TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Information Technology

MIKAEL LEPISTÖ

ASSEMBLY COMPILER FOR PARAMETRIZABLE PARALLEL PROCESSOR

Master of Science Thesis

Subject approved by Department Council
11th November 2005

Examiners: Prof. Jarmo Takala
Doc. Pertti Kellomäki

PREFACE

This M.Sc thesis was completed in Institute of Digital and Computer Systems of Tampere University of Technology (TUT) in 2005-2006 for codesign software implemented as part of the Flexible Design Methods for DSP Systems (FlexDSP) project funded by the National Technology Agency.

I would like to thank Professor Jarmo Takala for giving me a chance to be part of this interesting project and Docent Pertti Kellomäki for his valuable comments and improvement ideas for this thesis. I am also extremely grateful to PhD Andrea Cilio for always being ready to discuss the problems and for sharing suggestions how to improve the compiler during the project. I would also like to thank all the people in TCE project for making the relaxed and joyful atmosphere during the years.

I am also grateful to my friends, for always helping me out when I had too much spare time. Finally, I would like to thank my family for their support throughout my life.

Tampere, May 11, 2006

Mikael Lepistö

TABLE OF CONTENTS

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Information Technology

Institute of Digital and Computer Systems

Lepistö, Timo Mikael: Assembly Compiler for

Parametrizable Parallel Processor

Master of Science Thesis: 55 pages, 4 appendix pages

Examiners: Prof. Jarmo Takala and Doc. Pertti Kellomäki

Funding: The National Technology Agency

November 2006

Keywords: transport triggered architecture, assembly compiler

Often in embedded systems is a need to develop complex applications with hard performance and energy-efficiency requirements. This can be achieved with customizable processor architectures, which can be programmed with high-level languages. Application-specific instruction set processors (ASIP) are one approach to respond to the need. However customizing the processor for a specific task must be automatized as far as possible to make the design process cost-effective.

Only a few tool-sets have been developed to assist the design process. TTA-Based Codesign Environment (TCE) is one, being developed in Tampere University of Technology. TCE aims to provide semi-automatic design space exploration for finding the processor, which has the best trade-off between area, speed and power efficiency for the application. TCE exploits the Transport Triggered Architecture (TTA) paradigm where the resources of a processor, like function units, register files, operations and buses can be customized freely. The programming of the architecture is done by defining connections between the resources and the busses.

For this thesis, an assembly compiler for TCE was developed. The compiler allows developers to create parallel TTA programs for a freely chosen processor configuration as well as optimizing already parallelized programs. The thesis introduces the syntax of the assembly language and describes the main design and the maintenance issues. The syntax of the compiler is designed to be as generic as possible, so the same assembly program can be used for slightly different processor configurations. Finally, the thesis describes how the functionality of the compiler was verified.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO

Tietotekniikan koulutusohjelma

Digitaali- ja tietokonetekniikan laitos

Lepistö, Timo Mikael: Assembly Compiler for Parametrizable Parallel Processor

Diplomityö: 55 sivua, 4 liitesivua

Tarkastajat: Prof. Jarmo Takala ja Dos. Pertti Kellomäki

Rahoitus: Teknologian kehittämiskeskus (TEKES)

Toukokuu 2006

Avainsanat: transport triggered architecture, assembler kääntäjä

Usein sulautetuissa järjestelmissä on tarve kehittää ohjelmallisesti monimutkaisia sovelluksia, joilla on kovat tehokkuus sekä virrankulutus vaatimukset. Tämä voidaan saavuttaa räätälöitävillä prosessoriarkkitehtuureilla, joita voidaan ohjelmoida lausekielillä. Sovelluskohtaiset käskykantaprosessorit (ASIP) on yksi lähestymistapa ongelman ratkaisuun. Prosessorin räätälöinti täytyy pystyä automatisoimaan, jotta suunnittelu prosessi saadaan mahdollisimman kustannustehokkaaksi.

ASIP arkkitehtuureille on tehty muutamia kehitystyökaluja ja yksi niistä on TTA-Based Codesign Environment (TCE), jota kehitetään Tampereen Teknillisessä Yliopistossa. TCE tähtää puoliautomaattiseen suunnitteluavaruuden läpikäymiseen, jotta sovellukselle löytyisi prosessori, joka tarjoaa parhaan kompromissin tehokkuuden, pinta-alan ja virrankulutuksen suhteen. TCE hyödyntää transport triggered -suoritinarkkitehtuuria (TTA), jossa prosessorin resurssit, kuten laskentayksiköt, rekisterit, operaatiot ja väylät voidaan valita vapaasti. Tällaista prosessoria ohjelmoidaan määrittämällä resurssien ja väylien välisiä kytkentöjä jokaiselle kellojaksolle.

Tässä diplomityössä suunniteltiin, sekä toteutettiin symboolinen konekielikääntäjä TCE työkalupakettiin. Kääntäjä mahdollistaa rinnakkaisen TTA -ohjelman kääntämisen vapaasti valittavalle prosessorikonfiguraatiolle, sekä valmiiksi rinnakkaistetun ohjelman optimoinnin. Työssä kuvataan kielen syntaksi, esitellään kääntäjän arkkitehtuuri ja opastetaan miten kääntäjään voidaan lisätä uusia ominaisuuksia TCE:n kehittyessä. Kielen syntaksi on määritelty mahdollisimman riippumattomaksi prosessorikonfiguraationsta, jotta ohjelmakoodiin tarvitsee tehdä mahdollisimman vähän muutoksia prosessoria muokattaessa. Lopuksi työ kuvaa miten kääntäjän toiminnallisuus varmennettiin.

LIST OF ABBREVIATIONS

ADF	Architecture Definition File
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction set Processor
ASP	Application-Specific Processor
CISC	Complex Instruction Set Computer
DSP	Digital Signal Processor
EBNF	Extended Backus-Naur Form
FFT	Fast Fourier Transform
FU	Function Unit
GCU	Global Control Unit
GPP	General-Purpose Processor
GPR	General-Purpose Register
HLL	High-Level Language
IDE	Integrated Development Environment
IU	Immediate Unit
MAU	Minumum Addressable Unit
MOM	Machine Object Model
POM	Program Object Model
RF	Register File

RISC	Reduced Instruction Set Computer
STL	Standard Template Library
TCE	TTA-Based Codesign Environment
TPEF	TTA Program Exchange Format
TTA	Transport Triggered Architecture
VHDL	Very high speed integrated circuit Hardware Description Language
XML	Extensible Markup Language

1. INTRODUCTION

The complexity of software in embedded applications has been increasing all the time, together with increased performance and energy efficiency requirements. Conventionally, the application domain has been covered with combined application specific integrated circuit (ASIC) and digital signal processor (DSP) solutions. A DSP can be programmed with high-level languages (HLL), which handles the complex software, and ASIC designs implement the parts that meet the critical performance requirements. Designing combined ASIC and DSP systems is highly time consuming and expensive because of the ASIC design process. Moreover a DSP increases the production costs and reduces the power efficiency of the products, because of the extra functionality of a DSP, which cannot be exploited by the application.

Customizable application-specific instruction set processors (ASIP), which can be programmed with HLL, responds to the problem by cutting down production costs and by increasing power efficiency. With an ASIP it is possible to select a specific instruction set for the applications that are run in the processor, and to leave out all the functionality that is not needed. This reduces the size and makes the product more cost-effective. Further, ASIP allows developers to design special instructions for the processor, which makes it possible to run specific tasks very efficiently. Therefore ASIP reduces the need for ASIC designs.

Transport Triggered Architecture (TTA) is a modular parameterizable ASIP architecture that aims to shift complexity from hardware to software by having very complex instructions. TTA provides way to tailor the processor configuration, including function units, buses and register files, so the application will run efficiently enough with low power requirements. However, designing an optimal TTA processor for the application is hard and time consuming, since there can be hundreds of suitable configurations, and each time when the processor configuration is changed the program must be changed as well.

Development toolsets are needed to assist the design process. TTA-Based Codesign Environment (TCE) is one, being developed in Tampere University of Technology.

TCE aims to provide semi-automatic design space exploration for finding the processor which has the best trade-off between size speed and power efficiency for the application. TCE and especially compiler algorithms for the processor are still in development. Therefore, reference data for evaluating the developed algorithms are needed. An assembly compiler is a tool that can be used for creating reference implementations of programs. Efficiency of the compiler algorithms can be compared with the hand written assembly code.

In this thesis, TCE Assembler was developed, which allows developers to write parallel code for the TCE TTA processors. Further the assembler can be used for hand optimizing already parallelized programs. This thesis acts as a user manual and maintenance guide for TCE Assembler.

The thesis first briefly introduces the TTA architecture and file formats of TCE utilized by the compiler in Chapter 2 and continues by describing the syntax of the assembly language in Chapter 3. The fundamental design and maintenance issues as well as the compilation process are described in Chapter 4. Further the chapter introduces some suggestions for the future development, and emphasizes the main issues of the external libraries that the assembler utilizes. Chapter 5 describes how the functionality of the assembler was verified. This thesis is summarized in Chapter 6 by presenting the conclusions.

2. TRANSPORT TRIGGERED ARCHITECTURE

Transport Triggered Architecture (TTA) is a modular and customizable processor architecture, which provides an easy way to create application-specific processor (ASP) designs. TCE TTA template is a TTA that is specifically designed for TCE tools. A detailed overview of TCE tools is presented in [1].

Conventional processor architectures like RISC and CISC have a fixed set of resources available for the programmer. These architectures usually have only one general-purpose data bus, one address bus, some register files and some functional units that contain operations that processor can execute. Data paths of these processors are quite complex but fixed. In addition to the connection to the main data bus, the elements usually have connections between each other. For example, functional units may access register files directly so all the parameters for operations do not have to be transferred through the same bus. These extra buses make it possible to do complicated data transfers without consuming multiple clock cycles. For instance, a processor can read all operands for an instruction directly from registers and write the result back to some other register all in one clock cycle. With these processors, each instruction controls all the connections between elements and defines how the data is transferred on the data path of the processor.

For a programmer, conventional processors offer only a software interface in the form of an instruction set. This interface hides all the implementation details of a processor, like caches, instruction pipelines and internal buses. Each instruction in these processors is a small program, that does a sequence of things inside a processor resulting the required changes to the state of the processor after the execution of instruction is completed. The programmer never knows what was actually done inside the processor during the execution, but only the results of the execution.

With parametrizable TTA processors, the situation is completely different. TTA is not a fixed architecture and a designer can choose the best processor configuration for the application in hand. Basically, a TTA processor consists of multiple buses, register files, function units and connections between them. All the data needed for an operation is transferred through the buses. If a processor contains only one bus, it means that if an operation needs two operands from registers and writes the result back to a third register, it needs three clock cycles to execute. However, TTA processors rarely have only one bus. With three buses, the whole task can be executed in one cycle. Depending on the application it is possible to choose a processor configuration with the best trade-off between, speed, power consumption, and size of die. For instance, if there is a need to perform six multiplications at the same time, one can choose a processor configuration to have enough buses and multipliers. An instruction in a TTA processor defines all the data transfers through the buses for one clock cycle.

The transport triggered operation principle is based on data moves to the ports of function units. When data is written to the normal ports of a unit, the values are only read to input registers. Execution of an operation is launched when data is written to a triggering port of the unit. Depending on the length of the pipeline of the function unit, the results are written to the output registers in one or more clock cycles. Conversely, if a function unit is not triggered, it will not do computation and the unit stays in idle mode.

This chapter first briefly describes the structure of TCE TTA template from the point of view of a programmer in Section 2.1 and continues by explaining the programming of TCE TTA template in Section 2.2. Finally, the chapter introduces the file formats related with compilation in Section 2.3. More detailed information on TTA is described in [2] and TCE TTA template in [3].

2.1 Template Architecture

Figure 1 introduces the basic structure of a simple TCE TTA template processor. The architecture has four different type of units: immediate units, register files, function units and a global control unit. All units contain ports, and the units are connected together with buses and sockets. The processor in Figure 1 consist of three buses, two function units, a register file, an immediate unit and a global control unit.

2.1.1 Register File

Register files (RF) are arrays of registers that have the same bit width. Registers are used to store temporary values inside the processor. A RF may contain multiple ports

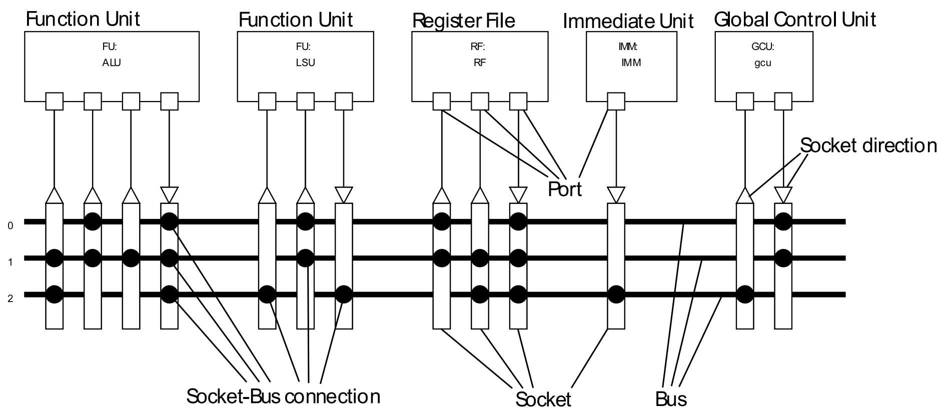


Figure 1. *Simple TCE TTA template processor.*

for reading and writing its registers. In that case, the RF can be accessed multiple times in one clock cycle.

2.1.2 Immediate Unit

Immediate units (IU) are a special type of register files which contain registers where long immediate values are loaded during instruction decoding. The value of an immediate is stored inside an instruction and it is read from the program memory together with the instruction. These units may only contain read ports, since the writing of the registers is never done through the ports of the unit. Long immediates are described in more detail in Section 3.1.4.

2.1.3 Function Unit

Function units (FU) contain all the operations that the processor can execute. The operations in TTAs can be chosen freely so there can be very complex special operations, for instance, address calculation operations or complex multiplication, in addition to conventional operations like addition and multiplication. Input parameters and results for the operations are read and written through the ports of a function unit.

Since a single function unit may contain multiple operations, in addition to the normal ports which are used to write input data and to read the results of an operation, there are two types of special ports: opcode ports and triggering ports. These special ports are used to select, which operation is executed, and when the execution is started. A write to an opcode port always contains extra information for selecting the operation to be executed and it also starts the execution of the selected operation. Writing to a triggering port starts an execution of the last selected operation. If a FU contains only one operation, no opcode is needed.

2.1.4 Bus

Buses are used to transfer data between all the units. Each bus may also contain guards, which makes it possible to have conditionally executed data transports. The value of a guard defines if the data transport through the bus is actually executed. For example, a conditional jump can be implemented by writing a jump operation through a bus which is guarded. Guards are described in Section 3.1.4.

2.1.5 Socket

Sockets are the connection between buses and ports of units. Each connection to a bus can be set or unset, depending how the data is transported inside a processor. The software of TTAs is basically instructions how the data is moved inside a processor, in other words the programming of the TTA processor is actually defining socket connections for each clock cycle.

Since a single socket may be connected to many different buses, it is possible to read a value from a port to many buses through one socket. However writing a value from many buses to one port is not possible.

2.1.6 Global Control Unit

The global control unit (GCU) is a special function unit that controls instruction fetching and decoding. In each TTA processor, there is only one GCU which controls all socket connections and other functions that are hidden from the programmer. This includes writing long immediates to immediate registers and selecting the guard that used for each bus.

Since the GCU is responsible for instruction fetching, it also contains the program counter, which is basically the only visible special property of GCU for the programmer. Because of program counter, all the control flow operations, like *jump*, must be in GCU.

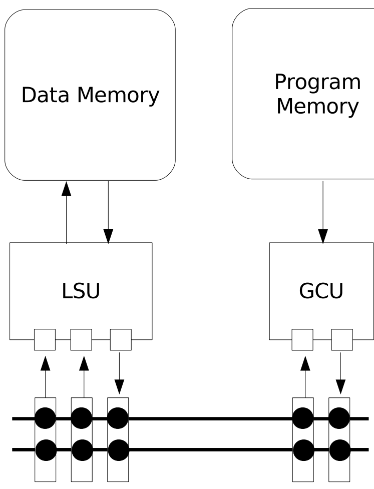


Figure 2. *Processor Containing Two Address Spaces.*

2.1.7 Program and Data Memories

TCE TTA has a separate program and data memories: a program memory is accessed by GCU and data memories are accessed from the function units, which contain memory operations. Figure 2 shows a simple processor configuration where is only GCU and load/store unit, that contains operations for writing and reading the data memory.

A processor configuration contains address space definitions, which describe the characteristics of the memories of a processor. The processor in Figure 2 contains two address spaces, one for data and one for the program. Address spaces define the starting and ending addresses of the memories and the number of bits that are stored in each memory address. This bit width is the minimum addressable unit (MAU).

2.2 Programming TCE TTA Processor

TTA processors are programmed by defining data transports between the ports of the units. One data transport defines a connection from a source port through a bus to a destination port. With a processor containing multiple buses, a set of simultaneous data transports in one clock cycle is an instruction of the processor. In addition to the data transports, an instruction may contain also long immediate assignments.

Instruction templates are definitions of what kind of data transports and long immediate assignments can occur in one instruction. A processor can contain many different instruction templates. For example, a machine containing four 16-bit buses could use the following templates; the first template requires four data transports and the second template requires two data transports in the first and third bus and one 32-bit long immediate assignment.

Often when data is read from a port or written to it, the transfer needs additional information to pass to a function unit or a register file. The extra information defines the register number or the operation of the function unit is accessed. Information is transferred by passing an additional opcode for the source and/or destination of data transport.

The trigger port must not be written before all the other operands are ready for operation. For example, with substraction both operands must be written in the same clock cycle, through different buses or the operand that is not triggering must be written first and after that the triggering operand.

Further information about the programming of TCE TTA template is found in [1] and [4].

2.3 Compilation Source and Result Formats

When assembly code is compiled for a specific processor configuration, the compiler needs to know the characteristics of the processor. This description of a processor configuration is stored in an Architecture Definition File (ADF) [3]. ADF is an XML file, which describes all the information, that is visible for the programmer about the processor. However, ADF does not describe implementation details of the processor, e.g., how the multiply operation of the processor is implemented in VHDL.

In addition to FUs, RFs, IUs, sockets, and buses, ADF describes the address spaces and instruction templates of a processor. The characteristics are described in a such detail, that it is possible to simulate the functionality of a processor, based on the processor description stored in ADF.

As illustrated in Figure 3, the compiler uses the Machine Object Model (MOM) library for reading ADF. MOM provides an object representation of the processor, which makes accessing ADF resources easier. During compilation, the assembler verifies, that all the resources used in the assembly code are found in MOM.

Figure 3 further shows that the compiler writes its output to a TTA Program Exchange Format (TPEF) [5] file, that is a binary file format for storing all the information about a TTA program. TPEF Library provides an object model of TPEF. The compiler uses

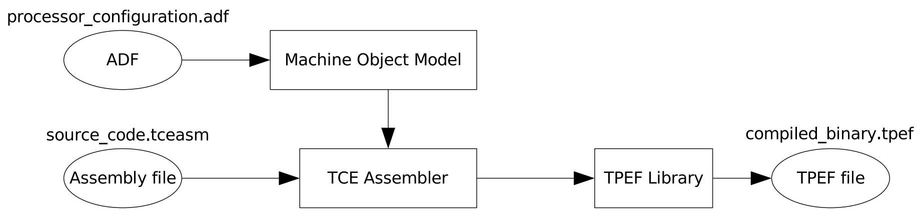


Figure 3. *Sources and Results of Compilation.*

the object model for accessing and creating a TPEF program and after compilation, TPEF Library writes its object hierarchy to a TPEF file.

Usually executable binary formats contains information, like a code section, data sections, labels, relocations, and debug data about programs. In addition to this TPEF also contains the names of the ADF resources that are used by the program and information about the address spaces that the processor contains.

TPEF files are specific for one ADF only. For example, if TPEF program is simulated with a different ADF, that was used for generating the program, the results will be unexpected.

3. USING COMPILER

There are basically two different ways to use the assembly compiler. For development tool developers, there is a library that can be used, for instance, by an IDE to compile assembly code without the need to call any external shell programs. The interface of the compiler library is described in Chapter 4. For those who write assembly code for TCE TTA template processors, there is a command line interface named *tceasm*.

Using the assembler is very straightforward, it reads ADF and assembly files and outputs the result to a TPEF file. Furthermore, the compiler does extensive error checking to find errors and suspicious areas in assembly code and outputs detailed information about the problem.

Writing an assembly program fundamentally requires knowledge of two things: the architecture definitions of the processor, and the syntax of the assembly language. The architecture defines all the resources of the processor such as buses, registers, memory areas, and the instruction set. The assembly syntax defines how to refer to the machine resources in assembly code and provides a way to define programs for the processor.

This chapter describes first the syntax of the assembly language in Section 3.1 and continues by discussing the disambiguation rules of the syntax in Section 3.2. Naming requirements for ADF resources and special operations are introduced in Section 3.3 and the behavior of the command line interface *tceasm* is presented in Section 3.4. Finally, the error messages of the compiler are listed in Section 3.5.

3.1 Syntax of Assembly Language

The compiler understands only a subset of the syntax described in [4], because the specification contains some additional features, which were not required for the first version of the compiler. Complete grammar of the compiler is described in Appendix B. The syntax is designed to be as user friendly as possible and especially in a way that the program for one processor configuration can be modified for a slightly different processor configuration with as few changes as possible. Thus the assembler

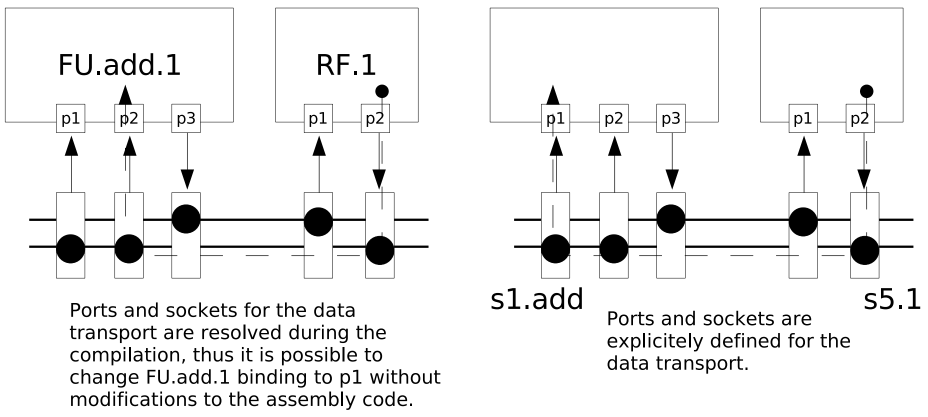

FU.add.1
RF.1
p1
p2
p3
p1
p2
Ports and sockets for the data transport are resolved during the compilation, thus it is possible to change FU.add.1 binding to p1 without modifications to the assembly code.
p1
p2
p3
p1
p2
s1.add
s5.1
Ports and sockets are explicitely defined for the data transport.

Figure 4. *Two Different Ways How to Define Data Transport.*

syntax does not define data transports by describing socket connections to the buses, but defines required data transports between units. As illustrated in Figure 4 this is done by referring to actual resources, like the registers and the operations of the units and describing the source and destinations where data should be read from and where it is transported to. This way it is possible, for instance, to add ports to the units and change the operation bindings of the ports of a processor without a need to modify the assembly code.

This section first describes basic rules, how white space and comments are interpreted, and continues by introducing the highest level of the structure of assembly code. Next the common syntax definitions for the whole assembly code are described, and finally the syntax for writing the code section and the data sections are presented. An example of a program written for the compiler is listed in Appendix A.

3.1.1 Comments and Logical Lines

An assembly program listing consists of logical lines. Each logical line is terminated by a semicolon, and can contain extra white-space characters or comments to improve readability. Therefore, each logical line may spread over one or more physical lines of an assembly file. Conversely, each physical line may contain multiple logical lines. Physical lines are actual text lines that are terminated by an end of line character. The ending character of physical line may differ on different platforms. A comment start with a hash character and continues until the end of physical line. The assembly syntax does not contain block comments, that may spread over multiple physical lines.

3.1.2 Section Header

In the highest level, assembly code is divided in sections. Sections represent memory areas of the program, and all the instructions and data have to be defined inside some section. Even the command directives must be defined in a section. However, there can be comment lines and white spaces before any section definition.

The assembler supports two different types of sections: code and data. These sections differ by contents, the code area contain program instructions and data sections contain data area definitions for a program. Unlike many general purpose processor assemblers, the TCE assembler does not support shared memory for code and data, so it is strictly forbidden to mix instructions and data definitions in the same section or the address space.

