Documentation.* URL: https://wiki.eclipse.org/Xcore#Modeling_for_Programmers_and_Programming_for_Modelers.

[17] *Xtext - A powerful tool for language engineers.* URL: https://www.itemis.com/en/xtext/.

List of Figures

Listingverzeichnis

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift