XASM - A Generic and Retargetable Assembler

Marco Kaufmann
Department of Computer Science, Technische Universität Dresden, Germany
marco.kaufmann@inf.tu-dresden.de
January 2010

This report introduces XASM, a generic and retargetable assembler that can be used as a stand-alone tool, as well as a Java library or framework providing assembler-related Java classes for application programs. It is not necessary to modify or recompile the assembler in order to adapt it to new target architectures. Retargetability is achieved by Opcode Maps, a textual description of the instruction mnemonics and encodings of the target architecture, for which to generate machine code. Opcode maps feature an inheritance model, which makes them clearer and also means less effort in describing an instruction set because only the extensions compared to its binary compatible parent instruction set(s) need to be specified.

1 Motivation and Introduction

Indeed, there are a number of assemblers, including free open-source alternatives such as the GNU Assembler [gnu] or the Netwide Assembler [nasm]. However, they are not retargetable in the sense that XASM is. Regarding the large number of prospective target architectures, in particular, regarding Design Space Exploration, prototyping, homebrew instruction set architectures intended for the use on FPGA platforms etc., retargetability may be a highly desired feature of an assembler. In XASM, this is achieved through Opcode Maps.

The development of XASM begun when an assembler was needed for the SHAP project [Pre+07] in order to compile microcode subroutines for the SHAP cpu. From the assembler’s view, this CPU is quite an uncommon architecture because its bytes are 9 bit wide. Trying to adapt the GNU Assembler to this architecture - by modifying the source code of the assembler - caused some severe problems. Thus, conventional assemblers are not only non-retargetable, but they are also not flexible enough as they make various assumptions about the target architecture.
This document subdivides into four more sections. Section 2 introduces the XASM preprocessor, which is used for opcode maps as well as assembler programs. Section 3 describes the syntax and semantics of opcode maps. Section 4 deals with assembler programs and the XASM assembler directives. Finally, section 5 gives an overview of how to use the assembler at the command line level.

2 The XASM Preprocessor

In the workflow of the assembler, the preprocessor resides between the tokenizer and the parser. Its input is an unprepared token stream from the tokenizer, that may or may not contain preprocessor statements. Its output is a token stream, in which the preprocessor statements have been processed and replaced. Both, assembler programs and opcode maps, can contain preprocessor statements. An additional task of the preprocessor is to strip comments off the source code.

2.1 Comments

Table 1 shows the types of comments stripped by the preprocessor.

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block comments</td>
<td>/* ...*/</td>
</tr>
<tr>
<td>Line comments</td>
<td>; ...</td>
</tr>
<tr>
<td></td>
<td>// ...</td>
</tr>
</tbody>
</table>

Table 1: Comment types

2.2 Include files

Further text files can be inserted into the source code using the preprocessor statements .include and .require. Unlike .include, .require inserts the specified file only if it has not yet been inserted previously. Directories scanned for include files are the current working directory, the root directory of the assembler program to compile, as well as the root directory of the assembler. With the preprocessor statement .incpath, additional directories can be added to the search list. Listing 1 shows an example of how to use the preprocessor statements related to include files. Path names may be denoted in a DOS or UNIX like style. XASM interprets them in a platform independent way.

```plaintext
1 .include "lib/timer.inc"
2 .require "lib\uart"
3 .incpath "lib"
4 .include "timer"
```

Listing 1: Usage of include files (example)
2.3 Macros

Macros are declared with the preprocessor statement `.macro`, which is followed by the name of the macro and an optional list of parameters expected on its invocation. Macro overload is supported, that is, several macros with the same name but of different arity may be declared. In the line after the declaration, the macro body begins, which contains the token sequence that replaces every invocation of the macro. The body is closed by the preprocessor statement `.endmacro`. Single line macros are supported as well as multi line macros. When the preprocessor replaces a macro call by the body of the callee, the replacement contains just as many line breaks as the body of the macro does. Thus, single line macros can be easily provided by letting the macro body contain one single line.

Macros are invoked by the statement `%%<macro identifier>`. If the macro is of non-zero arity, a parameter list is expected to follow the statement. Parameters inside the macro body are inserted in a similar manner as if they were locally defined macros with the arity of zero inside the macro body. Listing 2 provides an example for the usage of macros. Listing 2 shows what the preprocessor generates when given listing 2 as the input.

```
1  .macro get(base, recsize, index)
2      mov eax,%%recsize
3      mul %%index
4      mov al,[eax+%%base]
5  .endmacro
6
7  .macro memref(base, offset)
8      [offset %%base + %%offset]
9  .endmacro
10
11  ...
12  %%get(@myarray,20,ecx)
13  ...
14      mov %%memref(myvar,edx),eax
15  ...
```

Listing 2: Usage of macros (example)

2.4 Conditional Compilation

The preprocessor is able to generate different token sequences depending on whether identifiers are defined or not. This is known as conditional compilation. For the purpose of conditional compilation, the preprocessor provides the statements `.def`, `.undef`, `.ifdef`, `.ifndef`, `.else` and `.endif`. The `.def` statement is of the form `.def <identifier>` and creates an empty macro of the specified name with the arity of zero, whose only purpose is to provide an identifier for the conditional compilation. Vice versa, all macros of the
... 
1. mov eax, 20
2. mul ecx
3. mov al, [eax + 0xarray]
4. ...
5. mov [offset myvar + edx], eax
6. ...

Listing 3: Token sequence generated for Listing 2

specified name, or the macro with the specified name and arity can be removed with
.undef <identifier> [:<arity>].

The syntax of the .ifdef and .ifndef statement is shown in listing 1. Listing 5 provides examples for their usage. An .ifdef statement is replaced by the contents of
its then-branch in the case the specified identifier is defined, and replaced by the contents
of its optional else-branch in the case it is not. On the contrary, the .ifndef statement
is fulfilled if the identifier is not defined.

1. ifstmt =
2. (".ifdef" | ".ifndef") ident [":" number] token* 
3. [".else" token*] ".endif"

Listing 4: Syntax for the .ifdef- and .ifndef statement

1. .ifdef win32
2. ...
3. .else
4. ...
5. ".endif
6. ...
7. .ifdef debug
8. ...
9. ".endif
10. ...

Listing 5: Conditional compilation (example)

2.5 Nested preprocessor statements

When the preprocessor replaces a preprocessor statement in the source code, the replace-
ment itself also will be preprocessed. This enables nested preprocessor statements. An
if-statement can contain further if-statements, preprocessor statements can be used in-
side a macro body, and macros may even invoke other macros. For example, we could
define a kind of while-loop-macro as shown in listing 6
3 Opcode Maps

An opcode map is a textual description of the instruction mnemonics and encodings of an instruction set for which XASM is to generate machine code. XASM is able to compile and use these opcode maps, and is, thus adaptable to new target machines with no need to modify or recompile the assembler (retargetability). Next to the actual instruction mnemonics and encodings, opcode maps include a specification of the registers and register sets, segment types, segments, memory operands and addressing modes available on the target machine. Additionally, they feature an inheritance model, which makes them more structured and clearer, also leading to less effort in describing new instruction sets.

3.1 Inheritance and the basic architecture

Inheritance in opcode maps means less effort in describing new instruction sets, because not the complete instruction set, but only its extensions compared to its parent instruction sets, which are binary downwards compatible to it, need to be specified. An instruction set hierarchy is illustrated in figure 1 using the example of the x86 architecture. As can be seen, multiple inheritance is supported. The inheritance model is as follows. An instruction set inherits all registers and register sets, segment types, segments, memory operands, addressing modes and instruction mnemonics from its leftmost parent in the extends declaration (section 3.2). Then, it successively inherits the instruction set elements of the parent to the right, which will override all already inherited segment types, segments, memory operands, addressing modes and instruction mnemonics of the same name or format. Finally, the instruction set may define its own instruction set elements, which, again, will override all already inherited elements of the same name or format.

Figure 1: x86 instruction set hierarchy

An instruction set and all its parents must share the same basic architecture. This is a collection of Java classes that back the instruction sets of the respective basic archi-
tecture and are responsible for parsing the assembler program, resolving the assembler statements and assembling the instruction words. Instruction sets that share a quite common assembly syntax and instruction word format may use the same basic architecture. In order to adapt XASM to a new instruction set with an already known basic architecture, only a new opcode map is required. To adapt XASM to instruction sets based upon a different basic architecture, it is additionally required to implement the respective Java classes for that basic architecture, which override the parsing of the assembler program, the resolving of the assembler statements or the assembling of the instruction words.