A section start header adopts one of the following formats:

```
CODE [start-address];
DATA address-space-name [start-address];
```

The *address-space-name* field defines to which address space a currently introduced section belongs. A code section header does not need the parameter, due to the fact that TCE TTA template does not support multiple address spaces for code. The code address space is defined unambiguously by the address space of GCU which defined in ADF.

The *start-address* field is an optional unsigned value that may be given in a binary, a decimal, or a hexadecimal base. Its value defines the starting address of the section in hand. If the parameter is omitted, the compiler calculates the first unused address of the address space and sets the section to start from that address. This parameter is hardly ever needed, since the compiler always knows the optimal starting address. However, one case when the explicit address might be a good idea is when data should be placed to start from a specific memory page of an address space.

There are basically two restrictions how the starting addresses of sections can be selected:

1. Each new section must start from a higher address than the last address of previous section that was defined for the same address space.
2. There must be enough free addresses in the address space after the start of the section to hold all defined instructions or data areas of the section.

It is completely valid to write a program that contains only data sections or a code section. It is sufficient that there is only one section in a program. There can be many data sections concerning the same address space as well. If the addresses of the data sections are explicitly defined, then the rules above must be followed.

3.1.3 Common Behavior

The assembly syntax has common of representing constants, names and command directives. This section describes the main rules of what kind of label names can be used in assembly code and all other syntax definitions that are not specific for each section type, but apply to both the code section and data sections.

Constant Literals

Constant literals are used where constant values, which are not addresses, are written in a program. The most common cases of this are data area initializations in data sections and immediates in the code section.

The assembler supports three ways of presenting constants: binary, decimal, and hexadecimal forms. Binary literals start with prefix "0b", hexadecimal with "0x" and decimals without any prefix.

All binary, hexadecimal and positive decimal values are interpreted as unsigned integers. Only negative decimal values are handled as signed. The only thing that differs between signed and unsigned representation is the extending. Unsigned values are always zero-extended and negative decimals are sign-extended.

Table 1. *Maximum Values of Constant Representations.*

Literal Type	Maximum Value
binary:	0b11111111111111111111111111111111
signed decimal:	-2147483648
unsigned decimal:	4294967295
hexadecimal:	0xffffffff

Because of the default implementations of the literal parsers of the Spirit library, the maximum value that can be given in one constant literal is 32 bits, which means that these are the limits for different presentations. As Table 1 shows the decimal representation has value range from -2147483648 to 4294967295 which seems to be more than 32 bits. This is possible, because of binary representation of unsigned decimal overlaps partially with signed decimal representations. So actually if one writes 4294967295 the value has the same binary representation as 32-bit wide -1.

Allowed Names

Unlike the most assemblers, TCE Assembler is case sensitive, with one exception. Operation names are case insensitive and can be written as the user decides. Therefore, all labels and references to processor resources like function units, register files, and ports, must be written exactly as their names are defined in ADF.

The allowed characters to use in names are letters, numbers and underscore. Each name must start with an underscore or a letter. The formal representation of allowed names with regular expression is: [a-zA-Z_][0-9a-zA-Z_]*. ADF also allows to use colons in names, but the assembly syntax does not support compiling programs for an ADF that uses colons in names.

Labels

Labels are special constants referring to memory addresses, e.g., constant values for jumps or memory load operations must be given as labels. Labels are used as aliases for the referenced addresses. These aliases make the program more readable and especially allow the assembler to differentiate memory address references from plain constant values.

The ability to tell the difference between constants and addresses is essential for creating relocatable code [6], which can be placed to start from a freely selected address in an address space. This is required, for instance, when multiple separately compiled programs are loaded in memory for execution. The information whether the constant values are addresses or not is also needed when the real program image is generated from a TPEF file.

The memory range, where a program can be loaded, depends on the width of the address field reserved for address constants. The width of address fields is decided by the size of the referenced address space of ADF. Therefore, labels referencing to different address spaces might have different bit widths. For example, if an ADF contains an address space that starts from 0 and ends at 65535 then all the references to this address space are 16 bits wide. However, the field might be smaller, e.g., if an inline immediate does not have enough bits for representing 16-bit values.

Because labels are the only way to treat constant values as addresses, all references to addresses must be done with labels. Basically there are two different situations when labels are used in a program: declarations and references. Declarations can be added before any instruction or data area declaration and are terminated with a colon. The allowed format for *label-name* is described in this section earlier.

Label declarations are introduced by the following format:

```
label-name:  instruction or data area definition ;
```

Label names must be unique in the program. However, it is valid to have multiple labels referring to the same address, for example,

```
mainStart:
loopPoint:
loopPoint -> gcu.jump.1 ; # infinite loop
```

In this case, *mainStart* and *loopPoint* labels refers to the same instruction address.

Label declarations for data addresses are written as in the following example:

```
dataStruct1:  DA 16;
```

When a program refers to a label, it is called an expression. In addition to a plain label, an expression can contain an offset and a numerical representation of the referred address. A numerical address is only additional information that is verified to be consistent with the address resolved by the compiler.

Expressions are defined by the following regular expression:

```
label-name((+|-)unsigned-literal)?(=unsigned-literal)?
```

In the above regular expression, the *unsigned-literal* fields are either binary, unsigned decimal, or hexadecimal literal. Only the *label-name* field is obligatory. Therefore, the offset and validation parts can be left out.

In the following list, there are some examples of expressions, on condition that *label1* refers to in the address 0xb:

```
label1          ->  gcu.jump.1 ;  # jumps to address 11
label1=0xb      ->  gcu.jump.1 ;  # jumps to address 11
label1+1        ->  gcu.jump.1 ;  # jumps to address 12
label1+0x2      ->  gcu.jump.1 ;  # jumps to address 13
label1+4=0xf    ->  gcu.jump.1 ;  # jumps to address 15
label1-0b1011=0 ->  gcu.jump.1 ;  # jumps to address 0

label1+100=1    ->  gcu.jump.1 ;  # compiletime error
```

Command Directives

Command directives are instructions for the compiler, providing additional information about the program. All directives begin with a colon followed by the name of the directive and are terminated by a semicolon. However, the parameters for each directive are individual. Currently, the assembler supports two directives: *global* and *procedure*.

Global declares a label to be globally visible. Global visibility means that a code or data label can be used by other compilation modules. This makes possible, for instance, to compile library code and use it in other programs without a need for recompilation.

The directive is followed by one mandatory parameter, which is the label name that is made globally visible. The label that is set to be global must be defined in the same assembly file where the global directive is set. The following example illustrates how to use the directive:

```
:global _emptyFunction ;
_emptyFunction:  gcu.ra -> gcu.jump.1;
```

The *procedure* directive defines the starting point of a new procedure and defines a name for the procedure. The procedure directive is followed by one parameter, which defines the name. The format of valid names is described in this section earlier. In practice, the procedure names are needed by programs to make the debugging of assembly code easier, apart from that a parallel program have no need for the procedure information. It is possible to use the same name for a procedure and for a label in the same program as shown in the following example:

```
:procedure main ;
main:
gcu.ra -> gcu.jump.1 ;
```

Even though it is not necessary to have a label with the same name as a procedure, it is very common also to have a label. The main reason for this is that procedure names cannot be used as jump addresses like labels.

3.1.4 Code Section

A code section contains all the instructions of a program. The whole code section consists of instructions and directives. Each logical line defines either a directive or an instruction. There can only be one code section in a program and it starts with a code section header that is described in Section 3.1.2.

This section begins by describing how different machine resources of ADF are referred to in assembly code, and continues by describing the syntax for a single data transport in a bus. Finally, the section describes the whole syntax of instructions.

General-Purpose Register

General-purpose registers (GPR) are the registers of register files in an ADF. In the assembly syntax, general-purpose registers are referred to with a register file name, a register number, and optionally with a port that should be used for accessing the register file. General-purpose registers are referred to by the following format:

```
reg-file-name[.port-name].index
```

The *reg-file-name* field describes the name of the register file whose register is referenced. The *index* field is a positive decimal value, indicating which register of the register file is referenced to. The *port-name* field is an optional parameter, if a specific port is explicitly wanted to select for accessing the register file.

Long Immediate and Immediate Unit Register

Long immediates are immediates which are first assigned to a register of an immediate unit and then referred in assembly code with the same syntax that GPRs. The main difference between the registers of an immediate unit and registers of a register file is that registers of an immediate unit can only be read in code, writing is not allowed.

Assignment of a long immediate register is defined by the following syntax:

```
[imm-unit-name.index=expression-or-constant]
```

The *imm-unit-name* field defines the name of the immediate unit where to value is written. The *index* field defines the index of the register, that is used for storing the long immediate value. The *expression-or-constant* field is the value for an immediate and it can be either a plain constant literal or an expression, which are described in Section 3.1.3.

Function Unit Port

There are three ways to refer to function unit ports in code. Sometimes the type of reference depends on situation but mostly it is up to the user to decide which alternative is used. All three possibilities can be expressed by the following formats:

```
fu-name.port[.opcode]
fu-name.operation.operand-index
```

In the first syntax variant, the *fu-name* and *port* fields are the names of the function unit and the port to be accessed. The *opcode* field is an optional part that is needed only when a port which triggers the operation is written. The *opcode* field is the name of the operation to be executed.

In TCE, operations are defined to have a certain numbers of input and output operands. These operands have indices that can be used to refer them. Indices are bound to the real ports of function units in ADF. This makes it possible to use a consistent way to refer the same operations, which are located in different function units with different port bindings.

In the second syntax variant, there are three mandatory parts and it always contains information about the operation in addition to accessed port. In this syntax variant, the *fu-name* field is the name of the function unit and the *operation* field is the name of the operation whose operand is accessed. The *operand-index* field is the index of operand that is accessed. For example, for addition *add.1* is the first input operand, *add.2* is the second input operand and *add.3* is the result output operand. The compiler resolves the real function unit ports that are used for accessing the operands.

For instance, say that there is a *ldw* operation in two function units, *FU1* and *FU2*. In *FU1*, the operands are bound to the ports *P1* and *P3*, and in *FU2*, the operands are bound to the ports *P3* and *P4*. Now with the second syntax variant *ldw* operation of both function units can be accessed the same way instead of port references as shown in Table 2.

Table 2. *Equivalent Function Unit References With Both Syntax Variants.*

Syntax Variant 1	Syntax Variant 2
FU1.P1.ldw	FU1.ldw.1
FU1.P3.ldw	FU1.ldw.2
FU2.P3.ldw	FU2.ldw.1
FU2.P4.ldw	FU2.ldw.2

All the different types of function unit references are shown in the following example:

```
gcu.jump.1
alu.p3
alu.p3.add
```

Move

A move is the fundamental element of TTA code. It defines one data transport through a bus. A move consists of three parameter fields: source, destination, and guard. A move is defined by the following format:

```
[guard] source -> destination
```

The two mandatory parts of the move are the *source* and *destination* fields. The *source* field defines where data is read from and it can be either a general-purpose register, a function unit port, a long immediate register or an inline immediate. The *destination* field defines where data is written to. In contrast to the *source* field, the *destination* field can only be a general purpose register or a function unit port.

Inline immediates are constant values which are encoded inside the source field of the move. Therefore, these values are fetched along with instructions from the instruction memory.

An inline immediate can be given as a literal or an expression. The maximum bit width of an inline immediate is defined by the width of an inline immediate field of the bus that is used for the data transport. Extending the value to the size of an inline immediate field of a bus is done by the literal extending rules described in Section 3.1.3. Table 3 shows examples of moves moving inline immediates to a register.

Each move can be conditional and *guard* defines if the move is really executed. If the value of *guard* is non-zero it means that the guarded move will be executed. On the contrary if the value of *guard* is zero, then the guarded move is just omitted and the data transport is not done.

Table 3. *Examples of Moves Containing Inline Immediates.*

Move		Description
1	-> RF.1	Decimal literal.
0b1	-> RF.1	Binary literal.
-1	-> RF.1	Signed decimal literal.
label+4	-> RF.1	Label expression.

Guard is always prefixed with either a question mark (?) or an exclamation mark (!). When *guard* is prefixed with question mark it means that the value of *guard* is interpreted as was described in the previous paragraph. On the other hand, when *guard* is prefixed by exclamation mark it means that the value of *guard* is inverted before the execution decision, which means that the guarded move is executed only if the value of *guard* is zero.

Guard can either be a general-purpose register or a function unit port reference. When a register is read from the guard field of a move, only the *reg-file-name.index* syntax is allowed. If *guard* is a function unit port reference, the only allowed syntaxes for referring to the port *guard* are *fu-name.port* and *fu-name.operation.operand-index*.

