The DARK-Series of Virtual Machines

Ola Ågren
Ola.Agren@cs.umu.se

Technical Report, 2000
UMINF 00.15
ISSN 0348-0542
Abstract. This technical report describes the virtual machines used in the course in Computer Architecture as vehicles to teach computer concepts to the students. The report includes an introduction to the virtual assemblers, a users' guide, some example test codes and metrics on the source code and the final program. This series of virtual machines has been used at the department of Computing Science at Umeå University since autumn 1998.
# Table of Contents

1 General ......................................................................................... 1
  1 Introduction .............................................................................. 1
  1.1 Virtual Machines .............................................................. 2
2 The Different Types of Virtual Machines ................................... 3
  2.1 Design Considerations ...................................................... 3
  2.2 Accumulator Machine ...................................................... 4
  2.3 Load-Store Machine .......................................................... 4
  2.4 Memory-Memory Machine ............................................... 5
  2.5 Stack Machine ................................................................. 5
  2.6 Index Machine ................................................................. 6
3 Implementation ........................................................................... 6
  3.1 Terminal-Based Version ................................................... 7
  3.2 Java Version, 1999 ........................................................... 8
  3.3 Java Version, 2000 ........................................................... 9
4 Future Updates and Additions ................................................... 9

2 Users’ Guide .............................................................................. 11
  5 Availability .............................................................................. 11
  6 Starting the Application ....................................................... 11
  6.1 Commonalities of Both Versions ........................................ 11
  6.2 Terminal-Based Version ................................................... 11
  6.3 Java Version ........................................................................ 12
  7 Functionality ........................................................................... 12
  7.1 Java Version ........................................................................ 12
  8 User Interface ........................................................................... 13
  8.1 Terminal-Based Version ................................................... 13
  8.2 Java Version ........................................................................ 14

3 The Virtual Assemblers ............................................................. 17
  9 Notational Conventions ........................................................ 17
  10 Common Operations ............................................................ 17
  10.1 Symbolic addresses ........................................................ 17
  10.2 Comments ......................................................................... 17
  11 Accumulator Machine .......................................................... 18
  11.1 Variable Declarations and Definitions ............................. 18
  11.2 Instruction Set ............................................................... 18
  12 Index Machine ........................................................................ 22
  12.1 Instruction Set ................................................................... 22
  13 Load-Store Machine .............................................................. 25
  13.1 Registers ......................................................................... 25
  13.2 Variable Declarations and Definitions ............................. 25
13.3 Instruction Set ........................................... 25
14 Memory-Memory Machine .................................. 30
  14.1 Constant Definitions ................................. 30
  14.2 Variable Declarations and Definitions ............. 30
  14.3 Addresses ............................................ 30
  14.4 Instruction Set ....................................... 30
15 Stack Machine ............................................. 35
  15.1 Variable Declarations and Definitions .......... 35
  15.2 Instruction Set ....................................... 35

4 Appendices ..................................................... 41
A Test Codes .................................................... 41
  A.1 Accumulator Machine .................................. 41
  A.2 Index Machine ......................................... 42
  A.3 Load-Store Machine .................................... 43
  A.4 Memory-Memory Machine .............................. 43
  A.5 Stack Machine ......................................... 44
B Implementation Source Metrics ............................. 45
C Implementation Execution Time Metrics ................. 52
  C.1 Terminal-Based Version ............................... 52
  C.2 Java Version .......................................... 52
  C.3 Interpretation ......................................... 53

References ....................................................... 55

Index ........................................................... 56
List of Figures

1  Schematics of the accumulator machine. .......................... 4
2  Schematics of the load-store machine. ............................. 4
3  Schematics of the memory-memory machine. ....................... 5
4  Schematics of the stack machine. ................................. 5
5  Screenshot of the terminal-based user interface. .................. 13
6  Screenshot of the Java user interface. ............................. 15

List of Tables

1  Code sizes in lines of code for the terminal-based source code ...... 7
2  Code sizes in lines of code for the 1999 Java source code ........... 8
3  Code sizes in lines of code for the 2000 Java source code ........... 9
4  Metrics per package ............................................... 45
5  Metrics per unit .................................................. 45
6  Metrics per class .................................................. 46
7  Metrics per function ............................................... 47
8  Time measurements on terminal-based version ..................... 52
9  Time measurements on Java version ............................... 52
Acknowledgments

This project exists partly because of extra funding provided by the Department of Computing Science at Umeå University back in the summer of 1998.

I must personally thank my supervisor Jürgen Börstler for his invaluable comments on this report.

The following persons have given a lot of valuable support and feedback for the project:

- The students of the 1998 and 1999 courses in Computer Architecture.
- The teaching assistants and beta-testers:
  - André Henriksson,
  - Rikard Skelandar,
  - Markus Stoor,
  - Tobias Vesterlund, and
  - Urban Widmark.
- The final-version-testers:
  - Emma Bjurström, and
  - Tomas Ögren.
- The editor of the SIGCSE Bulletin, John Impagliazzo.
- People at the Workshop on Computer Architecture Education, especially
  - Edward F. Gehringer, North Carolina State University (workshop organizer),
  - Jon Squire, University of Maryland, Baltimore County, and
  - Mark Wolff, Georgia Institute of Technology.
- Frank Drewes, Per Lejontand and Rikard Skelandar, Umeå University, for reading and commenting the draft version of this report.

Last, but not least, I must thank my wife Anneli and my parents for their continual support.
Chapter 1

General

1 Introduction

Until the 1997 offering of the Computer Architecture at Umeå University we have had an enrollment of less than 100 students, which allowed us to have exercises and assignments that required a lot of processor time and specific hardware. The first assignment was to build an instruction tracer that executed other programs while collecting vital statistics about the program being traced (i.e., size of data used in the operations, register usage, length of basic blocks and calls, bits used in instructions, etc.). The assignment required that all students had access to machines with MIPS processors, and we had a total of 30 machines (28 SGI workstations and two SGI servers) available for the students.

Unfortunately, the course of autumn 1997 proved that the assignments had to be scrapped. Even letting the students work in pairs and setting up time slots for the pairs was deemed insufficient. Especially the second assignment that required tracing of multiple other programs and more than half an hour of CPU time per program and user put too much strain on both the users and the hardware.

One of the chapters in the textbook by Patterson and Hennessy [10] presents some architectural styles of computer processors. We wanted to highlight the differences between these styles, and planned a new assignment.

The idea was to let the students write small assembler functions for as many different architectures as we were able to lay our hands on. Unfortunately, a sufficiently large heterogeneous machine park is expensive both to acquire and maintain. Moreover, some of the architectural styles are somewhat scarce today (especially stack machines). We had to do something else, and the logical conclusion was to use virtual machines. There were three goals for the new assignment:

1. It should provide a deeper understanding of how different computer architectures behave,
2. the tools had to be easy to maintain, understand and use, and
3. the tools had to be portable to all Unix platforms available at the department (SGI Irix, Sun SunOS and IBM AIX).

A system of programs running on all available Unix platforms was created in 1998. It consisted of a virtual machine [11] together with four virtual assemblers that converted the assembler code of the different architectural styles to the virtual target machine assembler.
Unfortunately, some of the students complained that the terminal based interface was outdated and wanted a graphical user interface instead, preferably portable even to the Microsoft Windows NT machines available. Since we had already planned to use the same type of assignment in the autumn 1999 offering of the course (with an enrollment of over 120), we felt a need to supply one. The choices available at that time were to either set up a “wrapper” around the older programs\(^1\) or create a new program from scratch. A wrapper was deemed too limited, since it would require different sets of code for different platforms in order to implement the user interface. We chose to create an entirely new program written in Java [5] that reused as much of the code from the first version as possible but with a graphical user interface. We also added a fifth type of virtual assembler to both versions of the system.

This new program was used with great success during the course in 1999, but some minor errors and possible additions were found. The Java version of the system has gone through an upgrade during the summer of year 2000 leading to improved error handling capabilities, the addition of breakpoints and a new parser for one of the virtual assemblers.

1.1 Virtual Machines

The basis of virtual machinery is the actual hardware that is supposed to build up a machine. This is constructed in such a way that it “executes” the given program in the same way as an actual machine would do, but in software. Virtual machines have been used for a multitude of reasons. One use of virtual machines has been to test hardware systems prior to their actual construction (often using hardware description languages such as VHDL [4] or Verilog [9]). Another use is to give system developers the exclusive use of a machine of their own\(^2\) while still supplying a functional system for other users [11].

---

\(^1\) In the same way that *ddd* is a graphical wrapper for the *gdb* debugger.

\(^2\) Providing the user with the same interface as the underlying hardware.
2 The Different Types of Virtual Machines

There are four basic types of architectures described in the literature [10,1]: Accumulator, Load-Store, Memory-Memory and Stack. Since strict accumulator machines have not been in use for quite some time we added a fifth type that is based on it and is in use today, an Index Machine [2]. Rather than developing a virtual machine (with user interface) for each style we opted for a debugger containing all five virtual assemblers as autonomous systems.

2.1 Design Considerations

We wanted the assemblers to be as easy to learn as possible, so they were designed with very small instruction sets (between 25 and 30 instructions for each machine). It does not take much time for a user to build a functional model of each virtual assembler and the low instruction count makes it easier to see the differences between the machines. The downside to this is that sometimes two or more instructions are needed where one would be sufficient in a normal machine.

In order to keep both the number of available instructions and the instruction count in the users’ programs down in the memory-memory machine we had to use a quite complex notion of “address” (see section 14.3 on page 30). This allows multiple dereferences in each of the operands of the instructions (including the target). It does create a syntax that is somewhat harder to understand, though.

We wanted some parts of the assembler languages to be as regular as possible:

- Mnemonics that always meant basically the same thing;
- Two version of each logical connective (bitwise and integer based);
- Simplified call/return-handling; and
- Easy to parse.

We also wanted to highlight differences by simple means:

- Using branching in some assemblers and jumping in the others;
- Differences in how to create variables; and
- Some architecture-specific instructions.

We cannot deny that the extrapolations performed to expand the information given for each architecture given in [10] into the assemblers are heavily influenced by previous exposure to real-world processors.

- The accumulator machine is roughly based on Motorola 6800.
- The load-store machine is based on the MIPS processor.
- The memory-memory machine is loosely based on Motorolas 68000 and Digital Equipment Corporations (now Compaq) VAX processors.
- The stack machine is based on both the PicoJava and the Forth processor.
- The index machine is based on the Rockwell 6502 processor.

In the following sections each of the architectures is introduced shortly.
2.2 Accumulator Machine

The accumulator machine is one of the most basic processor architectures created. This type of machine has only one register (called the accumulator) that is implicitly used as one of the operands in all instructions and is the destination of all calculations. It is the target of all load instructions and the source for all store instructions.

![Fig. 1. Schematics of the accumulator machine.](image)

2.3 Load-Store Machine

The load-store machine has a fixed number of registers (in our case 32) that are used as operands and the destination of the result for operations, except for operations that transfer data between registers and memory or vice versa.

![Fig. 2. Schematics of the load-store machine.](image)
2.4 Memory-Memory Machine

A strict memory-memory machine does not have any general purpose registers. Operations in a memory-memory machine use the values in memory cells as operands and the result is stored in a memory cell.

![Fig. 3. Schematics of the memory-memory machine.](image)

2.5 Stack Machine

A strict stack machine does not have any general purpose registers at all. All data is handled in a last-in, first-out stack. Operations take their operands from the stack and the result is pushed back onto the stack, except for operations that move data from memory to stack or vice versa.

![Fig. 4. Schematics of the stack machine.](image)
2.6 Index Machine

The index machine is basically an accumulator machine with the addition of two index registers that can be used as temporary storage and/or offset register in load and store instructions. These extra operations make the machine more suitable for implementation of high-level languages and data constructs, e.g. vector operations.

We have modeled these operations after the Rockwell 6502 processor, since that processor used to be quite popular and has a user friendly instruction set.

3 Implementation

The virtual machine has been implemented twice (as terminal-based program and with a graphical user interface), with two versions of the second implementation.

The pros of the terminal-based version is the speed of the user interface, the possibility to work on any machine capable of handling VT100/ANSI code and the possibility to cut-and-paste between windows. The major drawback of the terminal-based version is the requirement for a Unix system to run the programs on.

The good things about the versions with a graphical user interface are that they can be run on any Java virtual machine and they are very easy for a novice to learn and use. The drawback being that it takes some time to update all information on the screen while running at full speed.
3.1 Terminal-Based Version

All parts of the terminal-based version had to be portable and preferably using only command line or terminal-based interfaces in order to maximize usability of the system. There had to be one (and only one) basic starting point for the system and a consistent interactive user interface of the virtual machines.

To accommodate these needs a small system of scripts and programs was devised. The main script checks the suffix of the given input file to run the correct virtual assembler for the given assembler program file (ac for accumulator, ix for index, ls for load-store, mm for memory-memory and st for stack). The output from the virtual assembler was then piped into the virtual machine that changes its behavior slightly depending on the command line arguments set by the main script. The main script handles the differences between the underlying architectures by having sets of the programs in directories with the same name as the standard Unix arch command gives.

All scripts are Bourne shell scripts and the virtual machines are written in ANSI C [7] (using curses to implement the interface), with some parts in yacc (memory-memory assembler only) and lex [3, 8].

All statistics in table 1 are based on lines of code and include lines of comments and type definitions. The total size of the entire set of source codes is approximately 90 kilobytes.

Table 1. Code sizes in lines of code for the terminal-based source code

<table>
<thead>
<tr>
<th>Program</th>
<th>C</th>
<th>lex</th>
<th>yacc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulator assembler</td>
<td>653</td>
<td>102</td>
<td>0</td>
</tr>
<tr>
<td>Index assembler</td>
<td>817</td>
<td>125</td>
<td>0</td>
</tr>
<tr>
<td>Stack assembler</td>
<td>720</td>
<td>109</td>
<td>0</td>
</tr>
<tr>
<td>Memory-memory assembler</td>
<td>318</td>
<td>70</td>
<td>620</td>
</tr>
<tr>
<td>Load-store assembler</td>
<td>794</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>Debugger and virtual machine</td>
<td>1058</td>
<td>137</td>
<td>0</td>
</tr>
<tr>
<td>Total:</td>
<td>3302</td>
<td>529</td>
<td>620</td>
</tr>
</tbody>
</table>
3.2 Java Version, 1999

In order to get portability even in the graphical user interface (which rules out C and X11) Java\(^3\) [5] was chosen as the implementation language. Another benefit of this choice was that some of the code written in C could be used with minor changes. The basic structure of both the virtual machine and most of the virtual assemblers could be retained, thereby increasing the probability that the new system would work in much the same way as the old.

Similar to the terminal-based version, the program uses the suffix of the input files to decide which virtual assembler to use to parse the source file (i.e. ac, ix, ls, mm and st, respectively).

All source code is written in Java, except for the lexical analyzers that are written in JLex (a Java implementation of lex, implemented by Elliot Berk and edited by Andrew Appel) and the memory-memory machine parser that is written in java\(_{\text{cup}}\) (by Scott Hudson, Frank Flannery and C. Scott Ananian) code [3, 8].

All statistics in the table below are based on lines of code and includes lines of comments (except for Java code). Please note that the symbol table (used by all modules) is included in the debugger and virtual machine column. The total size of the entire set of source codes is approximately 294 kilobytes.

Table 2. Code sizes in lines of code for the 1999 Java source code

<table>
<thead>
<tr>
<th>Program</th>
<th>Java</th>
<th>JLex</th>
<th>java(_{\text{cup}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugger and virtual machine</td>
<td>1045</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Accumulator module</td>
<td>299</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>Index module</td>
<td>515</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>Load-store module</td>
<td>403</td>
<td>118</td>
<td>0</td>
</tr>
<tr>
<td>Memory-memory module</td>
<td>0</td>
<td>131</td>
<td>783</td>
</tr>
<tr>
<td>Stack module</td>
<td>408</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>2670</strong></td>
<td><strong>579</strong></td>
<td><strong>783</strong></td>
</tr>
</tbody>
</table>

\(^3\) Java is a trademark of Sun Microsystems, Inc.
3.3 Java Version, 2000

The virtual assembler for the memory-memory machine in the terminal-based version was implemented partly using yacc, and java_cup was used as a substitute for yacc in the Java version of 1999. The two program packages are unfortunately not plug-and-play replacements for each other, so there was a difference in behavior in the end product. Whereas reductions are performed eagerly (as soon as a projection is fulfilled) [3, 8] in yacc, they require yet another token in java_cup. This meant that if a code contained an "end fct" (indicating that fct is the label to start execution at) at the end, execution would in the java_cup version start at the first valid operation in the file even if fct was further down in the source file.

The entire memory-memory assembler was rewritten using a recursive-descent parser in Java, correcting the problem described above. We also managed to correct some other minor problems and added breakpoint support. The table below shows the number of documented JLex and undocumented Java lines in all modules. The total size of the entire set of source codes is approximately 290 kilobytes.

<table>
<thead>
<tr>
<th>Program</th>
<th>Java</th>
<th>JLex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugger and virtual machine</td>
<td>1084</td>
<td>0</td>
</tr>
<tr>
<td>Accumulator module</td>
<td>299</td>
<td>110</td>
</tr>
<tr>
<td>Index module</td>
<td>515</td>
<td>110</td>
</tr>
<tr>
<td>Load-store module</td>
<td>403</td>
<td>118</td>
</tr>
<tr>
<td>Memory-memory module</td>
<td>558</td>
<td>131</td>
</tr>
<tr>
<td>Stack module</td>
<td>408</td>
<td>110</td>
</tr>
<tr>
<td>Total:</td>
<td>3267</td>
<td>579</td>
</tr>
</tbody>
</table>

4 Future Updates and Additions

The only major additions that we have had requests for so far are I/O-instructions, especially to files, and scripting to support the teaching assistants in their work. The updates might be added for next year's course, if they are found to be worthwhile.
Chapter 2

Users’ Guide

5 Availability

The latest Java version of the virtual machine together with some example assembler files is available for download through http://www.cs.umu.se/~ola/Dark/.

6 Starting the Application

This information is valid at the department of Computing Science at Umeå University. Please ask either your site administrator or look at http://www.cs.umu.se/~ola/Dark/ for more information if you wish to run the application elsewhere.

The terminal-based version will not be available for use elsewhere, since the source code is not public. Upon request, I might be convinced to provide executables for Suns SunOS, SGI’s Irix and IBMs AIX.

6.1 Commonalities of Both Versions

Both versions of the virtual machine use the suffix of the input files to decide which virtual assembler to use when parsing the file (i.e. ac, ix, st, mm and ls, respectively). This convention must be followed, otherwise the programs will not be compiled by the correct assembler.

Neither the assemblers, nor the virtual machines, contain any semantic checking. If any stack is filled or a pop (or ret) is performed with an empty stack the result is unspecified (probably a core dump in the terminal-based version). It is up to the user to provide correct code.

6.2 Terminal-Based Version

The programs that you will use to run the terminal-based version are available at /pub/kurser/TDBC06. To start the virtual machine just run the runlab1 script followed on the command line by the assembler file that you wish to run. If the assembler phase give no errors you will find yourself looking at the debugger.

You should either add the directory above to your PATH environment variable, or add an alias in your own .tcshrc file.4

4 alias run /pub/kurser/TDBC06/runlab1, just don’t forget to either source the file or log off and then on again afterwards.
6.3 Java Version

The program that you will use to run the version with graphical interface is available at /pub/kurser/TDBC06. To start the virtual machine just run the runlab script and the graphical user interface will pop up on the screen. You should either add the directory above to your PATH environment variable, or add an alias in your own .tcshrc file.\footnote{alias run2 /pub/kurser/TDBC06/runlab2, just don’t forget to either source the file or log off and then on again afterwards.}

7 Functionality

Both user interfaces provide the following functionalities\footnote{In the Java version only when a file has been loaded.}:

- Execute one symbolic step. Execute internal instructions\footnote{Every symbolic instruction is built up of one or more internal instructions.} until one marked with a different line number in the source file is found. Please note that this means that this function will not terminate if the source code contains jumps or branches back to the same source line.
- Run at full speed. Execute one internal instruction at a time until it either reaches a stop instruction or a non-initialized memory location. The on-screen information is continually updated during execution.

7.1 Java Version

The graphical user interface provides the following basic functionalities:

- Load an assembler file. The file will be assembled according to its suffix.
- Quit the debugger.
- Bring up an “about”-window.

There are a few other things that can be done once a program has been assembled and loaded:

- Reload the same assembler file. This makes it easy to load the same file if one wants the rerun the same source file (e.g. changed something in the source file, executed one instruction too many or run it with a different set of input data).
- Print out the current state to a file. Perfect for generating examples of test runs. Please note that no check is done to see if a file with the name already exists so it is very easy to overwrite a source file.
- Change the values of variables by clicking on them in the variable list. This will bring up a dialogue for changing the value of that variable.
- Setting and resetting breakpoints in the source file so that execution at full speed will stop at these points.
8 User Interface

8.1 Terminal-Based Version

The virtual machine debugger uses a terminal-based interface and tries to show the most interesting information at all times. The information that is presented to the user is the code line (in the source file) that is just about to be executed plus up to five lines before and after, the variables that have been declared in the source file, and finally (depending on the type of the source file given) the accumulator, the top five values on the stack, the index registers, the registers and/or the lowest 100 memory positions in the data memory.

You can press either s (step) or n (next) to execute one source instruction. To run at full speed press g (go). There is also a possibility to execute one internal instruction of the virtual machine by pressing i, but that is basically a debugging feature of the debugger (especially when one considers the number of internal steps required for a non-trivial program).

When the program has run to its completion (either executed a stop instruction or reached a non-initialized location in program memory) the debugger exits after writing the number of symbolic instructions (in the source file) and the number of internal instructions that have been executed. It is always possible to exit the debugger at any time by pressing Ctrl-C.

```
19      jtrue   endit
20      dup
21      push 1
22      sub
23      call fac
> 24      mul
25      endit ret
26      end main
```

Variables:
result: 0
parameter: 10
Stack:
24
5
6
7
8

s)tep or g)o?

Fig. 5. Screenshot of the terminal-based user interface.

---

8 It is a good idea to elongate the window so that all information will be visible.
9 It will not always show the expected line if a memory-memory instruction has been split over several lines.
8.2 Java Version

The screenshot on the next page shows the virtual machine/debugger in operation. It is currently processing a stack machine program that calculates the factorial of 10, and is just about to multiply the top two numbers on the stack.

The different parts of the user interface are (as can be seen in Figure 6):

- **Menu bar**: This gives access to the “File” menu and a help window. The commands available are “Load file”, “Reload file”, “Print to file” and “Quit”;
- **Top**: Current information and debugging commands, i.e. name of currently loaded file, number of assembler lines and internal steps executed, a “step” button (executes one assembler command), and a “go” button (execute code at full speed until a “stop”-instruction is executed or an invalid memory location is reached);
- **Left column**: Source view of the currently loaded file, with the line that is just about to be executed highlighted. Double clicking on a line will set/reset a breakpoint at the first executable line at or after the selected line, breakpoints are shown by a ‘B’ in the first column;
- **Upper right**: The variable window. It shows the current value of all variables defined in the program. Double clicking on a variable brings up a dialogue to set the value of that variable;
- **Lower right**: The miscellaneous window. It will always show non-zero memory locations in the lower part of memory, but also information that depends on the type of source code that is currently loaded:
  * **Accumulator**: For accumulator and index machine code;
  * **Index registers**: For index machine code;
  * **Stack**: For stack and load-store machine code;
  * **Non-zero registers**: For load-store machine code.
Fig. 6. Screenshot of the Java user interface.
Chapter 3

The Virtual Assemblers

9 Notational Conventions

- Anything in **courier** is a required keyword.
- Anything in **SMALL CAPS** is a required piece of data (e.g. variable names and labels) given by the programmer. They must not be the same as a keyword.
- Anything in *italic* is an optional value.
- Text in normal font in the semantic descriptions is an update of an internal (possibly hidden) register.
- \{a b\} means either a or b.
- **NUMBER** is any number.
- **VARIABLE** is any defined variable.
- &**VARIABLE** denotes the address of the given variable.
- A stack (e.g. return stack) on the left side of a \(\leftarrow\) indicates that a value is pushed on that stack, on the right side indicates that the top value of the stack is fetched and removed from the stack.

10 Common Operations

10.1 Symbolic addresses

**label**

*Syntax:* \(\text{LABEL}\)

*Semantics:* \(\text{LABEL} \leftarrow \text{memory position}\)

Binds **LABEL** to the actual address in program memory of the next assembler instruction. Labels may be placed wherever required except within instructions, e.g. `loop add 1`.

10.2 Comments

```
;
```

*Syntax:* `; text`

All text afterwards on the line will be ignored by the virtual assembler.
11 Accumulator Machine

11.1 Variable Declarations and Definitions

data
Syntax: data number label
The variable will initially hold the given number, or be undefined if none is given.

11.2 Instruction Set

add
Syntax: add \{NUMBER\|VARIABLE\}
Semantics: acc ← acc + \{NUMBER\|VARIABLE\}
The value of the accumulator is increased by the value of the numerical constant/variable.

and
Syntax: and \{NUMBER\|VARIABLE\}
Semantics: acc ← acc \& \{NUMBER\|VARIABLE\}
The accumulator is set to the logical conjunction of its current value and the value of the numerical constant/variable.

band
Syntax: band \{NUMBER\|VARIABLE\}
Semantics: acc ← acc .\& \{NUMBER\|VARIABLE\}
The accumulator is set to the bitwise logical conjunction of its current value and the value of the numerical constant/variable.

bnot
Syntax: bnot
Semantics: acc ← \neg acc
The accumulator is set to the bitwise logical negation of its current value.

bor
Syntax: bor \{NUMBER\|VARIABLE\}
Semantics: acc ← acc \lor \{NUMBER\|VARIABLE\}
The accumulator is set to the bitwise logical disjunction of its current value and the value of the numerical constant/variable.

bxor
Syntax: bxor \{NUMBER\|VARIABLE\}
Semantics: acc ← acc \xor \{NUMBER\|VARIABLE\}
The accumulator is set to the bitwise logical exclusive disjunction of the current value and the value of the numerical constant/variable.
call
Syntax: call LABEL
Semantics: return stack $\leftarrow \text{pc} + 1$
\text{pc} $\leftarrow$ LABEL
The next instruction is put on the return stack and the execution continues at LABEL.

cmp
Syntax: cmp \{NUMBER\|VARIABLE\}
Semantics: status $\leftarrow ?$
The accumulator is set to the status of a comparison between its current value and the value of the numerical constant/variable.

Note! This instruction must immediately be followed by a conditional jump instruction or the status of the comparison might get lost.

div
Syntax: div \{NUMBER\|VARIABLE\}
Semantics: acc $\leftarrow$ acc / \{NUMBER\|VARIABLE\}
The value of the accumulator is divided with the value of the numerical constant/variable.

end
Syntax: end \{\text{LABEL}\}
Semantics: pc $\leftarrow$ LABEL
End of source file. If \text{LABEL} is given it is the place at which the execution shall start after loading the program.

jeq
Syntax: jeq LABEL
Semantics: \{pc $\leftarrow$ LABEL\}
If the immediately proceeding cmp-instruction gave the result “equal” continue execution at LABEL, otherwise continue normally.

jge
Syntax: jge LABEL
Semantics: \{pc $\leftarrow$ LABEL\}
If the immediately proceeding cmp-instruction gave the result “greater than or equal” continue execution at LABEL, otherwise continue normally.

jgt
Syntax: jgt LABEL
Semantics: \{pc $\leftarrow$ LABEL\}
If the immediately proceeding cmp-instruction gave the result “greater than” continue execution at LABEL, otherwise continue normally.
\textbf{jle}

\textbf{Syntax:} jle LABEL
\textbf{Semantics:} \{pc ← LABEL\}

If the immediately proceeding \texttt{cmp}-instruction gave the result “less than or equal” continue execution at LABEL, otherwise continue normally.

\textbf{jlt}

\textbf{Syntax:} jlt LABEL
\textbf{Semantics:} \{pc ← LABEL\}

If the immediately proceeding \texttt{cmp}-instruction gave the result “less than” continue execution at LABEL, otherwise continue normally.

\textbf{jmp}

\textbf{Syntax:} jmp LABEL
\textbf{Semantics:} pc ← LABEL

The execution continues at LABEL.

\textbf{jne}

\textbf{Syntax:} jne LABEL
\textbf{Semantics:} \{pc ← LABEL\}

If the immediately proceeding \texttt{cmp}-instruction gave the result “not equal” continue execution at LABEL, otherwise continue normally.

\textbf{load}

\textbf{Syntax:} load \{NUMBER|VARIABLE\}
\textbf{Semantics:} acc ← \{NUMBER|VARIABLE\}

The accumulator is loaded with the value of the numerical constant/variable.

\textbf{mod}

\textbf{Syntax:} mod \{NUMBER|VARIABLE\}
\textbf{Semantics:} acc ← acc \mod \{NUMBER|VARIABLE\}

The accumulator is set to the remainder of its current value divided by the value of the numerical constant/variable.

\textbf{mul}

\textbf{Syntax:} mul \{NUMBER|VARIABLE\}
\textbf{Semantics:} acc ← acc \times \{NUMBER|VARIABLE\}

The value of the accumulator is set to the product of its current value and the value of the numerical constant/variable.

\textbf{not}

\textbf{Syntax:} not
\textbf{Semantics:} acc ← \neg acc

The accumulator is set to the logical negation of its current value.
or
Syntax: or \{NUMBER\}|\{VARIABLE\}
Semantics: acc ← acc \lor \{NUMBER\}|\{VARIABLE\}
The accumulator is set to the logical disjunction of its current value and the value of the numerical constant/variable.

ret
Syntax: ret
Semantics: pc ← return stack
The execution returns from a subroutine; this requires a matching call instruction.

stop
Syntax: stop
Semantics: -
Stops the execution of the program.

store
Syntax: store \{VARIABLE\}
Semantics: \{VARIABLE\} ← acc
The value of the accumulator is stored in the variable.

sub
Syntax: sub \{NUMBER\}|\{VARIABLE\}
Semantics: acc ← acc - \{NUMBER\}|\{VARIABLE\}
The value of the accumulator is decreased by the value of the numerical constant/variable.

xor
Syntax: xor \{NUMBER\}|\{VARIABLE\}
Semantics: acc ← acc \oplus \{NUMBER\}|\{VARIABLE\}
The accumulator is set to the logical exclusive disjunction of its current value and the value of the numerical constant/variable.
12 Index Machine

The index machine contains all instructions of the accumulator machine; the following instructions are the additions.

12.1 Instruction Set

**loadx**

**Syntax:** `loadx {NUMBER VARIABLE}

**Semantics:** `X ← {NUMBER VARIABLE}

Index register X is loaded with the value of the numerical constant/variable.

**loady**

**Syntax:** `loady {NUMBER VARIABLE}

**Semantics:** `Y ← {NUMBER VARIABLE}

Index register Y is loaded with the value of the numerical constant/variable.

**loadax**

**Syntax:** `loadax VARIABLE

**Semantics:** `X ← &VARIABLE

Index register X is loaded with the address of the variable.

**loaday**

**Syntax:** `loaday VARIABLE

**Semantics:** `Y ← &VARIABLE

Index register Y is loaded with the address of the variable.

**loadix**

**Syntax:** `loadix VARIABLE

**Semantics:** `acc ← mem[&VARIABLE + X]

The value of the accumulator is fetched from the address of the variable + index register X.

**loadiy**

**Syntax:** `loadiy VARIABLE

**Semantics:** `acc ← mem[&VARIABLE + Y]

The value of the accumulator is fetched from the address of the variable + index register Y.

**loadixy**

**Syntax:** `loadixy VARIABLE

**Semantics:** `acc ← mem[mem[&VARIABLE + X] + Y]

The value of the accumulator is fetched from the memory cell pointed to by (content of the memory cell pointed to by the address of the variable + index register X) and the index register Y.
storex
Syntax: storex VARIABLE
Semantics: VARIABLE ← X
The variable is set to the current value of index register X.

storey
Syntax: storey VARIABLE
Semantics: VARIABLE ← Y
The variable is set to the current value of index register Y.

storeix
Syntax: storeix VARIABLE
Semantics: mem[VARIABLE + X] ← acc
The memory position (the address of the variable + index register X) is set to the current value of the accumulator.

storeiy
Syntax: storeiy VARIABLE
Semantics: mem[VARIABLE + Y] ← acc
The memory position (the address of the variable + index register Y) is set to the current value of the accumulator.

storeixy
Syntax: storeixy VARIABLE
The memory cell pointed to by (content of the memory cell pointed to by the address of the variable + index register X) and the index register Y is set to the current value of the accumulator.

tax
Syntax: tax
Semantics: X ← acc
The value of the accumulator is copied to index register X.

tay
Syntax: tay
Semantics: Y ← acc
The value of the accumulator is copied to index register Y.

txa
Syntax: txa
Semantics: acc ← X
The value of index register X is copied to the accumulator.
txy
Syntax: txy
Semantics: Y ← X
  The value of index register X is copied to index register Y.

ty
Syntax: tya
Semantics: acc ← Y
  The value of index register Y is copied to the accumulator.

tyx
Syntax: tyx
Semantics: X ← Y
  The value of index register Y is copied to index register X.

incx
Syntax: incx
Semantics: X ← X + 1
  The value of index register X is increased by one.

incy
Syntax: incy
Semantics: Y ← Y + 1
  The value of index register Y is increased by one.

decx
Syntax: decx
Semantics: X ← X - 1
  The value of index register X is decreased by one.

decy
Syntax: decy
Semantics: Y ← Y - 1
  The value of index register Y is decreased by one.
13 Load-Store Machine

There are 32 registers available, but three of them are special and should be used with caution:

1. Register 0 (zero) is hardwired to the value zero and cannot be changed;
2. Register 1 is reserved for the assembler in order to implement some complex pseudo instructions; and finally
3. Register 31 (sp) is used as a stack pointer and points to the current top of the stack that grows downwards in data memory.

13.1 Registers

$ Syntax: $\text{NUMBER}$
   The number specifies which register to use.

sp Syntax: sp
   Symbolic name for register 31.

zero Syntax: zero
   Symbolic name for register 0.

13.2 Variable Declarations and Definitions

data Syntax: data \text{NUMBER} \label
   The variable will initially hold the given number, or be undefined if none is given.

13.3 Instruction Set

add Syntax: add \text{REGISTER,REGISTER,\{REGISTER\text{NUMBER,\text{VARIABLE}}}\}
   Semantics: register ← register + \{register\text{NUMBER,\text{VARIABLE}}\}
   The target register is set to the sum of the second register and the last value.

and Syntax: and \text{REGISTER,REGISTER,\{REGISTER\text{NUMBER,\text{VARIABLE}}}\}
   Semantics: register ← register \land \{register\text{NUMBER,\text{VARIABLE}}\}
   The target register is set to the value of the logical conjunction of the second register and the last value.
\textbf{band} \\
\textbf{Syntax:} \quad \texttt{band REGISTER,REGISTER,\{REGISTER|NUMBER|VARIABLE\}} \\
\textbf{Semantics:} \quad \texttt{register} \leftarrow \texttt{register} \land \{\texttt{register|NUMBER|VARIABLE}\} \\
\quad The target register is set to the value of the bitwise logical conjunction of the second register and the last value.

\textbf{bnot} \\
\textbf{Syntax:} \quad \texttt{bnot REGISTER,REGISTER} \\
\textbf{Semantics:} \quad \texttt{register} \leftarrow \neg \texttt{register} \\
\quad The target register is set to the bitwise logical negation of the second register.

\textbf{bor} \\
\textbf{Syntax:} \quad \texttt{bor REGISTER,REGISTER,\{REGISTER|NUMBER|VARIABLE\}} \\
\textbf{Semantics:} \quad \texttt{register} \leftarrow \texttt{register} \lor \{\texttt{register|NUMBER|VARIABLE}\} \\
\quad The target register is set to the value of the bitwise logical disjunction of the second register and the last value.

\textbf{bxor} \\
\textbf{Syntax:} \quad \texttt{bxor REGISTER,REGISTER,\{REGISTER|NUMBER|VARIABLE\}} \\
\textbf{Semantics:} \quad \texttt{register} \leftarrow \texttt{register} \oplus \{\texttt{register|NUMBER|VARIABLE}\} \\
\quad The target register is set to the value of the bitwise logical exclusive disjunction of the second register and the last value.

\textbf{call} \\
\textbf{Syntax:} \quad \texttt{call LABEL} \\
\textbf{Semantics:} \quad \texttt{return stack} \leftarrow \texttt{pc} \\
\quad \quad \texttt{pc} \leftarrow \texttt{LABEL} \\
\quad The next instruction is put on the return stack and the execution continues at \texttt{LABEL}.

\textbf{dec} \\
\textbf{Syntax:} \quad \texttt{dec REGISTER} \\
\textbf{Semantics:} \quad \texttt{register} \leftarrow \texttt{register} - 1 \\
\quad The target register is decreased by one.

\textbf{div} \\
\textbf{Syntax:} \quad \texttt{div REGISTER,REGISTER,\{REGISTER|NUMBER|VARIABLE\}} \\
\textbf{Semantics:} \quad \texttt{register} \leftarrow \texttt{register} / \{\texttt{register|NUMBER|VARIABLE}\} \\
\quad The target register is set to the quotient of the second register and the last value.

\textbf{end} \\
\textbf{Syntax:} \quad \texttt{end \{|LABEL\}} \\
\textbf{Semantics:} \quad \texttt{pc} \leftarrow \texttt{LABEL} \\
\quad End of source file. If \texttt{LABEL} is given it is the place at which the execution shall start after loading the program.
inc
Syntax: \texttt{inc} \texttt{REGISTER}
Semantics: \texttt{register} \leftarrow \texttt{register} + 1
The target register is increased by one.

ejq
Syntax: \texttt{jeq} \texttt{REGISTER},\texttt{REGISTER},\texttt{LABEL}
Semantics: \{\texttt{pc} \leftarrow \texttt{LABEL}\}
If the values of the registers are equal continue execution at \texttt{LABEL}, otherwise continue normally.

ej
Syntax: \texttt{jeq} \texttt{REGISTER},\texttt{REGISTER},\texttt{LABEL}
Semantics: \{\texttt{pc} \leftarrow \texttt{LABEL}\}
If the value of the first register is greater than or equal to the value of the second register continue execution at \texttt{LABEL}, otherwise continue execution.

jgt
Syntax: \texttt{jgt} \texttt{REGISTER},\texttt{REGISTER},\texttt{LABEL}
Semantics: \{\texttt{pc} \leftarrow \texttt{LABEL}\}
If the value of the first register is greater than the value of the second register continue execution at \texttt{LABEL}, otherwise continue normally.

jle
Syntax: \texttt{jle} \texttt{REGISTER},\texttt{REGISTER},\texttt{LABEL}
Semantics: \{\texttt{pc} \leftarrow \texttt{LABEL}\}
If the value of the first register is less than or equal to the value of the second register continue execution at \texttt{LABEL}, otherwise continue execution.

jlt
Syntax: \texttt{jlt} \texttt{REGISTER},\texttt{REGISTER},\texttt{LABEL}
Semantics: \{\texttt{pc} \leftarrow \texttt{LABEL}\}
If the value of the first register is less than the value of the second register continue execution at \texttt{LABEL}, otherwise continue normally.

jmp
Syntax: \texttt{jmp} \texttt{LABEL}
Semantics: \texttt{pc} \leftarrow \texttt{LABEL}
The execution continues at \texttt{LABEL}.

jne
Syntax: \texttt{jne} \texttt{REGISTER},\texttt{REGISTER},\texttt{LABEL}
Semantics: \{\texttt{pc} \leftarrow \texttt{LABEL}\}
If the values of the registers are not equal continue execution at \texttt{LABEL}, otherwise continue normally.
**load**

**Syntax:**  \( \text{load} \ \text{REGISTER}, \{\text{REGISTER|NUMBER|NUMBER|VARIABLE}\} \)

**Semantics:** \( \text{register} \leftarrow \{\text{mem|REGISTER+NUMBER|NUMBER|VARIABLE}\} \)

The target register is either set to the immediate number given, the value of the variable or the value of the memory position pointed to by the sum of the register and the number.