The only basic architecture available in XASM innately is the default architecture. It provides classes and methods suitable for a wide range of RISC CPUs and other instruction set architectures with a rather simple instruction word format and Intel style assembly syntax, such as the DLX architecture, the Z80 CPU, the 6502 CPU etc. For more complex basic architectures, such as the x86 architecture, the user must implement additional Java classes that override the default behavior of the assembler in the required manner. This way, both is achieved: fast and easy adaption of XASM for a wide range of target machines via opcode maps on the one hand, but also flexibility on the other hand as potential target architectures are not restricted by the capabilities of opcode maps.

3.2 General syntax

The general syntax for opcode maps is shown in listing 4. The first statement in an opcode map is the declaration of the basic architecture. The second is the instruction set description, beginning with the keyword set, followed by the identifier of the instruction set. The identifier may be used in the extends declaration of other opcode maps, as well as within the .cpu directive in assembler programs. Optionally, an additional string can be specified denoting the name which is displayed for the instruction set inside instruction set documentations automatically generated by XASM (section 5). If none is specified, the instruction set identifier is used as the displayed name. The extends declaration contains a list of the parents of the instruction set, from which to inherit instruction set elements as described in section 5.1. In braces, the declaration of the instruction set’s new elements follows, which includes a description of the registers and register sets, segment types, segments, memory operands, addressing modes, instruction prefixes and instructions, as well as their mnemonics and encodings.

3.3 Segment type and segment declaration

Segmemnts represent address spaces of the target machine, which are either real, physically separated address spaces such as the code and data memory for Harvard architectures, or which are logical address spaces that in fact map to the same memory and might also overlap each other. An example for the latter case are the segment registers of the x86 architecture. The general XASM concept about segments or address spaces is, that each addressing mode refers to a specific address space of the architecture, which is denoted in the declaration of the respective memory operand. For example, x86 addressing modes containing an ES override prefix refer to the ES address space. Another example is the
8-bit AVR instruction set [Atmel99], where the 1pm instruction refers to the code- and the 1d, 1di and 1ds instructions refer to the data memory.

Via the directive .assume in assembler programs, a segment of the program can be mapped to one of these address spaces, allowing the assembler to automatically select the matching addressing mode when accessing symbols declared in program segments. This is quite similar to the way that MASM uses the assume directive. It also will throw a compile time exception in the case no addressing mode does match, that is, the symbol resides within a program segment which currently cannot be accessed.

As the goal of XASM is adaptability for a wide range of target architectures, we want to make as few assumptions about the address spaces of an architecture as possible. Hence, opcode maps also include a description of the segment types, consisting of a number of attributes that denote the byte and word size, the maximum segment size, the addressing unit and the segment’s endianness. XASM can even deal with uncommon values of these attributes, such as 9 bits for the byte size or 3 bytes per word. Listing 8

---

```plaintext
1  segtypes =
2     "segments" "{" attrgroup* "}";
3
4  attrgroup =
5     "group" ident [string] "{" attr* "}";
6
7  attr =
8     ident "=" (number | ident);
9
10 segs =
11     "spaces" "{" seg* "}";
12
13 seg =
14     ident ident ("," ident)* [string]

Listing 8: Syntax for segment type and segment declaration
```
<table>
<thead>
<tr>
<th>attribute</th>
<th>default value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8</td>
<td>byte size in bits</td>
</tr>
<tr>
<td>word</td>
<td>2</td>
<td>word size in bytes</td>
</tr>
<tr>
<td>addressing</td>
<td>1</td>
<td>addressing unit in bytes</td>
</tr>
<tr>
<td>max</td>
<td>65536</td>
<td>maximum segment size in addressing units</td>
</tr>
<tr>
<td>endian</td>
<td>little</td>
<td>endianess (big or little)</td>
</tr>
</tbody>
</table>

Table 2: Segment type attributes

shows the syntax for the segment type and segment declaration in opcode maps, listing 9 and 10 provide examples of their usage. An overview of the segment type attributes available for the default basic architecture is given in table 2. Other basic architectures might introduce additional segment type attributes. Note, that the segment types code and data do always exist. If an opcode map does not declare them, they will be created by the assembler automatically with the default values for the attributes.

```java
1  segments {
2      group code "Code segment type" {
3          byte = 9
4          word = 1
5      }
6      group data "Data Segment type" {
7          byte = 32
8          word = 1
9      }
10  }
```

Listing 9: Declaration of segment types using the example of SHAP

```java
1  spaces {
2      cs code "Code segment"
3      ds data "Data segment"
4      ss data "Stack segment"
5      es data "Extra data segment"
6  }
```

Listing 10: Declaration of segments using the example of x86

3.4 Instruction operands

Instruction mnemonics use three kinds of instruction operands:

- register operands,
• immediate value operands and
• memory operands.

3.4.1 Registers and register groups

The declaration of the registers and register groups available for an instruction set starts with the keyword **registers**, followed by the declaration of the individual register groups in braces. A register group declaration starts with the keyword **group**, followed by the register group identifier, optionally the name that is displayed for the register group in the automatically generated instruction set documentation, and the declaration of the individual registers. The declaration of a register contains, next to its name, the register code, which is used for assembling the instruction word. The optional register code modifier specifies whether to concatenate the register code as a prefix to the instruction word (**pre**), to concatenate it as a postfix to the instruction word (**post**), or whether to add it to the instruction word (**add**). If no register code modifier is specified, **add** is used as the default.

In listing 11 the syntax for the register and register group declaration is shown. Listing 12 provides an example of their usage in opcode maps. Note, that different register groups may contain the same registers and may also assign different register codes to them. This is quite useful in the case a register is encoded in different ways depending on whether its a source or a destination operand, or depending on the type of the instruction.

```plaintext
1  regops =
2    "registers" "{" reggr* "}";
3
4  reggr =
5    "group" ident [string] "{" regdecl* "}";
6
7  regdecl =
8    ident ("," ident)* ":" number [modifier];
9
10 modifier =
11    "pre" | "post" | "add";

Listing 11: Syntax for register and register group declaration
```

3.4.2 Immediate value operands

An instruction set may contain several types of immediate value operands with different ranges and different encodings. The default basic architecture implements the immediate operand types shown in table 3. In the instruction mnemonic declaration, they are denoted in the form `<identifier><size>`, where **identifier** denotes the immediate type and **size** denotes the operand’s size in bits.
Listing 12: 8080 register declaration
<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>ID</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm&lt;n&gt;</td>
<td>0</td>
<td>Signed or unsigned immediate value in the range $-2^{n-1}$ to $2^n - 1$.</td>
</tr>
<tr>
<td>immu&lt;n&gt;</td>
<td>1</td>
<td>Unsigned immediate value in the range 0 to $2^n - 1$.</td>
</tr>
<tr>
<td>immi&lt;n&gt;</td>
<td>2</td>
<td>Two’s complement signed immediate value in the range $-2^{n-1}$ to $2^{n-1} - 1$.</td>
</tr>
<tr>
<td>rel&lt;n&gt;</td>
<td>3</td>
<td>Signed or unsigned immediate value, relative to the current program counter value, in the range $-2^{n-1}$ to $2^n - 1$.</td>
</tr>
</tbody>
</table>

Table 3: Default immediate operand types

An opcode map may define its own immediate value types for the instruction mnemonics. However, they must be backed by the basic architecture in order to do the range check and the instruction encoding. The syntax for immediate type declaration is shown in listing 13. It begins with the keyword `immediates`, followed by a list that assigns an id code to each type identifier, by which the basic architecture identifies the type. It is possible, though not useful to define different immediate value types with the same id code inside an opcode map, as the basic architecture would treat them as the same type.

```
1  immops =
2    "immediates" "{" [imm ("," imm)*] "}"
3
4  imm =
5    ident "=" number
```

Listing 13: Syntax for immediate value type declaration

3.4.3 Memory operands and addressing modes

The memory operands available in an instruction set are declared using the syntax shown in listing 14. The declaration starts with the keyword `memory`, followed by the declaration of the individual operand groups in braces. The definition of a memory operand consists of an expression describing the format of the operand in assembler statements, as well as the segment the operand refers to. The operand code and the optional modifier (section 3.4.1), which, again, is `add` by default, denote how to encode the operand in instruction words.

The expression describing the operand format can contain the operators `+`, `-` and `*`, numbers, as well as register or register group identifiers and even memory operand group identifiers. In the latter case, the instruction word will be assembled recursively by firstly applying the code of the inner operand, and then the code of the outer operand to the instruction word. Listing 15 shows the declaration of the 8086 memory operands, and also covers the rather complex mod r/m byte encoding and segment override prefixes.
3.5 Assembler statements and instruction prefixes

An assembler statement has the general form prefix instruction operands, where

- prefix denotes zero, one or more instruction prefixes,
- instruction denotes the name of the instruction
- operands is a list of operands, which are either register, immediate value or memory operands.

3.5.1 Instruction prefix declaration

The syntax for the declaration of instruction prefixes is shown in listing 14. It starts with the keyword prefixes, followed by the definition of the individual prefixes in braces. A prefix definition consists of the prefix name and the prefix code. Optionally, the instruction group that the prefix applies to can be specified, as well as the modifier that indicates whether to use the prefix code as an instruction prefix, instruction postfix or to add to the instruction word (section 3.4.1). The default behavior for instruction prefixes is pre. If no instruction group is specified, the prefix is applicable for all instructions. An example for instruction prefix declarations is shown in listing 17.

3.5.2 Instruction declaration

Unlike instruction prefixes, instructions are defined in groups. This has two reasons. The first is to restrict the use of certain instruction prefixes to certain groups of instructions. The second is to make the instruction set documentations automatically generated by XASM more readable. The syntax for the declaration of instruction groups and instructions is shown in listing 18. It starts with the keyword instructions, which contains the individual instruction groups in braces. An instruction group has an identifier and

```c
memops =
    "memory" "{" memgr* "}";

memgr =
    "group" ident [string] "{" memdecl ("," memdecl)* "}";

memdecl =
    expr =" [ident ":"] number modifier;

expr =
    atomic ("+" | "-" | ":") expr;

atomic =
    number | ident | ": [ expr ]";

Listing 14: Syntax for memory operand declaration
```
mem
{
  group dmem1 post
  {
    bx+si = 0, bx+di = 1, si = 4, di = 5, bx = 7
  }
}

group dmem
{
  [dmem1] = 0, [dmem1+imm8] = 40h,
  [dmem1+imm16] = 80h,
  [imm16] = 6 post direct
}

group smem1 post
{
  bp+si = 2, bp+di = 3
}

group smem
{
  [smem] = 0, [smem1+imm8] = 40h,
  [smem1+imm16] = 80h,
  [bp+imm8] = 46h post stack,
  [bp+imm16] = 86h post stack
}
dmem = data:0  // data segment access
smem = stack:0   // stack segment access

cs:dmem = text: 2Eh pre  // cs override
ss:dmem = stack: 36h pre  // ss override
ds:smem = data: 3Eh pre  // ds override
es:dmem = extra: 26h pre // es override
}

Listing 15: Memory operand declaration (example)

prefixes =
  "prefixes" "{" prefix " "}"
prefix =
  ident {"," ident} "=" [ident ":"] number [modifier]

Listing 16: Syntax for instruction prefix declaration
prefixes
{
  rep, repe, repz = string: 0F3h
  repne, repnz = string: 0F2h
  lock = 0F0h
}

Listing 17: 8086 instruction prefixes

an optional string which is displayed as the instruction group name in the instruction set
documentation.