Restrictions to syntax, how *guard* can be referred to, come from the special way, how the GCU receives the value of a guard port or a guard register. The value of guard register is not read through any port of the register file and opcode information of a guard port is never passed to the GCU.

For obtaining the value of a guarded register or a guarded port, TCE TTA processors have a hard wired connection between the register or the port and the GCU, which actually decides if the data transport is executed. Each bus in ADF has a static set of *guard*s defined for them. Therefore, only those registers and function unit ports can be used as a *guard*s, which are explicitly defined for the bus that is used for the data transport.

For instance, in case where simple C-code is translated to TCE assembly:

```
# condVar ?  (addOper1 = 0) :  (addOper1 = 10);
_condVar -> lsu.ldq.1, ...  ; # load value from memory
?lsu.ldq.2 0 -> alu.add.1, !lsu.ldq.2 10 -> alu.add.1 ;
```

In the example, the value stored at address *_condVar* is loaded, from memory and it is used as a *guard* for selecting the constant for the addition. Because all the *guard*s which can be used for a conditional execution have to be defined in ADF, the *lsu.ldq.2* port has to be defined for the bus one and an inverted *lsu.ldq.2* port for bus two.

Instruction

An instruction defines the moves through each bus and all the long immediate assignments to immediate units for one clock cycle. An instruction can also contain labels, for instance, to indicate jump destinations in code. Instruction templates are defined in ADF and they describe all the combinations of the long immediate assignments and the moves that may occur in a single instruction. Essentially, instruction templates define all the possible formats of an instruction. Instruction templates are explained in more detail in Section 2.2.

In assembly syntax, buses are represented by move slots. Each slot can define one move. All the move slots of an instruction must be in the same order that buses are introduced in ADF. In other words, the first move slot of an instruction defines the move for the first bus of ADF and the second move slot defines the move for the second bus. All the move slots are separated with commas.

Long immediate transports are defined after all the move slots of the instruction. Each long immediate assignment is defined inside square brackets.

A semicolon ends the instruction and the syntax for an instruction is defined as follows:

```
move, move, move [long immediate] [long immediate] ;
```

However the precise format of an instruction depends on the instruction templates available in ADF. For example, it is possible that some of the move slots may not contain anything else but empty moves if a certain long immediate assignment is done, or there might be no long immediate assignments at all.

If some of the move slots are not occupied in an instruction, those slots can be marked to be empty with a special empty move that is represented by the string "...". For example, a completely empty instruction for a machine containing three buses would look like this:

```
..., ..., ...  ; # completely empty instruction
```

However there is also an alternative way for writing an empty instruction. Special syntax ". . ." means that all the buses of an instruction are set empty.

```
.  .  .; # completely empty instruction
```

All the move slots that are left unused at the end of instruction are set to be empty. Say, if we again have ADF with three buses then

```
..., move ;
```

is equivalent to

```
..., move, ...  ;
```

Labels for an instruction are declared before the move slots. The address for the label refers to the next instruction that is found in the code. The syntax for labels is described in more detail in Section 3.1.3.

3.1.5 Data Sections

Data sections define all the data memory used by the program. Similar to the code sections, which consist of instructions, data sections consist of data area definitions. Each data section is specific to one address space, therefore all the data area definitions in one data section belong to the same address space. However, there can be multiple data sections concerning the same address space.

For instance, in the case when there are two data address spaces *dataSpace1* and *dataSpace2*, the following example shows how data area definitions will be organized to different address spaces.

```
DATA dataSpace1 ;
# data areas defined here go to memory of dataSpace1

DATA dataSpace2 ;
# data areas defined here go to memory of dataSpace2

DATA dataSpace 1 ;
# data areas defined here are placed to dataSpace1
# memory to addresses that are after the first
# dataSpace1 definitions.
```

There are basically two rules how data is placed in address spaces:

1. Each data area definition of the same address space is assigned to a higher address after the previous data area definition of the same section.
2. Each data section must start from a higher address than the address where the last section of the same address space ended.

Data Area Definitions

Data area definitions reserve contiguous blocks of memory from a specific address space. Each data area definition consists of the size of area which is reserved and optionally the initialization values for the data. There can also be labels before each data area definition. The use of labels is described in Section 3.1.3 and data areas are defined with the following syntax:

```
DA number-of-MAUs [init-data init-data ...]  ;
```

The *number-of-MAUs* field describes the number of MAUs that are allocated from the address space. The only way to ensure that reserved data is contiguous is to reserve it with one data area definition. If two separate data area definitions are written one after another, the assembler does not guarantee that the data areas are reserved from contiguous addresses, e.g., the compiler might align all the data areas to start at word boundaries.

The *init-data* fields are optional parameters, which describe if the area is initialized with some values. If there is no *init-data* parameters, then the data area is uninitialized, which does not increase the size of the created binary. If there are *init-data* fields, it means that there is an initialization value for all the MAUs of the defined data area definition.

The *init-data* field consists of two parts, *size* and *value*. The *size* field describes how many MAUs the user wants to reserve for the value. *Value* is the value that is written to those MAUs. The *init-data* field is defined with the following syntax:

```
[size:]value
```

The *size* field is an optional parameter and if *size* is not defined, the compiler calculates the smallest MAU count for storing the value. The *value* field can be either a literal or an expression referring to some label, which are described in section 3.1.3. Because of the restricted size of the literals, the maximum value for one *init-data* field is 32 bits. However, the *size* field can be greater than 32 bits, for example, 8:-1 will create 8 MAUs wide -1 to data area. Figure 5 shows three different data area definitions and how they are interpreted in a data address space whose MAU width is 8 bits.

In the first data definition of Figure 5, all the seven MAUs of the data area definition are initialized. The definition contains two initialization fields, first field initializes 4

```
DATA data_memory ;
DA 7 4:0x10 3:12 ; # initialized data area with MAUs:
                   # 0x00 0x00 0x00 0x10 0x00 0x00 0x0c

DA 8 ;             # 8 MAUs of uninitialized data

DA 5 0xfff ;       # initialized data area:
                   # 0x0f 0xff 0x00 0x00 0x00
```

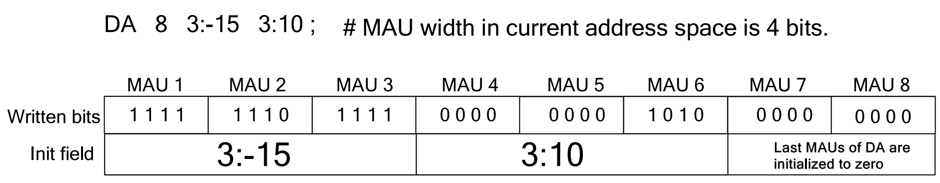
Figure 5. *Examples of Different Data Area Initializations.*

MAUs to the value 0x10. Because the processor architecture is big endian the most significant byte is the leftmost byte of data area and the least significant byte is the rightmost byte of the value. The second initialization data field initializes the last 3 MAUs of a data area with the decimal value 12.

In the second data definition, simply 8 MAUs of uninitialized data are reserved. Because the MAUs are in the same address space that the first definition, these 8 MAUs will be placed to higher addresses than the first 7 MAUs. However, the compiler does not guarantee that the second definition starts straight after last MAU of the previous definition.

The third data definition defines 5 MAUs of initialized data. There is only one initialization data field, which has no field size. In this case, the compiler calculates the minimum MAU count for storing the value, which is 2 MAUs for 0xfff. The remaining 3 MAUs of the data area are initialized with zeroes.

As shown in Figure 6, each initialization value is sliced to bit sequences that are as wide as the MAU width of the address space. Parts are placed in MAUs of the address space, so that the least significant bit of the value is in the rightmost bit of the last MAU of the initialization field of the data area. If the leftmost MAU of the value is not entirely filled with bits of the value, then the value width is extended to be a multiple of MAU. Figure 6 further shows that each separate *init-data* field is placed one after another to the data area. If the *init-data* value is wider than the *size* that is reserved for



```
DA 8 3:-15 3:10 ; # MAU width in current address space is 4 bits.
```

	MAU 1	MAU 2	MAU 3	MAU 4	MAU 5	MAU 6	MAU 7	MAU 8
Written bits	1 1 1 1	1 1 1 0	1 1 1 1	0 0 0 0	0 0 0 0	1 0 1 0	0 0 0 0	0 0 0 0
Init field	3:-15			3:10			Last MAUs of DA are initialized to zero	

Figure 6. *Data Area Definition Written to Memory.*

it or if all *init-data* fields together are wider than the data area width, then an error is emitted.

3.2 Disambiguity Rules

When a grammar has many alternatives to parse one string, the grammar is ambiguous [6]. Disamiguity rules are used for resolving such cases. ADF defines the names for all the processor resources and it has very loose naming restrictions. Therefore, it is perfectly valid, e.g., to have a register file, a function unit, and an immediate unit with the same name.

This causes problems in cases where the assembly syntax does not specify whether the referred machine resource is a function unit or a register file. Especially if an immediate unit and a register file have the same name, then it is impossible for the compiler to know which one the user wants to use. In this case, the compiler always selects the general-purpose register to use. The best way to avoid these problems is to not to use the same name for different units of the processor.

However, in most cases the type of resource can be resolved from syntax. Therefore, conflicting resource names do not always create problems for the compiler. In addition to immediate units and register files, there is another case when the compiler cannot know which processor resource was referred to in the code. Say there is a function unit and a register file named *"unit_name"*, the function unit has an operation named *"add"* and a register file contains a port by name *"add"*. Now the compiler cannot know what user meant with *"unit_name.add.1"*. In this case, the string is always interpret to be a register of a register file. If the function unit is referred to, *unit_name.P1.add* syntax variation must be used, where *P1* is the name of the function unit port that is bound for *add.1* operand.

3.3 External Requirements

The assembly syntax uses the colon as a syntactic character. Therefore, resource names or operation names in ADF cannot use this character in their names. There is also the recommended naming policy for ADF resources to prevent the need for the disambiguity rules described in Section 3.2. The naming policy is simple:

- Different units of the ADF should never have the same names.

The compiler further requires that the instruction address space is selected for the global control unit. Even if there is no program section in assembly code, the space must be defined in ADF. Also instruction templates for all the move and long immediate assignment combinations that are used in the program must be defined.

3.4 Command Line Interface

The command line interface of the compiler requires two mandatory parameters and optional switches. The required parameters are the assembly code file and the architecture definition file of the target processor. Usage of the command line interface is described by the following format:

```
tceasm [options] adf-file assembler-file
```

By default the command line interface writes the result of compilation to the same directory, where the assembly source is. The output file is named similarly to the source file, except the extension of the file name is replaced with ".tpef". If the source file does not have any extension, then extension is just added to the end of the name. For instance, "file.tceasm" and "file" would result "file.tpef" output file.

There are two optional switches that can be passed to the compiler:

```
-o file-name Override the output file name.
-q           Do not output anything unless an error
             is encountered during the compilation.
```

When compilation is successful without any warnings or errors the command line interface does not output any information about the compilation. If an error occurred during the compilation, the following type of an error message is printed out:

```
Error while compiling file:  memoryworm.tceasm
Error in line 31:  RF.0 -> SU.stq.1, 0 -> LSU.stq.2 ;
Message:  Can’t find connection for write-fu term:  SU.stq.1
```

The first and the second lines indicate the file name, the line number and the line contents, where the error occurred. The last line is the error message. In this case, there is no function unit by name "SU".

3.5 Error and Warning Messages

When the compiler detects problems during compilation, it outputs either an error or a warning message. Warnings are emitted, when a suspicious part is recognized in the source code, but the compiler can continue with the compilation. When a warning occurs, the compilation continues as if nothing had happened, and just logs the type of the warning and the line number where the warning was encountered. Errors stop the compilation at once. Usually, an error is emitted when the user tries to use a processor resource that does not exist or when the user tries to create a data transport between resources that are not connected in the processor.

3.5.1 Warnings

The following list introduces all the warning messages and explains possible reasons what is wrong with the processor or with the program.

Move destination: *destination-string* is already written in current instruction.

This warning is emitted when there are definitely two values written to the same destination resource in the same instruction. This is set to be only a warning to make sure that possible false detection does not halt compilation. The solution to resolve the warning is to check if there is the same destination used in the erroneous instruction twice.

Bus width is smaller that source.

The bit width of the source port is wider than the bus that is used to transport the data. If it does not make a difference, that the most significant bits are lost from the value, the warning can be safely ignored. Otherwise the width of the bus should be increased or the data transport must be done through another bus.

Source is wider than destination.