**mod**

**Syntax:**  \( \text{mod} \ \text{REGISTER|REGISTER}, \{\text{REGISTER|NUMBER|VARIABLE}\} \)

**Semantics:** \( \text{register} \leftarrow \text{register} \mod \{\text{register | NUMBER | VARIABLE}\} \)

The target register is set to the remainder of the second register divided by the last value.

**mov**

See **mv**.

**mul**

**Syntax:**  \( \text{mul} \ \text{REGISTER|REGISTER}, \{\text{REGISTER|NUMBER|VARIABLE}\} \)

**Semantics:** \( \text{register} \leftarrow \text{register} \times \{\text{register | NUMBER | VARIABLE}\} \)

The target register is set to the product of the second register and the last value.

**mv**

**Syntax:**  \( \{\text{mv|mov}\} \ \text{REGISTER|REGISTER} \)

**Semantics:** \( \text{register} \leftarrow \text{register} \)

The target register is set to the value of the second register.

**not**

**Syntax:**  \( \text{not} \ \text{REGISTER|REGISTER} \)

**Semantics:** \( \text{register} \leftarrow \neg \text{register} \)

The target register is set to the logical negation of the second register.

**or**

**Syntax:**  \( \text{or} \ \text{REGISTER|REGISTER}, \{\text{REGISTER|NUMBER|VARIABLE}\} \)

**Semantics:** \( \text{register} \leftarrow \text{register} \lor \{\text{register | NUMBER | VARIABLE}\} \)

The target register is set to the value of the logical disjunction of the second register and the last value.

**ret**

**Syntax:**  \( \text{ret} \)

**Semantics:** \( \text{pc} \leftarrow \text{return stack} \)

The execution returns from a subroutine; this requires a matching call instruction.
stop
Syntax:  stop
Semantics: -
Stops the execution of the program.

store
Syntax:  store REGISTER,{REGISTER,NUMBER|VARIABLE}
Semantics:  {mem[register+NUMBER]|VARIABLE} ← register
The register is stored in the given variable/memory location pointed to by the sum of the register and the number.

sub
Syntax:  sub REGISTER,REGISTER,{REGISTER|NUMBER|VARIABLE}
Semantics:  register ← register - {register|NUMBER|VARIABLE}
The target register is set to the difference of the second register and the last value.

xor
Syntax:  xor REGISTER,REGISTER,{REGISTER|NUMBER|VARIABLE}
Semantics:  register ← register ⊕ {register|NUMBER|VARIABLE}
The target register is set to the value of the logical exclusive disjunction of the second register and the last value.
14 Memory-Memory Machine

14.1 Constant Definitions


cost/equ
Syntax: \texttt{LABEL \{const|equ\} number}
Semantics: \texttt{LABEL \leftarrow number}

Defines a symbolic constant in the same way as \texttt{#define A 7} does in C. It can be used anywhere where a number can be used.

14.2 Variable Declarations and Definitions

mem
Syntax: \texttt{LABEL mem \{NUMBER\|(VARIABLE)\}}
Semantics: \texttt{ADD\leftarrow \{LABEL \{ADDR\}\{ADDR\{ADDR\NUMBER\}\}}

The variable will initially hold the given number, or be undefined if none is given.

stack
Syntax: \texttt{LABEL stack number}
Semantics: \texttt{LABEL \leftarrow \&\text{variable}}

This directive allocates memory for \texttt{number} addresses (positive) and sets the \texttt{LABEL}-variable contains the address to the first address in the stack (itself).

14.3 Addresses

Semantics: \texttt{ADDR\leftarrow \{LABEL \{ADDR\}\{ADDR\{ADDR\NUMBER\}\}}

label
Syntax: \texttt{LABEL}
Semantics: \texttt{\leftarrow \&LABEL}

Denotes the address of the variable.

() Syntax: \texttt{( ADDR} \}
Semantics: \texttt{\leftarrow mem[ADDR]}

Denotes the content of the memory address of the given value, dereference.

( ) Syntax: \texttt{( ADDR, ADDR\NUMBER} \}
Semantics: \texttt{\leftarrow ADDR+\{mem[ADDR]\NUMBER}}

Calculates an address using displacement.

14.4 Instruction Set

add
Syntax: \texttt{add ADDR\{ADDR\NUMBER}}
Semantics: \texttt{mem[ADDR] \leftarrow mem[ADDR] + \{mem[ADDR]\NUMBER}}

The target address is set to the sum of the second address and the last value.
and
Syntax: \text{and ADDR,ADDR\{ADDR\|NUMBER\}}
Semantics: \text{mem[ADDR] \leftarrow mem[ADDR] \land \{mem[ADDR]\|NUMBER\}}

The target address is set to the value of the logical conjunction of the second address and the last value.

\textbf{band}

Syntax: \text{band ADDR,ADDR\{ADDR\|NUMBER\}}
Semantics: \text{mem[ADDR] \leftarrow mem[ADDR] \land \{mem[ADDR]\|NUMBER\}}

The target address is set to the value of the bitwise logical conjunction of the second address and the last value.

\textbf{beq}

Syntax: \text{beq ADDR,\{ADDR\|NUMBER\},LABEL}
Semantics: \{\text{pc} \leftarrow \text{LABEL}\}

If the value of the two first operands are equal continue execution at LABEL, otherwise continue normally.

\textbf{bge}

Syntax: \text{bge ADDR,\{ADDR\|NUMBER\},LABEL}
Semantics: \{pc \leftarrow \text{LABEL}\}

If the value of the first address is greater than or equal to the value of the second operand continue execution at LABEL, otherwise continue execution.

\textbf{bgt}

Syntax: \text{bgt ADDR,\{ADDR\|NUMBER\},LABEL}
Semantics: \{pc \leftarrow \text{LABEL}\}

If the value of the first address is greater than the value of the second operand continue execution at LABEL, otherwise continue normally.

\textbf{ble}

Syntax: \text{ble ADDR,\{ADDR\|NUMBER\},LABEL}
Semantics: \{pc \leftarrow \text{LABEL}\}

If the value of the first address is less than or equal to the value of the second operand continue execution at LABEL, otherwise continue execution.

\textbf{blt}

Syntax: \text{blt ADDR,\{ADDR\|NUMBER\},LABEL}
Semantics: \{pc \leftarrow \text{LABEL}\}

If the value of the first address is less than the value of the second operand continue execution at LABEL, otherwise continue normally.
bne
Syntax:  bne ADDR,{ADDR|NUMBER},LABEL
Semantics:  {pc ← LABEL}
    If the value of the two first operands are not equal continue execution at
LABEL, otherwise continue normally.

bnot
Syntax:  bnot ADDR,ADDR
Semantics:  mem[ADDR] ← ¬ mem[ADDR]
    The target address is set to the bitwise logical negation of the second address.

bor
Syntax:  bor ADDR,ADDR,{ADDR|NUMBER}
    The target address is set to the value of the bitwise logical disjunction of the
second address and the last value.

bra
Syntax:  bra LABEL
Semantics:  pc ← LABEL
    The execution continues at LABEL.

bxor
Syntax:  bxor ADDR,ADDR,{ADDR|NUMBER}
    The target address is set to the value of the bitwise logical exclusive disjunc-
tion of the second address and the last value.

call
Syntax:  call LABEL
Semantics:  return stack ← pc
            pc ← LABEL
    The next instruction is put on the return stack and the execution continues
at LABEL.

div
Syntax:  div ADDR,ADDR,{ADDR|NUMBER}
    The target address is set to quotient of the second address and the last value.

end
Syntax:  end {LABEL}
Semantics:  pc ← LABEL
    End of source file. If LABEL is given it is the place at which the execution
shall start after loading the program.
**mod**

**Syntax:**  \( \text{mod } \text{ADDR,ADDR,} \{\text{ADDR}\text{NUMBER}\}\)

**Semantics:** \( \text{mem}[\text{ADDR}] \leftarrow \text{mem}[\text{ADDR}] \mod \{\text{mem}[\text{ADDR}] \text{NUMBER}\} \)

The target address is set to the remainder of a division of the second address and the last value.

**mov**

See \( \text{mv} \).

**mul**

**Syntax:**  \( \text{mul } \text{ADDR,ADDR,} \{\text{ADDR}\text{NUMBER}\}\)

**Semantics:** \( \text{mem}[\text{ADDR}] \leftarrow \text{mem}[\text{ADDR}] \times \{\text{mem}[\text{ADDR}] \text{NUMBER}\} \)

The target address is set to the product of the second address and the last value.

**mv**

**Syntax:**  \( \{\text{mv|mov}\} \text{ADDR,ADDR}\)

**Semantics:** \( \text{mem}[\text{ADDR}] \leftarrow \text{mem}[\text{ADDR}] \)

The target address is set to the value of the second address.

**not**

**Syntax:**  \( \text{not } \text{ADDR,ADDR}\)

**Semantics:** \( \text{mem}[\text{ADDR}] \leftarrow \neg \text{mem}[\text{ADDR}] \)

The target address is set to the logical negation of the second address.

**or**

**Syntax:**  \( \text{or } \text{ADDR,ADDR,} \{\text{ADDR}\text{NUMBER}\}\)

**Semantics:** \( \text{mem}[\text{ADDR}] \leftarrow \text{mem}[\text{ADDR}] \lor \{\text{mem}[\text{ADDR}] \text{NUMBER}\} \)

The target address is set to the value of the logical disjunction of the second address and the last value.

**ret**

**Syntax:**  \( \text{ret}\)

**Semantics:** \( \text{pc} \leftarrow \text{return stack} \)

The execution returns from a subroutine; this requires a matching call instruction.