The declaration of an instruction inside a group consists of the instruction name, or
a list of alternative names that refer to the same instruction, the operand list and the
instruction opcode. A list of modifiers may follow that opcode, however, none are defined
for the default basic architecture. An operand is either the identifier of a register group,
a memory operand group or an immediate operand type, denoting that in assembler
statements, a register or a memory operand of the respective group, or an immediate
operand of the respective type can be inserted for the operand. Also, a specific register or
immediate value operand may be specified, denoting that only that register or immediate
value can be insterted. As that register or immediate value is addressed implicitly, no
additional operand code is applied to the instruction word in that case. Listing [19] shows
instruction declarations using the example of some 8086 instructions.

insns =
  "instructions" "{" insngr "}";

insngr =
  "group" [ident] [string] "{" insn* "}";

insn =
  ident ("," ident)* [operand ("," operand)*]
    "=" code modifier;

operand =
  ident | number;

code =
  number ("," number)*;

Listing 18: Syntax for instruction declaration

14
group transfer "Data Transfer" {
  cbw = 0x98 o16
  cwd = 0x99 o16
  lahf = 0x9F

  lea r16,m = 0x8D o16
  lds r16,m32 = 0xC5 o16
  les r16,m32 = 0xC4 o16

  mov r8,r8 = 0x88
  mov m8,r8 = 0x88
  mov r16,r16 = 0x89 o16
  mov m16,r16 = 0x89 o16
  mov r8,r8 = 0x8A
  mov r8,m8 = 0x8A
  mov r16,r16 = 0x8B o16
  mov r16,m16 = 0x8B o16
  mov r16,sreg = 0x8C
  mov m16,sreg = 0x8C

  ...
}

Listing 19: Declaration of some 8086 instructions
4 Assembler programs

The XASM assembly syntax is quite similar to that of conventional assemblers with Intel style assembly syntax, such as MASM or NASM. There are three types of statements in assembler programs:

- assembler statements,
- assembler directives and
- label - and variable definitions.

Assembler statements are symbolic machine instructions of the target CPU that XASM will compile into machine code. The mnemonics used for their notation are defined in the opcode map of the respective target CPU, while the basic assembly syntax is Intel style rather than AT&T style. A basic architecture may override that however. Labels mark a specific address offset inside a program segment. Variables in contrast additionally reserve some space within the segment, according to their size. Finally, assembler directives instruct the assembler to perform some action, such as selecting the target CPU or using a certain value for the address adjustment.

4.1 Assembler directives

4.1.1 Target CPU

The target CPU for an assembler program can either be selected via command line arguments, or by specifying the target CPU inside the assembler program using the .cpu directive of the form .cpu <identifier>. The identifier denotes any instruction set described by an opcode map, e.g., .cpu 386. If used, the .cpu directive must be the very first statement of the assembler program.

4.1.2 Constant declaration and constant expressions

The XASM preprocessor supports macros, which already can be used for the purpose of constant declaration. Nevertheless, XASM provides an additional directive for constant declarations in assembler programs. It is of the form <identifier> equ <expression>, where identifier is the name of the constant to define, and expression the value of the constant. The expression can either be an integer number literal (table 4), the name of a constant, or a more complex expression composed of the operators shown in table 4.

4.1.3 Segments

Assembler programs can be separated into multiple segments. A new program segment is opened by the assembler directive .segment, whose form is .section <name> <type>. All labels, variables and assembler statements that follow will reside within this segment at ascending addresses, until the next segment is opened. The segment name is used to identify it in the .assume directive. The segment type denotes the attributes of the segment and is one of the segment type identifiers defined in the opcode map of the target CPU (section 5.3). Listing 21 gives an example for the usage sections.
<table>
<thead>
<tr>
<th>Notation</th>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal</td>
<td>( \text{(&quot;0&quot;..&quot;9&quot;)}^+ )</td>
<td>123</td>
</tr>
<tr>
<td>hexadecimal</td>
<td>( \text{(&quot;0&quot;..&quot;9&quot;)</td>
<td>&quot;a&quot;..&quot;f&quot;</td>
</tr>
<tr>
<td>binary</td>
<td>( \text{(&quot;0&quot;</td>
<td>1&quot;)}^+ \text{&quot;b&quot;} )</td>
</tr>
<tr>
<td>octal</td>
<td>( \text{(&quot;0&quot;..&quot;7&quot;)}^+\text{&quot;o&quot;} )</td>
<td>173o</td>
</tr>
</tbody>
</table>