The bit width of the source port, where data is read from is wider than the bit width of the destination port. If it does not make a difference, that the most significant bits are lost from the value, the warning can be safely ignored. Otherwise the width of the bus should be increased or the data transport must be done through another bus.

Selected different port for move that was given by user.

The user has defined, the port of the register file, that should be used for reading or writing the register, but the compiler chose different port. This is not serious warning, since the port that is used for accessing a register file is not currently stored in TPEF file and the information about the preferred port is lost after the compilation. The warning can be solved by removing the explicit port definition from the erroneous move or by selecting the same port that is selected by the compiler.

3.5.2 Errors

The following list describes all the error messages and possible reasons what is wrong with the code or with the processor.

Syntax error.

Code cannot be parsed. Check the line where the syntax error was found and verify that all semicolons and commas are written properly. Check that there is no illegal characters in the resource or label names. Check also that all the data definitions are in the data sections and all the instructions in the code section.

Long immediate destination must be immediate unit.

The destination of the erroneous long immediate assignment is not an IU. Check that you have specified the correct destination unit and verify, that it is specified to be an immediate unit in ADF.

Too many bus slots used.

The erroneous instruction contains more moves than the processor contains buses. Check that there are as many moves in instruction as buses are defined in ADF.

Can't find value for label: *label-name*

The label that is referred to in the code is not declared. Either declaration has not been done or there is misspelling in the label reference.

Can't find address space for label: *label-name*

This is an internal compiler error. If, the compiler is working correctly this should never occur.

Multiply defined label: *label-name*

The label name is declared multiple times in the program.

Can't find label: *label-name*

This is an internal compiler error. The label is not found during the recovery from an error. This should never happen if the compiler works correctly.

Can't set label to be global. ***label-name*** **is undefined.**

The label that should be set to be global, with *:global* directive is not declared in the program. There can be misspelling either in the *:global* directive or in the label declaration.

Invalid procedure declaration: ***procedure-name***

The *:procedure* directive is used at the end of the code section after the last instruction. The error is solved by removing the erroneous procedure declaration.

Can't find destination section for address: ***address*** **of address space:** ***aspace-name***

This is an internal compiler error. This should never happen if the compiler works correctly.

Can't find code address space.

The address space of GCU is not set in ADF. Error can be solved by defining a code address space to ADF and by setting the GCU of the processor to use it.

ADF doesn't contain address space: ***aspace-name***

The address space that was requested was not found in ADF. Possibly the address space name in a data section header declaration is misspelled.

Method can't be used for getting code address space.

The user tried to declare a data section in the code address space. If ADF does not contain a separate data address space, it needs to be defined and all the data should be stored there.

Can't find operand: ***operation*****.index**

The *operation* does not contain operand for the requested *index*. Check how many operands are bound for the operation from ADF.

Can't find operation: ***operation-name***

The function unit does not contain operation for the requested *operation-name*. Add operation to the function unit or check if another function unit contains the requested operation.

Can't find connection for ***request-type*** **term:** ***unit***

Request-type is "guard", "read" or "write". Depending on the request type this means that there is no connection between buses and the used resource or there is no guard specified for the used bus. Check that the needed guards are defined for the bus and that the referred resources have suitable sockets for executing the data transport.

Not enough registers in register file.

The user tried to read or write a register index that is bigger than the number of registers in register file. The error can be solved, for instance, by adding more registers to the register file or by using some other register.

Can't find function unit from machine: *fu-name*

The *fu.port.opcode* reference was used to reference some function unit name that was not found in the ADF. Probably there is misspelling in the name of the function unit. Unit names are case-sensitive and they have to be be written exactly as they are defined in ADF.

Can't find port: *port-name*

The *fu.port.opcode* reference was used to refer to a port name that is not found in the function unit. Check the port names of the function unit or use the other less restrictive syntax variant for a function unit reference.

Operation *operation-name* is not bound to port: *port-name*

The operation does not have any operands which are bound to the accessed port. Check that there is an operation binding defined between *operation-name* and the *port-name* port in the function unit.

Operation *operation-name* is not found in fu: *fu-name*

The requested operation is not found in the function unit. Add the operation to the function unit or try to use another function unit containing the requested operation.

Defined expression value (*defined-value*) does not match with resolved (*resolved-value*) value.

The *resolved-value* of an expression does not match with the explicitely defined *defined-value*. If there is no need to check that the label references to a specific address, the explicit address check should be removed. Otherwise the code must be reshaped in a way that the label value will be correct.

Can't start data section from address: *invalid-address* first unused address in address space: *valid-address*

There is explicit start address defined for a data section, but the previous data section of the address space has already used the requested start address. If it is possible to change the place of the fixed data area then the fixed data section should be set to start from *valid-address*. Otherwise the previous data section should be moved or reshaped to not overlap with the other data section.

Init field contains too long value. Reserved MAUs: *init-field-width* Needed MAUs: *init-value-width*

Initialization value of a init data field is bigger than the field size for the value. Check that the MAU width of the address space is correct and that you actually have reserved enough MAUs for the initialization value.

Data line contains too much init data. Reserved MAUs: *data-line-width* Needed MAUs: *init-data-width*

The erroneous data area definition contains more initialization data that is reserved. Check that MAU width for the address space is correct in ADF and/or fix the data area definition.

4. IMPLEMENTATION

At the highest level, the assembler is divided in two parts: the command line client and the compiler. This chapter focuses on the compiler part of the design and leaves out the trivial implementation of the command line client.

The module design of the compiler is introduced in Section 4.1. Next, the compilation process is described in Section 4.2. Then the external libraries are introduced in Section 4.3 and finally the maintenance of the compiler is discussed in Section 4.4.

4.1 Module Design

As illustrated in Figure 7, the compiler is divided in six modules: Assembler, AssemblerParser, CodeSectionCreator, DataSectionCreator, MachineResourceManager and LabelManager.

The compiler is designed to be modular so that access to different resources is as restricted as possible. Thereby the use of external libraries is limited to one module per library with the exception of TPEF library. However, TPEF library access is limited as well, each module has only certain area in TPEF that they are allowed to use directly. Table 4 describes the fundamental responsibilities of each module.

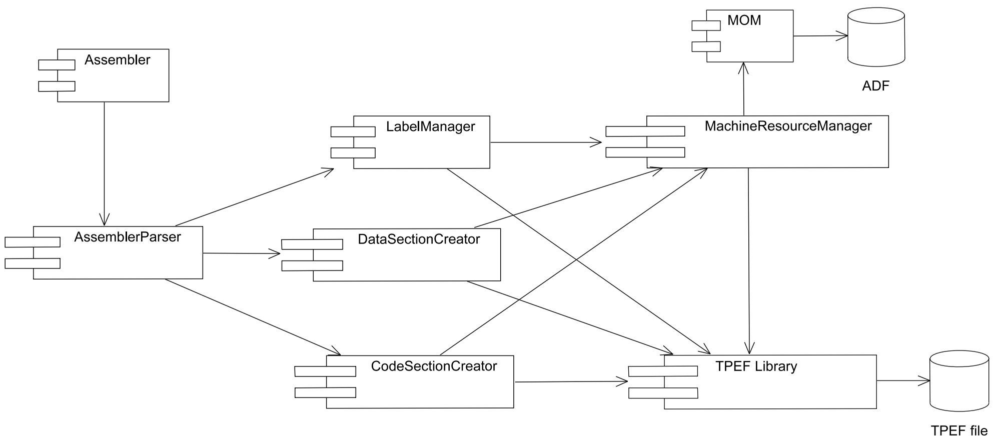


Figure 7. *Modules of Compiler.*

The modules are introduced in order of dependence. The most independent classes, used by all the other modules are introduced first. In the end, the client interface for the compiler library is described.

4.1.1 Parser Structures

The parser structures are used to store the parsed assembly program before the entire assembly code is interpreted by the compiler. The structures are hierarchical as shown in Figures 8-11. Each parser structure might contain internally other parser structures, for instance, RegisterTerm consists of FUTerm, BusTerm and IndexTerm and additionally RegisterTerm contains information about the field of a structure, that is actually used.

Each structure class contains only public attributes and a debug method for outputting the data of a structure in a textual format. There are no specific setter or getter methods in order to make the structures more compatible with different parser libraries. For the same reason, dynamic binding is not used. Parsed tokens are just copied to a suitable field of the destination structure and the type of the destination structure is set to correspond with the copied data.

For example, when RegisterTerm is used to contain FUTerm, then wanted FUTerm is copied to the *fuTerm* attribute of RegisterTerm and the *type* attribute is set to be *FUNCTION_UNIT*.

Table 4. *Designed Functionality of Modules.*

Module	Designed functionality
Assembler	The client interface of the compiler library.
AssemblerParser	Parses the assembly code and wraps all the code specific for the Spirit library.
DataSectionCreator	Stores an internal representation of the data sections and creates the data sections to TPEF.
CodeSectionCreator	Stores an internal representation of the code section and creates the code section to TPEF.
LabelManager	Stores the relocations and the labels, and creates the symbol and the relocation sections to TPEF.
MachineResourceManager	Access ADF through the MOM library and creates corresponding resources to TPEF. Writes all the other needed sections to TPEF.

All parser structures are described in order of dependence, thus the most independent structures, which represent the simplest tokens of the assembly syntax are introduced first. In the end, ParserMove and DataLine structures are described, which represent a move of an instruction and a data area definition of a data section.

BusTerm

ADF allows two buses to contain bridges between them. The BusTerm structure contains a reference to the previous or the next bus, so the value from one bus can be read to another. However, this structure is now only for the future implementation to support bridges between buses.

FUTerm

The structure stores the parsed information of the syntax variant 1 of the function unit port reference, which is described in Section 3.1.4. FUTerm consists of three string attributes: *part1*, *part2* and *part3*, and of one boolean attribute *part3Used*, that indicates if the *part3* attribute of the structure is used.

The *part1* attribute stores the name of the referred function, the *part2* attribute stores the name of the referred port and the optional *part3* attribute stores the operation name of a reference.

IndexTerm

IndexTerm stores the parsed information of the syntax variant 2 of the function unit port reference, a general purpose register reference or a long immediate register reference, which all are described in Section 3.1.4. The structure consists of two string attributes: *part1* and *part2*, of one unsigned *index* attribute, and of one boolean attribute *part2Used* which indicates, if the *part2* attribute is used.

In case of a function unit reference, the *part1* attribute stores the name of a function unit, the *part2* attribute stores the name of the operation and the *index* field stores the operand number. Because the syntax variant 2 of a function unit port reference requires the operation name to exists, the *part2Used* attribute is always set.

If a register file or an immediate unit is accessed, *part1* stores the name of the referred unit. The *part2* attribute stores the name of the referred port if it is defined in the

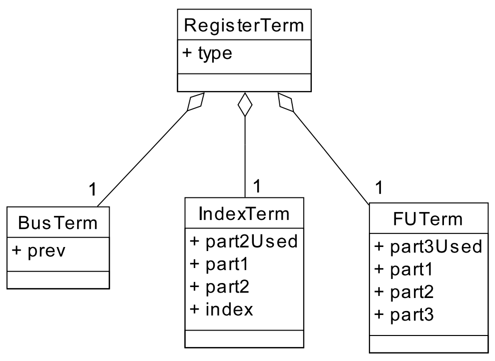


Figure 8. *RegisterTerm Structure.*

assembly code. However, it is very exceptional to define a port name for a register references. The *part2Used* is used to indicate if the *part2* attribute contains valid data. The *index* attribute stores the index of the register that is accessed.

RegisterTerm

The RegisterTerm structure, illustrated in Figure 8 represents any type of a port or a register reference. The structure consists of the *type*, *busTerm*, *fuTerm* and *indexTerm* attributes. In addition to the *type* attribute, only one of the *busTerm*, *fuTerm* or *indexTerm* attributes is valid at the time. The *type* defines, which type of value is currently stored in the structure.

Expression

Expression represents an expression of the assembly syntax described in Section 3.1.3. The structure shown in Figure 9 consists of the *label*, *hasOffset*, *isMinus*, *offset*, *hasValue* and *value* attributes.

The *label* attribute stores the name of the label, whose value is used as the base value for calculating the resolved value of an expression. The *hasOffset*, *isMinus* and *offset* attributes define if there is an offset defined to the referred *label*.

The *hasValue* and *value* attributes define, if there is a resolved value set for the expression. This value can be given in assembly code or otherwise it is resolved and set by the compiler.