**stop**

**Syntax:**  \( \text{stop}\)

**Semantics:**  

Stops the execution of the program.
sub
Syntax: \texttt{sub ADDR,ADDR,\{ADDR\}[NUMBER]}
Semantics: \texttt{mem[ADDR] \leftarrow mem[ADDR] - \{mem[ADDR]\}[NUMBER]}

The target address is set to the difference of the second address and the last value.

xor
Syntax: \texttt{xor ADDR,ADDR,\{ADDR\}[NUMBER]}
Semantics: \texttt{mem[ADDR] \leftarrow mem[ADDR] \oplus \{mem[ADDR]\}[NUMBER]}

The target address is set to the value of the logical exclusive disjunction of the second address and the last value.
15 Stack Machine

15.1 Variable Declarations and Definitions

data
Syntax: data number LABEL
The variable will initially hold the given number, or be undefined if none is given.

area
Syntax: area NUMBER
Reserves NUMBER of memory locations after the last declared variable.

15.2 Instruction Set

add
Syntax: add
Semantics: stack ← stack + stack
The two top values of the stack are added and the sum is put on the stack in their place.

and
Syntax: and
Semantics: stack ← stack ∧ stack
The logical conjunction of the top two values of the stack is calculated and is put on the stack in their place.

band
Syntax: band
Semantics: stack ← stack .∧ stack
The bitwise logical conjunction of the top two values of the stack is calculated and is put on the stack in their place.

bnot
Syntax: bnot
Semantics: stack ← .¬ stack
The bitwise logical negation of the top value of the stack is calculated and is put on the stack in its place.

bor
Syntax: bor
Semantics: stack ← stack .∨ stack
The bitwise logical disjunction of the top two values of the stack is calculated and is put on the stack in their place.
**The bitwise logical exclusive disjunction of the top two values of the stack is calculated and is put on the stack in their place.**

**The next instruction is put on the return stack and the execution continues at LABEL.**

**The second top value of the stack are divided by the topmost one and the quotient is put on the stack in their place.**

**Removes the top value from the stack.**

**Duplicates the top value on the stack.**

**End of source file. If LABEL is given it is the place at which the execution shall start after loading the program.**

**If the top two values of the stack are equal put a one in their place, otherwise put a zero.**
ge

Syntax:  \texttt{ge \texttt{LABEL}}
Semantics:  \texttt{tmp} ← \texttt{stack}  \\
\hspace{1em} \texttt{stack} ← \texttt{stack} ≥ \texttt{tmp}  \\
\hspace{1em} \text{If the second value on the stack is equal to or greater than the top put a one}  \\
\hspace{1em} \text{in their place, otherwise put a zero.}

gt

Syntax:  \texttt{gt}
Semantics:  \texttt{tmp} ← \texttt{stack}  \\
\hspace{1em} \texttt{stack} ← \texttt{stack} > \texttt{tmp}  \\
\hspace{1em} \text{If the second value on the stack is greater than the top put a one in their}  \\
\hspace{1em} \text{place, otherwise put a zero.}

jfalse

Syntax:  \texttt{jfalse \texttt{LABEL}}
Semantics:  \texttt{tmp} ← \texttt{stack}  \\
\hspace{1em} \{ \texttt{pc} ← \texttt{LABEL} \}  \\
\hspace{1em} \text{If the top value in the stack is a zero continue execution at \texttt{LABEL}, otherwise}  \\
\hspace{1em} \text{continue normally.}

jmp

Syntax:  \texttt{jmp \texttt{LABEL}}
Semantics:  \texttt{pc} ← \texttt{LABEL}  \\
\hspace{1em} \text{The execution continues at \texttt{LABEL}.}

jtrue

Syntax:  \texttt{jtrue \texttt{LABEL}}
Semantics:  \texttt{tmp} ← \texttt{stack}  \\
\hspace{1em} \{ \texttt{pc} ← \texttt{LABEL} \}  \\
\hspace{1em} \text{If the top value in the stack is not zero continue execution at \texttt{LABEL}, otherwise}  \\
\hspace{1em} \text{continue normally.}

le

Syntax:  \texttt{le}
Semantics:  \texttt{tmp} ← \texttt{stack}  \\
\hspace{1em} \texttt{stack} ← \texttt{stack} ≤ \texttt{tmp}  \\
\hspace{1em} \text{If the second value on the stack is equal to or less than the top put a one in}  \\
\hspace{1em} \text{their place, otherwise put a zero.}
lt
  Syntax:  lt
  Semantics: tmp ← stack
             stack ← tmp > stack
           If the second value on the stack is less than the top put a one in their place, otherwise put a zero.

mod
  Syntax:  mod
  Semantics: tmp ← stack
             stack ← stack \ tmp
          The second top value of the stack are divided by the topmost one and the remainder is put on the stack in their place.

mul
  Syntax:  mul
  Semantics: stack ← stack * stack
           The two top values of the stack are multiplied and the product is put on the stack in their place.

ne
  Syntax:  ne
  Semantics: stack ← stack ≠ stack
           If the top two values of the stack are equal put a zero in their place, otherwise put a one.

not
  Syntax:  not
  Semantics: stack ← ¬ stack
           The logical negation of the top value of the stack is calculated and is put on the stack in its place.

or
  Syntax:  or
  Semantics: stack ← stack ∨ stack
           The logical disjunction of the top two values of the stack is calculated and is put on the stack in their place.

pop
  Syntax:  pop VARIABLE
  Semantics: VARIABLE ← stack
           The top value on the stack is stored in VARIABLE.
popa
Syntax: popa
Semantics: \( \text{tmp} \leftarrow \text{stack} \)
\( \text{mem[\text{tmp}]} \leftarrow \text{stack} \)
The topmost value on the stack is used as an address to store the next value of the stack in.

pull
Syntax: pull
Semantics: \( \text{stack} \leftarrow \text{mem[stack]} \)
The topmost value on the stack is used as an address to fetch a value to put on the stack.

push
Syntax: push \{NUMBER\|VARIABLE\}
Semantics: \( \text{stack} \leftarrow \{\text{NUMBER}\|\text{VARIABLE}\} \)
The value of the variable/constant is put on the stack.

pusha
Syntax: pusha VARIABLE
Semantics: \( \text{stack} \leftarrow \&\text{VARIABLE} \)
The address of the variable is put on the stack.

ret
Syntax: ret
Semantics: \( \text{pc} \leftarrow \text{return stack} \)
The execution returns from a subroutine; this requires a matching call instruction.

rot
Syntax: rot
Semantics: \( \text{tmp1} \leftarrow \text{stack} \)
\( \text{tmp2} \leftarrow \text{stack} \)
\( \text{tmp3} \leftarrow \text{stack} \)
\( \text{stack} \leftarrow \text{tmp1} \)
\( \text{stack} \leftarrow \text{tmp3} \)
\( \text{stack} \leftarrow \text{tmp2} \)
The three top values are rotated in such a way that the current top value becomes the third value.

stop
Syntax: stop
Semantics: -
Stops the execution of the program.
 sub
 Syntax: sub
 Semantics: \( \text{tmp} \leftarrow \text{stack} \)
 \[
 \text{stack} \leftarrow \text{stack} - \text{tmp}
\]
The two top values of the stack are subtracted and the difference of the two
is put on the stack in their place.

 swap
 Syntax: swap
 Semantics: \( \text{tmp1} \leftarrow \text{stack} \)
 \[
 \text{tmp2} \leftarrow \text{stack} \\
 \text{stack} \leftarrow \text{tmp1} \\
 \text{stack} \leftarrow \text{tmp2}
\]
The two top values on the stack are exchanged.

 xor
 Syntax: xor
 Semantics: \( \text{stack} \leftarrow \text{stack} \oplus \text{stack} \)
 The logical exclusive disjunction of the top two values of the stack is calcu-
lated and is put on the stack in their place.
Chapter 4

Appendices

A Test Codes

These test codes are not examples of entire programs, they have but two purposes. Their first use is testing parsing and internal code generation. Their second use is to give the users some idea about what can be done with the special instructions of each machine.

A.1 Accumulator Machine

data 12 hubba
data bubba
main load 1
add hubba
add 7
store bubba
cmp 10
jgt yes
endit stop
yes cmp 20
jle endit
load 2
stop
end main
A.2 Index Machine

data 12 hubba
data  bubba
main
  load 1
tax
  loadix  hubba
  loaday  bubba
  loadx  hubba
  add  hubba
  add  7
tay
  store  bubba
  cmp  10
  jgt  yes
endit  tyx
  storex  hubba
  storey  bubba
  storeix  hubba
  storeiy  bubba
  load 1
  loadix  hubba
  load 1
  loadiy  bubba
  storeixy  bubba
  load 1
  loadixy  bubba
  stop
yes
  cmp  20
  jle  endit
  load  2
  stop
  end main
A.3 Load-Store Machine

data 72 hubba
data bubba
main load $2,$0,0
    load $3,$2,98
    mul $4,$3,$2
    store $4,hubba
    bxor $5,$3,$2
    add $0,$3,$2
    mv $6,$4
    stop
end main

A.4 Memory-Memory Machine

; Create some variables
A mem
B mem 1
C mem (C)

; and a constant
D const 7

; Try some instructions
start add ((C,0)),B,1
    add C,C,D
    add A,B,C
    mul A,A,A
    mov C,7
    mov B,A
    call snurr
    bge A,B,snurr2
    stop
snurr2 ble A,B,snurr3
    stop
snurr3 stop
snurr mov C,2
    ret

; Make sure that the program execution starts at the right spot
end start
A.5 Stack Machine

data 12 hubba
data bubba
area 11
data gubba
main push hubba
dup
pusha bubba
add
push 1
rot
popa
call do0
jmp quits
stop
do0
dup
pop bubba
dup
ne
jfalse true
ret
true ret
quits stop
end main
B Implementation Source Metrics

This section contains metrics that are valid for the 2000 version in Java (as of 19th of June 2000) and has been collected using JavaNCSS\textsuperscript{10} version 7.21. The tables in this section contain metrics not only for source code written by hand in Java, but also for code generated by JLex. NCSS is non-comment source statements.

Table 4 contains the number of classes, functions and source statements for the six packages in the program.