Table 4: Integer number literals

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Type</th>
<th>Form</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Logical</td>
<td>binary</td>
<td>&amp;, l, ~, and, or, xor, nand, nor, xnor</td>
</tr>
<tr>
<td>1</td>
<td>Logical</td>
<td>unary</td>
<td>!, not</td>
</tr>
<tr>
<td>2</td>
<td>binary</td>
<td>binary</td>
<td>=, &lt;, &gt;, &lt;=, &gt;=, !=, &lt;&gt;</td>
</tr>
<tr>
<td>3</td>
<td>Arithmetical</td>
<td>binary</td>
<td>+, -</td>
</tr>
<tr>
<td>4</td>
<td>Arithmetical</td>
<td>binary</td>
<td>*, /, %, div, mod</td>
</tr>
<tr>
<td>4</td>
<td>Shift</td>
<td>binary</td>
<td>&lt;&lt;, &gt;&gt;, shl, shr</td>
</tr>
<tr>
<td>5</td>
<td>Arithmetical</td>
<td>binary</td>
<td>**</td>
</tr>
<tr>
<td>6</td>
<td>Arithmetical</td>
<td>unary</td>
<td>+, -</td>
</tr>
<tr>
<td>7</td>
<td>Operand size</td>
<td>unary</td>
<td>byte, word, dword, fword, qword</td>
</tr>
<tr>
<td>7</td>
<td>Miscellaneous</td>
<td>unary</td>
<td>sizeof, offset, @</td>
</tr>
</tbody>
</table>

Table 5: XASM operators

```assembly
1 .segment seg data
2   msg db "Hello, world!", 0
3
4 .segment seg data
5   ...
6
7 .segment seg code
8   mov ax, offset msg
9   ...
```

Listing 20: Segments in an assembler program (example)
4.1.4 Assume

The .assume <segment>,<section> directive is used to tell the assembler, which address spaces of the target CPU currently map to which segments of the assembler program. This enables the assembler to automatically select a suitable addressing mode when accessing variables. It, therefore, has the same meaning as the assume directive in MASM. If, for example, the accessed variable resides in a program segment that the ES segment register of the machine currently maps to, a direct addressing memory operand with ES segment override prefix will be encoded by XASM. Listing 21 provides an example for the usage of the .assume directive.

```
1 .segment code code
2   .assume ds, data
3     mov al, myvar
4 ...
5
6 .segment data data
7     myvar resb
8 ...
```

Listing 21: Example for the .assume directive

4.1.5 Address adjustment

The .org directive is used to define an address adjustment for either code addresses, data addresses or both. It is of the form .org [code|data] <adjust>, where adjust is the value to add to the addresses. With the keyword data specified, the adjustment is applied to data addresses (variables) only. With the keyword code specified, the offset is applied to code addresses (labels) only. Listing 22 shows as an example for the usage of the .org directive the startup sequence of a DOS COM file.

```
1 .segment seg code ; DOS COM -> we need to add
2   .org 100h ; 100h to each address
3     .assume cs, seg ; CS maps to seg
4
5 startup:
6     mov ax, cs
7     mov ds, ax ; let DS point to seg, too
8     .assume ds, seg ; DS maps to seg
9 ...
```

Listing 22: Example for the .org directive
4.1.6 Address alignment and zero-byte padding

Address alignment of labels, variables and assembler statements inside program segments is done by the assembler directives .align and .absolute. The .align directive is of the form .align <n> and pads zero bytes, such that the address of the label, variable or assembler statement that follows is a multiple of n. In contrast, the .absolute directive, which is of the form absolute <n>, is padding zero bytes until the address of the label, variable or assembler statement that follows it equals n. Therefore, the .absolute directive assigns a symbol an absolute address inside its segment. In the case the address of n is already exceeded when the directive is stated, XASM produces an error message. An example for the usage of the directives .align and .absolute is given in listing 26.

```assembly
1  jmp m1
2  ...
3
4  .align 16
5  m1:
6  ...
7  .absolute 0x8000
8  msg db "Hello, world!",0
```

Listing 23: Example for .align and .absolute

4.2 Labels and variables

4.2.1 Labels

A label declaration is of the form <identifier>; and marks the address of the variable or assembler statement that follows it. Whenever the label’s identifier is used inside an expression, e.g., as the immediate operand of a branch instruction, it will be replaced with the address it marks. Labels are always “short”, that is, the address contains only the offset of the label inside the segment where it is defined, but not the segment value. Unlike variables, labels do not reserve any memory space. A special label is the startup label, which marks the entry point of the program and is needed by the linker for some target formats. The listings 22 and 23 provide examples for the usage of labels.