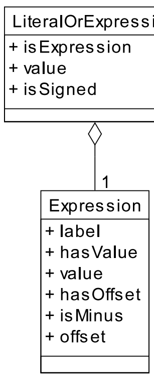


Figure 9. *LiteralOrExpression Structure.*

LiteralOrExpression

LiteralOrExpression shown in Figure 9 is used to store either Expression or a constant value. The structure consists of the *isExpression*, *expression*, *value*, and *isSigned* attributes. The *isExpression* attribute defines if the structure is used to store an Expression or a constant value. The *value* and *isSigned* fields define the value and information about how it should be interpreted when it is written to a data memory or to an immediate.

ParserSource

The ParserSource structure is used to store the source field of a move of the assembly syntax described in Section 3.1.4. It can contain either a RegisterTerm or LiteralOrExpression, which are the only possible sources of a move.

The structure consists of the *isRegister*, *regTerm*, and *immTerm* attributes. The *isRegister* is boolean attribute which defines if ParserSource stores LiteralOrExpression or RegisterTerm. The *regTerm* and *immTerm* attributes are used to store the data of the corresponding RegisterTerm and LiteralOrExpression structures.

ParserGuard

ParserGuard is used to store the guard field of a move of the assembly syntax. The structure shown if Figure 10 consists of the *isGuarded*, *isInverted*, and *regTerm* attributes.

The *isGuarded* and *isInverted* attributes are boolean attributes, which define if the

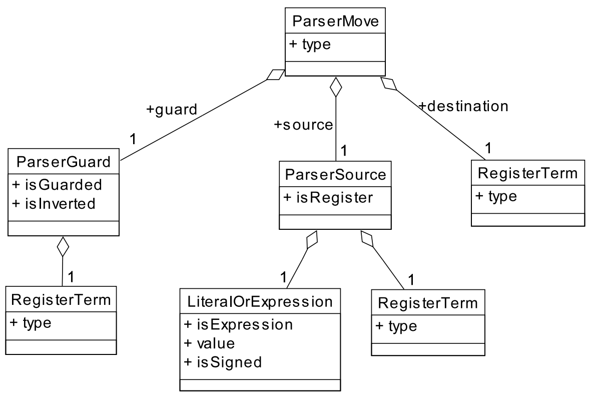


Figure 10. *ParserMove Structure.*

structure is used at all and how the value of a guard is interpreted. The *regTerm* attribute is a RegisterTerm that defines the port or a register that is used as a guard.

ParserMove

The ParserMove structure presented in Figure 10 represents a move or a long immediate assignment of the assembly syntax, which are described in Section 3.1.4. The structure consists of the *type*, *isBegin*, *guard*, *source*, *destination*, and *asmLineNumber* attributes.

The *type* attribute defines if the structure is used to store an empty move, a normal move or a long immediate assignment. The *isBegin* attribute defines if the structure starts a new instruction in assembly code. The *guard*, *source*, and *destination* attributes are used to store the corresponding fields of a move.

The *guard* and *source* attributes are stored as a ParserGuard and ParserSource structures, but the *destination* attribute is a plain RegisterTerm, because the destination field of a move can be only a port or a register. The *asmLineNumber* field is an unsigned value which stores the line number of the assembly code where the current move is placed. This field is used to resolve an error line number, if the compilation is not successful.

InitDataField

InitDataField represents a single init data field of a data area definition described in Section 3.1.5. The structure consists of the *width* and *litOrExpr* attributes. The *width*

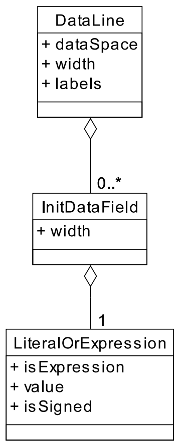


Figure 11. *DataLine Structure.*

attribute defines the number of MAUs that are used for storing the value stored in the *litOrExpr* attribute.

DataLine

The DataLine structure shown in Figure 11 stores the information of a data area definition of the assembly syntax. The structure consists of the *width*, *dataSpace*, *initData*, *labels*, and *asmLineNumber* attributes.

The *width* attribute defines the number of MAUs that are defined by the structure and the *dataSpace* attribute defines the name of the address space to which the data belongs. The *initData* attribute stores a table of InitDataField structures. This attribute is also used to determine whether the DataLine structure contains an uninitialized or an initialized data area definition. If the *initData* table is empty then all the MAUs defined by the *width* attribute are uninitialized data, otherwise the MAUs are initialized. The *labels* attribute is a table of all the names of the labels, that are declared for the current DataLine structure.

4.1.2 MachineResourceManager

The MachineResourceManager module consists of a single class that is used to get various resources needed by the other modules. It is responsible for retrieving data from ADF and for creating corresponding resources to TPEF. Module also provides TPEF objects for other modules to use. The sections of TPEF which MachineResourceManager creates and manages are NullSection, StringSection, ResourceSection and ASpaceSection.

When a resource is requested from MachineResourceManager, it first checks if the same resource is already retrieved earlier. If it is the first time when the resource is requested, the module finds the resource from ADF and creates a corresponding resource to TPEF. The module also adds the created resource to a cache, so the next time when the resource is requested MachineResourceManager already knows that the resource is added earlier to TPEF and request was valid. Therefore the resource does not even have to be verified again.

An ADF resource request defines a resource which is wanted to be retrieved, a bus that is used for the access and the direction if the resource is read or written. For instance, if a function unit port write is requested, the module checks that the bus and the port are connected and it also verifies, that the connection may be used for writing. If an invalid request is made, the module throws an exception with an error message and the error line number where in the code the error occurred.

4.1.3 LabelManager

LabelManager is responsible for storing all the information of the labels and the relocations in the program. CodeSectionCreator and DataSectionCreator add labels and relocations to this manager. After the labels and their resolved values have been added, the other modules can request the information of the labels.

The LabelManager module is used in two phases in the compilation, in the first phase all the information of labels, relocations and procedures are added. In the second phase the *finalize()* method is called and the data is written to TPEF sections and all the internal data of the LabelManager module is cleared.

Before the execution of the *finalize()* method, all the referred instructions and the data must exists in TPEF. In other words, the LabelManager module has to be the last module that is finalized. If the finalizing is tried to do before all the other required resources are written to TPEF, the method throws an exception and restores the state of the module to the same as it was before the failed function call.

4.1.4 DataSectionCreator

DataSectionCreator stores the DataLine objects and organizes them to TPEF data sections. The module works in two phases: in the first phase, DataLine objects are added to the module and, in the second phase, the module resolves the start addresses for the data definitions and adds resolved addresses of the data labels to the LabelManager module. In the second phase, the data sections are created and added to TPEF.

The second phase is started by executing the *finalize()* method, when the module verifies, that all the DataLine objects are valid and they can be placed in TPEF data sections. In this phase, all the code labels have to exists in the LabelManager module. After a successful execution of the second phase, all the data stored in the module is cleared. If there is an error during the finalization, an exception is thrown, and the state of the module is restored to be the same as it was before the failed function call.

4.1.5 CodeSectionCreator

CodeSectionCreator stores the ParserMove objects and organizes them to instructions and finally generates the TPEF data section out of them. As most of the modules, the functionality of CodeSectionCreator is divided in two phases. In the first phase, ParserMoves are stored to the module and immediately checked that they are valid as far as possible.

The second phase is started by executing the *finalize()* method, like with the other modules with two stages. The finalizing creates the code section to TPEF and resolves all expressions that are stored in the immediates in code. The expression resolving requires that the addresses for all the labels are already resolved, therefore it is required that the DataSectionCreator module is finalized before CodeSectionCreator. If the execution of the second phase fails, an exception is thrown and the module is restored to the same state that it was before the failed function call.

The addresses of the code label declarations are known already when the ParserMove objects are stored in the modules, because of this all the code labels are already added and resolved straight after the parsing. Therefore there is no need to resolve code symbols during the finalization phase.

4.1.6 AssemblerParser

The AssemblerParser module is the core of the compilation and it contains all the syntax definitions of the compiler. It reads assembly code and creates TPEF with the help of the other modules. The module does the compilation in two phases: In the first phase, the assembly code is parsed and all the parsed data is stored to the other modules. In the second phase, the module writes parsed data to TPEF, which is done by commanding all the other modules to finalize themselves in the required order.

The first phase of the compilation is started by executing the *compile()* method, which requires the assembly code in a string as a parameter. If there are any problems in the syntax of the assembly code, the compilation is stopped and the line number where the syntax error was is stored.

4.1.7 Assembler

The Assembler module is the only interface for the external clients of the compiler. It hides all the internal parts of the compiler and just takes the file containing the assembly code and the machine which for the code is compiled. As a result, the module returns the TPEF object and list of warnings that occurred during the compilation. If there is an error during the compilation, an exception is thrown and the error line and message can be requested from the module.

The module handles storing and managing all the warnings. When the other modules detect a warning in the code, they add it straight to the Assembler module.

4.2 Compilation Flow

Compilation is carried out in two phases. In the first phase, the compiler parses the assembly code to an internal representation. Most of the error checking is done in this phase. The syntax of code is checked and the availability of the used machine resources are verified. In the second phase, the internal representation of the assembly code is converted to a TPEF object hierarchy and all label addresses are resolved.

As shown in Figure 12 the compilation is started by the Assembler module, which first cleans up all the information stored from the possible previous compilation process. Then it reads the assembly code from a file to a string and creates TPEF and AssemblerParser instances for the compilation. Next the Assembler module tells the created AssemblerParser to parse the code to an internal format.

The parsing of the code is done by starting from the first line and when a whole move or a data line is parsed, AssemblerParser adds it to DataSection creator or to CodeSectionCreator. Further the parser adds all the code labels and addresses for them to LabelManager. When the parser encounters a procedure or a global directive, it tells

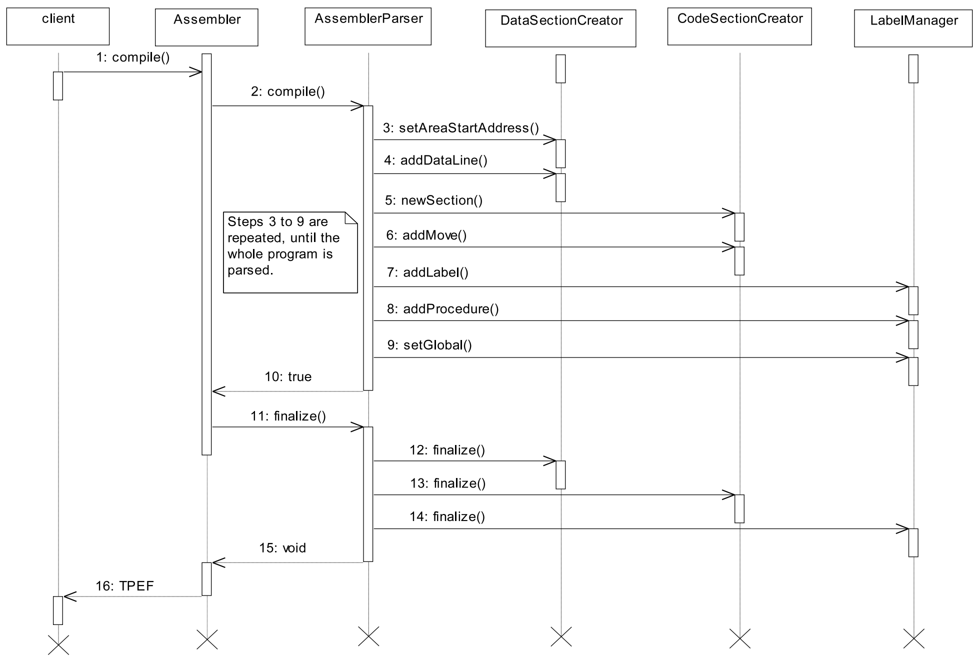


Figure 12. *Compilation Process*

LabelManager to set the label to be global, or to add a procedure start symbol for the currently compiled instruction. When a section declaration is encountered, the parser passes information of the declaration to CodeSectionCreator or DataSectionCreator. Details of the parsing process are described better in Section 4.3.

When the parsing is completed, the control is returned to the Assembler module, which tells AssemblerParser to finalize the compilation. In the finalization, the Assembler-Parser first sets the file architecture and type parameters for TPEF and continues by finalizing the other modules; the DataSectionCreator is the first, followed by the Code-SectionCreator, and finally LabelManager is finalized.

The finalization order of the modules is vital for the successful compilation. The Data-SectionCreator requires that all the addresses for the code labels have already been resolved. That is the main reason why all the code labels are already added to La-belManager during the parsing. The LabelManager requires that the code and data sections of TPEF are already added, thereby the CodeSectionCreator is finalized next and LabelManager after all the other modules. Since MachineResourceManager cre-ates all the resources to TPEF on the fly when they are requested for the first time, it does not need any finalization.