<table>
<thead>
<tr>
<th>Package</th>
<th>Classes</th>
<th>Functions</th>
<th>NCSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dark</td>
<td>12</td>
<td>69</td>
<td>1084</td>
</tr>
<tr>
<td>Dark Accu</td>
<td>5</td>
<td>22</td>
<td>562</td>
</tr>
<tr>
<td>Dark Index</td>
<td>5</td>
<td>22</td>
<td>778</td>
</tr>
<tr>
<td>Dark Load</td>
<td>5</td>
<td>22</td>
<td>679</td>
</tr>
<tr>
<td>Dark Memory</td>
<td>8</td>
<td>38</td>
<td>870</td>
</tr>
<tr>
<td>Dark Stack</td>
<td>5</td>
<td>22</td>
<td>671</td>
</tr>
<tr>
<td>Total</td>
<td>40</td>
<td>195</td>
<td>4644</td>
</tr>
</tbody>
</table>

Table 5 contains the number of packages, classes, functions and source statements per project, package, class and function.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Packages</th>
<th>Classes</th>
<th>Functions</th>
<th>NCSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project</td>
<td>6.00</td>
<td>40.00</td>
<td>195.00</td>
<td>4,644.00</td>
</tr>
<tr>
<td>Package</td>
<td>6.67</td>
<td>32.50</td>
<td>774.00</td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>4.88</td>
<td></td>
<td>116.10</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td></td>
<td></td>
<td>23.82</td>
<td></td>
</tr>
</tbody>
</table>

Table 6 on the following page contains the number of non-comment source statements and functions per class.

\textsuperscript{10} Please look at \url{http://www.kclee.com/clemens/java/javancss/} for more information about JavaNCSS.
Table 6. Metrics per class

<table>
<thead>
<tr>
<th>Class</th>
<th>NCSS</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dark.About.Win</td>
<td>51</td>
<td>6</td>
</tr>
<tr>
<td>Dark.Change.Value</td>
<td>86</td>
<td>13</td>
</tr>
<tr>
<td>Dark.Debug</td>
<td>504</td>
<td>20</td>
</tr>
<tr>
<td>Dark.Main</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Run</td>
<td>222</td>
<td>6</td>
</tr>
<tr>
<td>Dark.Symbol</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>Dark.SymbolTable</td>
<td>29</td>
<td>6</td>
</tr>
<tr>
<td>Dark.instr</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>Dark.instructions</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>Dark.inte</td>
<td>72</td>
<td>0</td>
</tr>
<tr>
<td>Dark.mem</td>
<td>19</td>
<td>4</td>
</tr>
<tr>
<td>Dark.retst</td>
<td>19</td>
<td>4</td>
</tr>
<tr>
<td>Dark.Accu.Yytoken</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Accu.Utility</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Accu.Yylex</td>
<td>233</td>
<td>16</td>
</tr>
<tr>
<td>Dark.Accu.parser</td>
<td>252</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Accu.sym</td>
<td>39</td>
<td>0</td>
</tr>
<tr>
<td>Dark.Index.Yytoken</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Index.Utility</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Index.Yylex</td>
<td>233</td>
<td>16</td>
</tr>
<tr>
<td>Dark.Index.parser</td>
<td>446</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Index.sym</td>
<td>61</td>
<td>0</td>
</tr>
<tr>
<td>Dark.Load.Yytoken</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Load.Utility</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Load.Yylex</td>
<td>246</td>
<td>16</td>
</tr>
<tr>
<td>Dark.Load.parser</td>
<td>350</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Load.sym</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>Dark.Memory.Main</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Memory.internals</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>Dark.Memory.LexInterface</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Memory.Utility</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Memory.Yylex</td>
<td>247</td>
<td>16</td>
</tr>
<tr>
<td>Dark.Memory.Yytoken</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Memory.parser</td>
<td>512</td>
<td>16</td>
</tr>
<tr>
<td>Dark.Memory.sym</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>Dark.Stack.Yytoken</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Stack.parser</td>
<td>349</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Stack.Utility</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Stack.Yylex</td>
<td>233</td>
<td>16</td>
</tr>
<tr>
<td>Dark.Stack.sym</td>
<td>51</td>
<td>0</td>
</tr>
<tr>
<td>Dark.Stack.sym</td>
<td>51</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 7 contains the number of non-comment source statements and the cyclomatic complexity number (CCN, McCabe metric [6]) per function.

<table>
<thead>
<tr>
<th>Function</th>
<th>NCSS</th>
<th>CCN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dark.AboutWin.AboutWin()</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>Dark.AboutWin.mousePressed(MouseEvent)</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Dark.AboutWin.mouseClicked(MouseEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.AboutWin.mouseReleased(MouseEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.AboutWin.mouseEntered(MouseEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.AboutWin.mouseExited(MouseEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.ChangeValue.ChangeValue(Frame,String,int,int)</td>
<td>48</td>
<td>1</td>
</tr>
<tr>
<td>Dark.ChangeValue.mousePressed(MouseEvent)</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>Dark.ChangeValue.windowClosed(WindowEvent)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.ChangeValue.mouseClicked(MouseEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.ChangeValue.mouseReleased(MouseEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.ChangeValue.mouseEntered(MouseEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.ChangeValue.mouseExited(MouseEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.ChangeValue.windowOpened(WindowEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.ChangeValue.windowClosing(WindowEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.ChangeValue.windowIconified(WindowEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.ChangeValue.windowDeiconified(WindowEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.ChangeValue.windowActivated(WindowEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.ChangeValue.windowDeactivated(WindowEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Debug Debug(boolean)</td>
<td>80</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Debug actionPerformed(ActionEvent)</td>
<td>142</td>
<td>56</td>
</tr>
<tr>
<td>Dark.Debug.mousePressed(MouseEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Debug.mouseReleased(MouseEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Debug.mouseClicked(MouseEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Debug.mouseEntered(MouseEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Debug.mouseExited(MouseEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Debug.windowIconified(WindowEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Debug.windowDeiconified(WindowEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Debug.windowOpened(WindowEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Debug.windowClosed(WindowEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Debug.windowClosing(WindowEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Debug.windowActivated(WindowEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Debug.windowDeactivated(WindowEvent)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Debug.updateLine()</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Debug.updateScreen()</td>
<td>39</td>
<td>17</td>
</tr>
<tr>
<td>Dark.Debug.parseFile()</td>
<td>122</td>
<td>32</td>
</tr>
<tr>
<td>Dark.Debug.clearAll()</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>Dark.Debug.errorPopup(String)</td>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

*continued on next page*
### continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>NCSS</th>
<th>CCN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dark.Debug.errorPopup(String[])</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>Dark.Main.main(String[])</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Run.Run()</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Run.Init()</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Run.getLineNo()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Run.resetBreakFromLineNo(int)</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>Dark.Run.executeOne(boolean)</td>
<td>185</td>
<td>70</td>
</tr>
<tr>
<td>Dark.Run.dump()</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Symbol.Symbol(String,int,int)</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Symbol.dump()</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Dark.SymbolTable.addSymbol(Symbol)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.SymbolTable.getSymbol(String)</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Dark.SymbolTable.getSymbol(int)</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Dark.SymbolTable.isOK()</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Dark.SymbolTable.empty()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.SymbolTable.dump()</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Dark.instr.instr(int,int,int)</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Dark.instr.dump()</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Dark.instructions.add(instr)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Dark.instructions.next()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.instructions.getInstr(int)</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Dark.instructions.empty()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.instructions.dump()</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Dark.mem.setValue(int,int)</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Dark.mem.getValue(int)</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Dark.mem.empty()</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Dark.mem.dump()</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Dark.rest.push(int)</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Dark.rest.pop()</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Dark.rest.empty()</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Dark.rest.dump()</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Accu.Yytoken.Yytoken(int,int,String,int,int)</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Accu.Utility.assert(boolean)</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Accu.Utility.error(int)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Accu.Yylex.Yylex(int)</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Accu.Yylex.yybegin(int)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Accu.Yylex.yy_advance()</td>
<td>29</td>
<td>11</td>
</tr>
<tr>
<td>Dark.Accu.Yylex.yy_move_start()</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Accu.Yylex.yy_pushback()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Accu.Yylex.yy_mark_start()</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

### continued on next page
Function | NCSS | CCN
--- | --- | ---
Dark.Acu.Yylex.yy_mark_end() | 2 | 1
Dark.Acu.Yylex.yy_openmark() | 2 | 1
Dark.Acu.Yylex.yytext() | 2 | 1
Dark.Acu.Yylex.yyleNGTH() | 2 | 1
Dark.Acu.Yylex.yy_double(char[]) | 7 | 2
Dark.Acu.Yylex.yy_error(int,boolean) | 5 | 3
Dark.Acu.Yylex.unpackFrom(String,int,String) | 31 | 6
Dark.Acu.Yylex.yyleX() | 89 | 36
Dark.Acu.parser.parser(Debug) | 41 | 5
Dark.Acu.parser.parse() | 203 | 67
Dark.Acu.parser.parse_error(String) | 2 | 1
Dark.Index.Yytokem.Yytokem(int,int,String,int,int) | 7 | 1
Dark.Index.Utility.assert(boolean) | 3 | 3
Dark.Index.Utility.error(int) | 2 | 1
Dark.Index.Yylex.Yylex(java.io.Reader) | 5 | 3
Dark.Index.Yylex.Yylex(java.io.InputStream) | 5 | 3
Dark.Index.Yylex.YyleX() | 9 | 1
Dark.Index.Yylex.yybegin(int) | 2 | 1
Dark.Index.Yylex.yy_advance() | 29 | 11
Dark.Index.Yylex.yy_moveStart() | 5 | 2
Dark.Index.Yylex.yy_pushback() | 2 | 1
Dark.Index.Yylex.yy_markStart() | 7 | 3
Dark.Index.Yylex.yy_markEnd() | 2 | 1
Dark.Index.Yylex.yy_closemark() | 2 | 1
Dark.Index.Yylex.yytext() | 2 | 1
Dark.Index.Yylex.yyleNGTH() | 2 | 1
Dark.Index.Yylex.yy_double(char[]) | 7 | 2
Dark.Index.Yylex.yy_error(int,boolean) | 5 | 3
Dark.Index.Yylex.unpackFrom(String,int,String) | 31 | 6
Dark.Index.Yylex.yyleX() | 89 | 36
Dark.Index.parser.parser(Debug) | 63 | 5
Dark.Index.parser.parse() | 371 | 115
Dark.Index.parser.parse_error(String) | 2 | 1
Dark.Load.Yytokem.Yytokem(int,int,String,int,int) | 7 | 1
Dark.Load.Utility.assert(boolean) | 3 | 3
Dark.Load.Utility.error(int) | 2 | 1
Dark.Load.Yylex.YyleX() | 9 | 1
Dark.Load.Yylex.yybegin(int) | 2 | 1
Dark.Load.Yylex.yy_advance() | 29 | 11
Dark.Load.Yylex.yy_moveStart() | 5 | 2