4.2.2 Variables

Variables are declared using the define or the reserve directive, whose syntax is shown in listing 24. While reserve preserves space for the variable within the segment and initializes it with zero bytes, define allows the programmer to initialize a variable with custom values. Also, there are short forms for these two directives (table 9). Examples for variable declarations are given in listing 20.
defdecl =
    identifier "define" type
    [expression (",", expression)*]
resdecl =
    identifier "reserve" type expression
type =
    "byte" | "word" | "dword" | "fword" |
    "qword" | "tbyte" | "dqword";

Listing 24: Syntax for the declaration of variables

<table>
<thead>
<tr>
<th>Short form</th>
<th>Equivalent</th>
<th>Short Form</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>define byte</td>
<td>resb</td>
<td>reserve byte</td>
</tr>
<tr>
<td>dw</td>
<td>define word</td>
<td>resw</td>
<td>reserve word</td>
</tr>
<tr>
<td>dd</td>
<td>define dword</td>
<td>resd</td>
<td>reserve dword</td>
</tr>
<tr>
<td>df</td>
<td>define fword</td>
<td>resf</td>
<td>reserve fword</td>
</tr>
<tr>
<td>dq</td>
<td>define qword</td>
<td>resq</td>
<td>reserve qword</td>
</tr>
<tr>
<td>dt</td>
<td>define tbyte</td>
<td>rest</td>
<td>reserve tbyte</td>
</tr>
<tr>
<td>ddq</td>
<td>define dqword</td>
<td>resdq</td>
<td>reserve dqword</td>
</tr>
</tbody>
</table>

Table 6: Short forms for variable declaration

mybyte db 123
message define byte "Hello, world!"
primesnumbers dw 2,3,5,7,11,13,17,19

Listing 25: Examples for the declaration of variables
5 Usage of XASM

XASM comes as a jar file and is started with the following input on the command line shell:

```java
java -jar xasm.jar <command line arguments>
```

5.1 Compiling opcode maps

To compile an opcode map, `-opcmp <filename>` is given as the command line argument. When the opcode map has been compiled, XASM can generate machine code for instruction set architecture described in it.

5.2 Generating instruction set documentations

XASM can generate documentations for the instruction sets it supports. An instruction set documentation is generated by passing `-opdoc <identifier>` as the command line argument, where the identifier names a target instruction set architecture.

5.3 Compiling assembler programs

Assembler programs are compiled by passing the file name of the assembler program to compile as the command line argument. If no file extension is given, XASM will assume `asm` as the file extension. Some additional command line arguments are available when compiling assembler programs. They are shown in table 4.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>`--case on</td>
<td>off`</td>
</tr>
<tr>
<td>`-warnings on</td>
<td>off`</td>
</tr>
<tr>
<td><code>-cpu &lt;identifier&gt;</code></td>
<td>Target CPU. Overrides the .cpu directive in assembler programs.</td>
</tr>
<tr>
<td><code>-o &lt;filename&gt;</code></td>
<td>Output file name. Default value is the input file name with the file extension that equates the output format.</td>
</tr>
<tr>
<td><code>-tbin</code></td>
<td>Output format: flat binary.</td>
</tr>
<tr>
<td><code>-tmem</code></td>
<td>Output format: flat binary in Xilinx MEM format.</td>
</tr>
<tr>
<td><code>-tmif</code></td>
<td>Output format: flat binary in Altera MIF format.</td>
</tr>
</tbody>
</table>

Table 7: Command line options

6 Acknowledgements

The author would like to thank Thomas B. Preusser for his help in making this report, as well as Martin Zabel for his valuable ideas and suggestions on the XASM project.
References

[Atme09] ATMELE: 8-bit AVR Instruction Set, 2009

[gnu] GNU Binutils.
http://www.gnu.org/software/binutils

http://www.masm32.com

http://www.nasm.us