After the finalization is completed, control is returned again back to the Assembler module, that just returns the compiled TPEF for the client software.

4.3 Used Libraries

The implementation of the compiler is quite simple because all the base functionality for the file writing and the parser functionality was already implemented by different libraries. The compiler mostly checks if the resources used by the assembly code are found from ADF.

4.3.1 Spirit

Spirit [7] is a parser library provided by Boost library collection [8]. The library is wrapped in the AssemblerParser module, which is the only module that contains Spirit specific code.

Grammar and Actors

Grammar [6] of the parser is implemented as a grammar class of Spirit. Spirit has various predefined parser classes to parse, for instance, integers, characters and strings, from the input. By combining these primitive elements, the entire assembly syntax is described. Functionality to be performed when each token is recognized from the assembly code, is implemented with the actor classes of Spirit.

Actor classes are normal classes, which have ()-operator implemented. For example, an actor that copies a parsed string to a vector could be implemented as in Figure 13.

Spirit also has many useful predefined actors, such as *increment_a*, *assign_a*, *push_back_a*. These actors can be used to increase the value of an integer, for assigning a string or a value to a variable and for adding an element to a STL [9] container. AssemblerParser mostly uses these predefined actors but has a few specific actors for adding data to the other modules as well.

The use of the actor and the parser classes of Spirit is very straightforward. With Spirit, a parser which increments the *wordCount* variable each time when a word "magic" is encountered in the input, could be defined as follows:

```
str_p("magic")[increment_a(wordCount)]
```

```
class AddToVectorActor {
public:
   AddToVectorActor(vector<string>& aVec) :
       vector_(aVec) {}

   void operator()(const char* start, const char* end) {
        string parsedString(start, end);
        vector_.push_back(parsedString);
   }
private:
   vector<string>& vector_;
};
```

Figure 13. *Example Implementation of Actor Class.*

Incrementing of *wordCount* is done with the *increment_a* actor and finding the word "magic" with the *str_p* parser. Action that is executed when a pattern is found is given after the parser in square brackets. If many actions are need to be done for the same pattern, many square brackets may be defined one after another.

Based on the experience of using Spirit, the parameters for the actors are very limited. It seems to be that the only valid parameters are plain statically allocated variables. It is not allowed, for example, to write a function call or pass a pointer of a variable to any actor. Since only the variables can be passed as a parameter, constant literals can not be used, for example, to set a boolean variable to be *true* or *false*, but there have to be constant variable defined, which value is used for assignment. Spirit does not recognize when invalid values are passed for the actors, so there are no warnings or errors emitted by the compiler. One valid and three invalid uses of the *assign_a* actor are illustrated in Figure 14.

In the first example, string "end" is parsed from the input. When the parser encounters the string, *assign_a* actor is used for setting the value of the *endFound* variable to be the same as the constant variable *MY_TRUE*.

In the second example, the same string is parsed, but constant literal for assigning a value for the *endFound* variable is tried to use. This applies also, for instance, if zero or an enumeration is used in assignment. Only variables can be used as parameters.

The third example makes a function call inside the constructor of the actor. The function call might be run once during the parsing but certainly will not be executed for all

```
// valid
(1) str_p("end")  [assign_a(endFound, MY_TRUE)]
// invalid
(2) str_p("end")  [assign_a(endFound, true)]

// invalid
(3) str_p("end")  [assign_a(endFound, object->retBool())]

// invalid
(4) str_p("swap") [assign_a(&temp, &obj1)]
                  [assign_a(&obj1, &obj2)]
                  [assign_a(&obj2, &temp)]
```

Figure 14. *Examples of Valid and Invalid Uses of assign_a Actor.*

encounters of the "end" strings.

In the fourth example, the problem is that, swap is done by using assignments of pointers. The same problem arises when an object is passed for the actor through a pointer with the *-operator.

Parsing

Assembly code is first parsed to *parserTemp_* structure. The parsed values are stored there until a whole move or a data line is parsed. After a move or a data line has been parsed they are stored in the other modules.

First when pattern is recognized, for example, when a constant is parsed, it is stored temporarily in *parserTemp_.lastNumber*. When compilation finds a higher pattern, like a register reference, which is parsed to IndexTerm structure, *parserTemp_.lastNumber* is copied to *parserTemp_.indexTerm.index* variable. For example, the parsing of the following piece of assembly code

```
FU.add.3
```

utilizes syntax declarations presented in Figure 15.

The parser first tries to find as long pattern as possible and then executes all the actions for the pattern in the same order, that they are defined. If there are actions at many levels as in *index*(3) pattern, then the actions of the lowest level are executed first.

```
(1) uNumber = uint_p
              [assign_a(t.lastNumber)]
              [assign_a(t.isLastSigned, MY_FALSE)];

(2) name = (alpha_p | chset\_p("_")) >>
           *(alnum_p | chset_p("_"));

(3) index = uNumber[assign_a(t.index, t.lastNumber)];

(4) indexTerm =
      eps_p
      [assign_a(t.indexTerm.part2Used, MY_FALSE)] >>
      name
      [assign_a(t.indexTerm.part1)] >> '.' >>
      !(name[assign_a(t.indexTerm.part2)] >> '.')
      [assign_a(t.indexTerm.part2Used, MY_TRUE)] >>
      index
      [assign_a(t.indexTerm.index, t.index)];
```

Figure 15. *Syntax Declarations Utilised When Parsing String "FU.add.3".*

The parser recognizes *FU.add.3* to be *indexTerm*(4), so the first thing is to reset *index-Term*(4) structure with the epsilon *eps_p* parser and *the assign_a* actor. The epsilon parser, is a parser, which always matches, so when the *indexTerm*(4) is recognized *t.indexTerm.part2Used* is initially set to *MY_FALSE* by the *assign_a* actor.

Next the "FU" and "add" strings are recognized to be the *name*(2) patterns and copied by *assign_a* actor to the *t.indexTerm.part1* and *t.indexTerm.part2* strings. Since there was *t.indexTerm.part2* defined in the recognized pattern the *t.indexTerm.part2Used* is also set to be *MY_TRUE*.

The number "3" at the end of the whole pattern is recognized by *index*(3) pattern, which is further recognized to be *uNumber*(1), which finally interprets the value to be "3". The interpret value is copied by the *assign_a* actor to *t.lastNumber* and it is set to be handled as an unsigned value. Then the *index*(3) pattern copies *t.lastNumber* to *t.index* and finally *indexTerm*(4) pattern copies *t.index* to *t.indexTerm.index*.

4.3.2 TPEF

The library provides the object representation of the TPEF binary [5] file. Each module of the compiler has a limited part, that they are responsible for. The Assembler module creates an initial empty TPEF for the compiler. The AssemblerParser module sets the file type and the architecture of the TPEF. The CodeSectionCreator module creates the code section and DataSectionCreator creates the data sections. The LabelManager creates the symbol and relocation sections and the MachineResourceManager creates all the rest of the needed sections.

4.3.3 Machine Object Model

Machine Object Model (MOM) contains the object representation of ADF. This library is used by MachineResourceManager to find and verify that the resources used in the assembly code are available in the processor.

4.4 Maintenance and Future Development

The most fundamental guide to the maintenance of the compiler is: keep implementation of features in places where they belong. Table 4 revises the designed functionality of each module. If support for writing a new section is added to the compiler, usually a new section creator class is needed for the implementation. For instance, if TPEF debug section writing is wanted, then the DebugSectionCreator module has to be implemented into the compiler.

The section first describes how new features are added to the compiler in Section 4.4.1, and continues by describing how the implementation of the compiler can be changed to use POM instead of the TPEF library in Section 4.4.2. Finally, a possible solution to initializing over 32-bit values is presented in Section 4.4.3.

4.4.1 Adding Support For a New Feature of the Processor

The TCE TTA processor is still in development and new features are added to the processor frequently. Usually, the assembler is the first tool, used for generating a program which utilizes these new features. Therefore, maybe the most important case for the maintenance is adding support for them.

When a new feature is added to the compiler, the first thing is to specify syntax for supporting the feature. The syntax should be selected in a way that it is user friendly and if possible, it should not contain any special syntactic characters which are not already used by the current syntax definitions. After the syntax is specified it can be implemented in the AssemblerParser module. Currently the compiler has additional syntax declarations for string literals and bridge register references. However, there is no implementation for actually compiling them.

If the parser structures do not provide a way to store the parsed information of the new syntax, then new structures need to be written. After this, the AssemblerParser can be set to read the parsed data to the new parser structures.

Finally the implementation of the new feature is added to a suitable module. If many modules are needed for the feature, then the implementation is divided based on the current design, for instance, if a new register reference type is added, the implementation has to be divided in the CodeSectionCreator and the MachineResourceManager.

Sometimes new syntax does not need to be parsed to the parser structures but the parsed information can be added directly to the module that implements it. For example, all the current directives are implemented that way.

4.4.2 Changing Implementation to Use POM

The original reason, for writing the compiled program directly to TPEF model is, that POM did not support all the features, which were needed by the assembler. In the early development of POM, this was an advantage since the assembler could be used to generate parallel test programs for POM. However, writing data directly to TPEF has some serious drawbacks.

TPEF is a complex structure and there are plenty of details that have to be known by the user to be able to generate a valid TPEF model. Further it has been necessary to duplicate a lot of functionality, which is already implemented in TPEF to POM and POM to TPEF converters. Each time when a new feature is added to the assembler, more code is duplicated.

The current design of the compiler is suitable for replacing TPEF with POM structure but it means that most of the interfaces of the MachineResourceManager need to be changed and almost all the implementation except parsing have to be rewritten.

In addition to this the error bookkeeping system needs to be designed for making it possible to resolve the error line of the assembly code if an exception is thrown by POM to TPEF converter. Also the POM to TPEF converter exceptions should be modified to contain information of the address, where the error occurred in conversion.

4.4.3 Support for Over 32-bit Value Initializations

The compiler supports an initialization of over 32-bit wide data areas with one value, for example, initialization 6:16 with a 8-bit address space writes 16 as a 48-bit wide value to a data address space. However, due to a limitation of the how values are parsed from assembly code, it is possible to write at most 32-bit values, even if the field for writing is wider. This limitation also makes it impossible, for example, to read immediate assignments to double precision float registers.

Since the assembler understands all the data as a raw sequence of bits, it is possible to write an implementation for general value storing, that is consistent for reading integers, floating point values and even string literals. The good thing with this is that internally the compiler always interprets the data the same way, so there will not be the need for writing any special cases when the data is read from a different source value. All that is seen outside of the parser is a sequence of bits and how the sequence should be extended if necessary.

One possibility for implementing the feature is to store read data to a data type that may store an arbitrary number of bits. The GNU multiple precision arithmetic library [10] is a library providing all the needed data types and a functionality for an easy implementation of the feature. However, the functions and data types of the library should be wrapped in own its class, so the dependency for the external libraries remains as restricted as possible.

5. VERIFICATION

Since the compiler has been quite ready for a few months, it has already been used, for instance, to generate parallel code for the parallel simulation benchmark of TCE simulator [1]. For the benchmark, the valid functionality of the compiled assembler code was verified. The compiler has also been used for a radix-4 FFT implementation which is used as a test program for extensive verification.

Fundamentally, there are two requirements for the working assembly compiler: the compiler produces correct output, and the compilation speed is linear with respect to the size of a program.

The functionality of the compiler was verified by compiling a program which utilizes most of the features of the compiler and by running the program with TCE simulator. The validity of the simulation was verified by comparing the simulated results with a known correct output. Further the structure and the elements of the compiled binary were compared with expected results with *dumptpef* program, which shows the internals of TPEF binary in human readable form. Linearity of the compilation speed was roughly verified by compiling three programs, which contained 10000, 20000, and 30000 moves. Further the error handling of the compiler was tested by generating 50 different erroneous pieces of assembly code.

5.1 Validity of Compiler Output

The program chosen for the test is radix-4 FFT [11] implementation. Since the implementation of the algorithm did not use many of the properties of the compiler, the program was modified to exploit more features. However, the changes were added to the end of the program in a way that they do not have an effect on the result of the algorithm.

The correctness of the program was verified by dumping the memory where the result values of the algorithm after the simulation. These values were compared with the known expected results. The correct result values were acquired from a VHDL [12] simulation of the same program and processor generated with MOVE framework [13].

Successful simulation of the compiled program does not verify that the whole compiler works correctly. Relocations and symbols are not verified and in this case we also need to verify that multiple data sections are created correctly to the binary. This was verified with *dumptpef* tool by comparing generated binary to the assembly code and by verifying that all the necessary data areas, relocations and symbols were created as supposed.

5.2 Compilation Speed

The compilation speed was tested by compiling three very similar programs. Programs were written for a machine with two buses and the only difference between the programs were that the first one had 5000 copy pasted instructions, the second one 10000 and the last one 15000.

Compilation times for the programs were:

Table 5. *Compilation Times for Different Sized Programs*

Compiled instructions	Compilation time	Seconds / instruction
5000	6.8	0.00136
10000	14.3	0.00143
15000	19.8	0.00132

As we can see from Table 5, it is clear that the compilation time of one instruction does not depend on the size of the program.

6. CONCLUSIONS

This thesis describes an assembler compiler: TCE Assembler for a parametrizable parallel processor architecture exploited by TTA-based codesign environment (TCE). The thesis describes the syntax of the assembly language and instructs how the processor is programmed as well as the fundamental structure of TCE TTA template. Furthermore the thesis guides how the compiler is maintained and expanded to meet the new features of the processor architecture.

The syntax is designed to be as user friendly as possible and especially in a way that the program for one processor configuration can be modified for a slightly different processor configuration with as few changes as possible. Further the syntax also allows user to use white spaces and comments to make code more readable, and labels for referring to data and jump addresses to remove the need for calculating real addresses. For helping the debugging process of a program, the compiler does extensive error checking to find errors and suspicious code and outputs detailed information about the errors.

The architecture of the compiler is designed to be modular and easy to adopt for a maintainer. Fundamentally, the assembler is divided into two parts, the compiler library and the command line client. Each module in the compiler library has a well defined area in the compilation and the responsibilities are divided as follows: the client interface of the library, parsing the code, handling the labels, handling the instructions, handling the data memories, and handling all the common processor and TPEF resources. The compilation process is divided in two phases: in the first phase the code is parsed and code addresses are resolved, in the second phase all the rest of the addresses are resolved and the program is written to TPEF. The use of external libraries is limited to small parts of code, to make it possible to easily change the used library if it is necessary.

New features to the compiler are added by first defining the syntax for the new feature. Next the internal parsing structures are updated for storing the information of the new syntax. Finally the implementation for the new feature is divided in the appropriate modules of the compiler.

The functionality and validity of the compiled programs were verified by compiling and simulating test programs, which utilize the most of the features of the compiler. Compilation speed was verified to be linear towards the size of the program.

The main objectives for the compiler were easy maintenance and good usability. Fundamentally, both of the goals were reached. However, the design of the compiler library could have been better in many ways. The modules of the compiler could have been divided in smaller functional blocks, e.g., relocation and label handling could have been divided in separate modules. Further, the common interfaces and functionality could have been implemented in a base class for the modules, which contain the internal representation of the program.

Bookkeeping of the errors and warnings would have been better to isolate to a separate module to reduce the interfaces that are visible for the clients of the library. Moreover if the implementation is changed to use POM as described in Section 4.4.2, there will be more bookkeeping functionality. For this reason the error handling should be moved to a separate class anyway.

Another thing that makes the design somewhat incoherent is that the code labels are added to LabelManager already during the parsing. A cleaner solution would be compilation in three phases. In the first phase, code would be parsed to the internal representation. In the second phase, the internal representation would be analyzed and all the addresses of the labels would be resolved. Finally, in the third phase, the data would be written to TPEF.

In conclusion, there are some implementation details which may confuse the maintainer but the high level design of the compiler is reasonable, and the module interfaces are consistent and easy to adopt. The compilation sequence is easy to understand and the high level design can be preserved when the compiler is updated to meet the future requirements. Further, the usability of the compiler is particulary good for an assembly compiler because of the informative error messages and the intuitive and versatile syntax.

BIBLIOGRAPHY

[1] P. Jääskeläinen, “Instruction Set Simulator For Transport Triggered Architectures,” Master’s thesis, Tampere Univ. Tech., Tampere, Finland, Sept. 2005.

[2] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA.* John Wiley & Sons, 1997.

[3] A. Cilio, H. J. M. Schot, and J. A. A. J. Janssen, “Architecture Definition File: Processor Architecture Definition File Format for a New TTA Design Framework,” Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2003-2006.

[4] A. Cilio, “TCE Architecture Template Programming Interface Functional Requirements,” Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2004-2006.

[5] ——, “TPEF: The Program Exchange Format,” Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2003-2006.

[6] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Prentice-Hall, Inc., 2003.

[7] “Spirit User’s Guide,” Website, 2003, http://www.boost.org/libs/spirit/index.html.

[8] B. Dawes, “Boost Libraries and Documentation,” Website, 2004, http://www.boost.org/libs/libraries.htm.

[9] “Standard Template Library Programmer’s Guide,” Website, 2003, http://www.sgi.com/tech/stl/index.html.

[10] “GNU multiple precision arithmetic library,” Website, 2005, http://www.swox.com/gmp/manual/.

[11] R. Mäkinen, “Fast Fourier Transform on Transport Triggered Architectures,” Master’s thesis, Tampere Univ. Tech., Tampere, Finland, Sept. 2005.

[12] *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076-1993, 1994.

[13] *The MOVE Framework User’s Manual*, Tampere Univ. Tech., 2004.

Appendix A

PARALLEL ASSEMBLER CODE

```
##########################################################
#
# TTA paraller assembler code example.
#
# Brainless worm tries to find the way out
# from the room without an exit. In the other words,
# the worm wanderers in a restricted memory area.
#
# ADD Function unit is a 5-bit adder that is used
#     as a counter to keep track the direction where
#     to the worm is heading.
#
# ALU function unit is used for address calculating
#     when necessary.
#
# RF.0 - RF.3 Stores the current position of worm.
#
##########################################################

CODE ;

# all code labels are set to be global
:global main;
:global loopforever ;
:global trynextdirection;

:procedure main;
main:
    # initial worm position in map
    map+18 -> RF.0, map+19 -> RF.1 [IMM.0=1] [ IMM1.0 = map+20 ] ;
    IMM1.0 -> RF.2, map+21 -> RF.3 ;

loopforever:
    # write 0x00 to remove the end piece of worm
    RF.0 -> LSU.stq.1, 0 -> LSU.stq.2 ;

    # test if next direction in try sequence is ok
    trysequence -> ALU.P1, ADD.P3 -> ALU.P2.add;

    # ldq.2 got offset to next position in map where we go
    # worm head position address to adder
    ALU.add.3   -> LSU.ldq.1, RF.3 -> ALU.add.1 ;

    # calculate next address to map
    LSU.ldq.2 -> ALU.add.2, ... ;

    # mask one bit out from 9 bit result address by copying value to 8 bit reg
    ALU.add.3 -> RF.4, ... ;

trynextdirection:
    # if LSU.ldq.2 is not zero, try next direction
    RF.4 -> LSU.ldq.1, ALU.add.3 -> RF.4 ;
    ? LSU.ldq.2 trynextdirection -> GCU.jump.1, ? LSU.ldq.2 1          -> ADD.add.2 ;
    ? LSU.ldq.2 trysequence      -> ALU.add.1,  ? LSU.ldq.2 ADD.add.3 -> ALU.add.2 ;
    ? LSU.ldq.2 ALU.add.3        -> LSU.ldq.1,  ? LSU.ldq.2 RF.3      -> ALU.add.1 ;
    ? LSU.ldq.2 LSU.ldq.2        -> ALU.add.2,  ? LSU.ldq.2 ADD.add.3 -> ADD.add.1 ;

    # we found good direction for next move, map position where to go is now
    # stored in ALU.add.3

    # move worm and jump back to start
    loopforever -> GCU.jump.1, ... ;       # jump to start jump delay is 3 cycles
    RF.1 -> RF.0, RF.2 -> RF.1 ;           # move body parts
    RF.3 -> RF.2, ALU.add.3 -> RF.3 ;      # move body parts
    RF.3 -> LSU.stq.1, 0x88 -> LSU.stq.2 ; # write 0x88 to the map (new head piece of worm)

    # an example of a completely empty instruction
    . . . ;

```

```
DATA datamemory ;

# set map and trysequence labels to be global
:global trysequence ;
:global map;

# map where the worm wanderers
map: DA 256
     -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
     -1 00 00 00 -1 00 00 00 00 00 00 00 00 00 00 -1
     -1 00 00 00 00 00 00 00 00 00 00 00 00 00 00 -1
     -1 00 00 -1 -1 -1 -1 -1 -1 -1 00 -1 -1 -1 00 -1
     -1 -1 00 00 -1 00 00 -1 00 00 00 00 00 00 00 -1
     -1 00 00 00 -1 00 00 -1 00 00 00 -1 -1 00 -1 -1
     -1 00 00 00 -1 00 00 -1 00 00 00 00 00 00 00 -1
     -1 00 00 00 -1 00 00 -1 -1 -1 00 -1 -1 -1 00 -1
     -1 -1 00 00 00 00 00 -1 00 00 00 00 00 00 00 -1
     -1 -1 00 00 00 00 00 00 00 00 00 00 00 00 00 -1
     -1 00 00 -1 00 00 00 -1 -1 00 -1 -1 -1 00 00 -1
     -1 00 -1 -1 -1 00 00 -1 -1 00 00 00 -1 00 -1 -1
     -1 00 00 -1 00 00 00 00 00 00 -1 00 00 00 -1 -1
     -1 00 00 00 00 00 00 00 00 -1 -1 -1 00 00 00 -1
     -1 00 00 00 00 00 -1 00 00 00 00 00 00 00 00 -1
     -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 ;

# sequence for finding next direction to move
trysequence:
    DA 32
    -1   1 -16  16   1 -16   1  16 -16   1  -1   1  16  -1  16   1
    -16  1  -1  16  -1   1 -16  -1  16   1  16 -16  -1  1  -16  16 ;

# an example of 4 MAUs wide address in data
datatocodereloc:
    DA 4 4:main ;

# an example of data address stored in memory
datatodatareloc:
    DA 4 4:map ;

# an example of different sized fields for storing addresses
differentsizerelocs:
    DA 15 4:loopforever
          3:trynextdirection
          2:map
          1:loopforever ;

# an example of unused addresses in the middle of an address space
DATA datamemory 371 ;
data_label:
    DA 4 ; # this DA definition defines addresses 371-374

DATA datamemory 400;
data_label2:
    DA 8;  # this DA definition defines addresses 400-407
```

🟥 Comments	🟪 Directives and section headers
🟦 Constant literals	🟩 Guards
🟪 Label declarations	🟦 Label references and address space names

Appendix B

ASSEMBLY GRAMMAR

EBNF description of the grammar. Only those white spaces, which are mandatory for the parsing are shown in the grammar.

comment	:=	’#’ character* end_of_line_character
wSpace	:=	(comment | white_space_character)+
binNumber	:=	’0b’[01]+
hexNumber	:=	’0x’[0-9a-fA-F]+
uNumber	:=	’+’?[0-9]+
sNumber	:=	’-’[0-9]+
uLiteral	:=	binNumber | hexNumber | uNumber
literal	:=	uLiteral | sNumber
name	:=	[a-zA-Z_][a-zA-Z0-9_]*
fuTerm	:=	name ’.’ name (’.’ name)?
indexTerm	:=	name ’.’ (name ’.’)? uNumber
regTerm	:=	indexTerm | fuTerm
expression	:=	name ((’+’ | ’-’) uLiteral)? (’=’ literal)?
lieralOrExpression	:=	literal | expression
source	:=	regTerm | literalOrExpression
guard	:=	(’!’ | ’?’) regTerm
transport	:=	guard? wSpace source ’->’ regTerm
immediateSpec	:=	’[’ destination ’=’ literalOrExpression ’]’
move	:=	’...’ | transport
instruction	:=	’. . .’ | (move (’,’ move)*)
label	:=	name ’:’
procedureDirective	:=	’procedure’ wSpace name
globalDirective	:=	’global’ wSpace name
directive	:=	’:’ (procedureDirective | globalDirective)
codeLine	:=	label* instruction immediateSpec*
codeLines	:=	((codeLine | directive) ’;’)*
codeHeader	:=	’CODE’ (wSpace literal)? ’;’
codeArea	:=	codeHeader codeLines

initDataField	:=	(uNumber ’:’)? literalOrExpression
dataLine	:=	label* ’DA’ wSpace uNumber (wSpace initDataField)*
dataLines	:=	((dataLine | directive) ’;’)*
dataHeader	:=	’DATA’ wSpace name (wSpace literal)? ’;’
dataArea	:=	dataHeader dataLines
area	:=	dataArea | codeArea
program	:=	area+