*continued on next page*
### continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>NCSS</th>
<th>CCN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dark.Load.Yylex.yy_pushback()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Load.Yylex.yy_mark_start()</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Load.Yylex.yy_mark_end()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Load.Yylex.yy_mark()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Load.Yylex.yy_text()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Load.Yylex.yy_length()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Load.Yylex.yy_error(int,boolean)</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Load.Yylex.unpackFromString(int,int,String)</td>
<td>31</td>
<td>6</td>
</tr>
<tr>
<td>Dark.Load.Yylex.yylex()</td>
<td>101</td>
<td>43</td>
</tr>
<tr>
<td>Dark.Load.parser.parser(Debug)</td>
<td>46</td>
<td>5</td>
</tr>
<tr>
<td>Dark.Load.parser.parse()</td>
<td>296</td>
<td>101</td>
</tr>
<tr>
<td>Dark.Load.parser.parse_error(String)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Memory.Main.main(String[])</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>Dark.Memory.Utiliy.assert(boolean)</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Memory.Utiliy.error(int)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Memory.Yytokon.Yytokon(int,String,int,int,int)</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Memory.Lexlnterface.init(java.io.InputStream)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Memory.LexInteace.nextToken()</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.Yylex(java.io.Reader)</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.Yylex(java.io.InputStream)</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.Yylex()</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.yybegin(int)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.yyAdvance()</td>
<td>29</td>
<td>11</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.yy_move_start()</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.yy_pushback()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.yy_mark_start()</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.yy_mark_end()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.yy_mark()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.yy_text()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.yy_length()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.yy_double(char[])</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.yy_error(int,boolean)</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.unpackFromString(int,int,String)</td>
<td>31</td>
<td>6</td>
</tr>
<tr>
<td>Dark.Memory.Yylex.yylex()</td>
<td>103</td>
<td>47</td>
</tr>
<tr>
<td>Dark.Memory.parser.parser(Debug)</td>
<td>43</td>
<td>5</td>
</tr>
<tr>
<td>Dark.Memory.parser.parseSimpleAddress(Yytokon)</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Memory.parser.getImmediate(Yytokon)</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Dark.Memory.parser.parseBaseAddress(Yytokon)</td>
<td>53</td>
<td>16</td>
</tr>
<tr>
<td>Dark.Memory.parser.parseAddress()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Memory.parser.getComma()</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Memory.parser.getCommand2Args(Symbol)</td>
<td>11</td>
<td>7</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Function</th>
<th>NCSS</th>
<th>CCN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dark.Memory.parser.getCommand3Args(Symbol)</td>
<td>27</td>
<td>23</td>
</tr>
<tr>
<td>Dark.Memory.parser.parseCommand(Yytoken)</td>
<td>94</td>
<td>21</td>
</tr>
<tr>
<td>Dark.Memory.parser.jump1Arg(Symbol)</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Dark.Memory.parser.jump3Args(Symbol)</td>
<td>17</td>
<td>13</td>
</tr>
<tr>
<td>Dark.Memory.parser.parseJump(Yytoken)</td>
<td>73</td>
<td>17</td>
</tr>
<tr>
<td>Dark.Memory.parser.parseEnd()</td>
<td>30</td>
<td>12</td>
</tr>
<tr>
<td>Dark.Memory.parser.parseNextLine(Yytoken)</td>
<td>107</td>
<td>51</td>
</tr>
<tr>
<td>Dark.Memory.parser.parse()</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Dark.Memory.parser.parseError(String)</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Stack.Yytoken.Yytoken(int,int,String,int,int,)</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Stack.parser.parser(Debug)</td>
<td>50</td>
<td>5</td>
</tr>
<tr>
<td>Dark.Stack.parser.parse()</td>
<td>291</td>
<td>80</td>
</tr>
<tr>
<td>Dark.Stack.parser.parseError(String)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Stack.Utility.assert(boolean)</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Stack.Utility.error(int)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Stack.Yylex.Yylex()</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Stack.Yylex.yybegin(int)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Stack.Yylex.yyadvance()</td>
<td>29</td>
<td>11</td>
</tr>
<tr>
<td>Dark.Stack.Yylex.yymove_start()</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Stack.Yylex.yypushback()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Stack.Yylex.yymark_start()</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Stack.Yylex.yymark_end()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Stack.Yylex.yy_to_mark()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Stack.Yylex.yytext()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Stack.Yylex.yylength()</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Dark.Stack.Yylex.yy_double(char[])</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Dark.Stack.Yylex.yyerror(int,boolean)</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Dark.Stack.Yylex.unpackFromString(int,int,String)</td>
<td>31</td>
<td>6</td>
</tr>
<tr>
<td>Dark.Stack.Yylex.yylex()</td>
<td>89</td>
<td>36</td>
</tr>
</tbody>
</table>
C Implementation Execution Time Metrics

This section contains execution times for assembling and executing Ackermans function with two different sets of parameters, a) (2, 2) and b) (3, 3). The test stack assembler code required 322 symbolic and 1217 internal steps for the first test case and 29133 symbolic and 112705 internal steps for the latter.

All tests have been performed on a Sun Ultra-10 with 248 MB of main memory and a 333 MHz SUNW UltraSPARC-IIi processor running SunOS Release 5.6.

C.1 Terminal-Based Version

The table below gives user and system time in seconds (with standard deviations) used by the C program (compiled with gcc version 2.95.2, no optimizations) for each stage.

<table>
<thead>
<tr>
<th>Step</th>
<th>user time (s)</th>
<th>σ</th>
<th>system time (s)</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>0.002</td>
<td>4.216</td>
<td>0.008</td>
<td>4.216</td>
</tr>
<tr>
<td>Execute a</td>
<td>0.626</td>
<td>3.975</td>
<td>0.050</td>
<td>1.871</td>
</tr>
<tr>
<td>Execute b</td>
<td>55.980</td>
<td>1.112</td>
<td>3.576</td>
<td>0.194</td>
</tr>
</tbody>
</table>

C.2 Java Version

The table below gives user and system time in seconds (with standard deviations) used by Suns JDK 1.1.6 version of Java for each stage. The Assembly stage times include the start-up stage times and the execution times include both start-up and assembly times.

<table>
<thead>
<tr>
<th>Step</th>
<th>user time (s)</th>
<th>σ</th>
<th>system time (s)</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start-up</td>
<td>0.470</td>
<td>4.743</td>
<td>0.164</td>
<td>0.037</td>
</tr>
<tr>
<td>Assembly</td>
<td>0.630</td>
<td>2.915</td>
<td>0.174</td>
<td>3.362</td>
</tr>
<tr>
<td>Execute a</td>
<td>2.968</td>
<td>6.181</td>
<td>0.272</td>
<td>4.087</td>
</tr>
<tr>
<td>Execute b</td>
<td>557.422</td>
<td>3.797</td>
<td>16.562</td>
<td>0.468</td>
</tr>
</tbody>
</table>
C.3 Interpretation

There are a few reasons for the higher throughput of the C version. C is many times faster than Java, since Java is interpreted rather than compiled. The graphical user interface of the Java version requires more CPU power than the curses-based interface in C. Moreover, the terminal-based version shows less information than the Java version, e.g. only the top five values on the stack compared to the entire stack.
References

Index

const, 30
curses, 7
data, 18, 25, 35
DEC, 3
dec, 26
decy, 24
Digital Equipment, 3
div, 19, 26, 32, 36
drop, 36
dup, 36
div, 19, 26, 32, 36
eq, 36
equ, 30
Forth, 3
ge, 37
gt, 37
IBM, 1
inc, 27
incx, 24
incy, 24
Irix, 1, 11
.iix, 7, 8, 42
Java, 2, 6, 8, 9, 11, 12, 14, 15, 45–53
JavaNCSS, 45
jeq, 19, 27
.jfalse, 37
.jge, 19, 27
.jgt, 19, 27
.jle, 20, 27
.JLex, 8, 9, 45
.jlt, 20, 27
.jmp, 20, 27, 37
.jne, 20, 27
.jtrue, 37
label, 17, 30
le, 37
lex, 7, 8

() , 30
( , 30
.ac, 7, 8, 41
.iix, 7, 8, 42
.is, 7, 8, 43
.mm, 7, 8, 43
.st, 7, 8, 44
;, 17
$, 25

6502, 3, 6
6800, 3
68000, 3
.ac, 7, 8, 41
addl, 18, 25, 30, 35
AIX, 1
andl, 18, 25, 31, 35
arch, 7
area, 35
assembler
– accumulator, 3, 4, 18
– index, 3, 6, 22
– load-store, 3, 4, 25
– memory-memory, 3, 5, 30
– stack, 3, 5, 35
– virtual, 1–3, 7–9, 11, 17

band, 18, 26, 31, 35
beq, 31
bge, 31
bgt, 31
ble, 31
blt, 31
bne, 32
bnot, 18, 26, 32, 35
bor, 18, 26, 32, 35
bra, 32
breakpoint, 2, 12, 14
bxor, 18, 26, 32, 36

C, 7, 8
call, 19, 26, 32, 36
CCN, 47–51
cmp, 19
Compaq, 3
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>load, 20, 28</td>
<td>ret, 21, 28, 33, 39</td>
</tr>
<tr>
<td>loadax, 22</td>
<td>rot, 39</td>
</tr>
<tr>
<td>loaday, 22</td>
<td>SGI, 1, 11</td>
</tr>
<tr>
<td>loadix, 22</td>
<td>sp, 25</td>
</tr>
<tr>
<td>loadiy, 22</td>
<td>.st, 7, 8, 44</td>
</tr>
<tr>
<td>loadixy, 22</td>
<td>stack, 30</td>
</tr>
<tr>
<td>loadxy, 22</td>
<td>stop, 21, 29, 33, 39</td>
</tr>
<tr>
<td>loadx, 22</td>
<td>store, 21, 29</td>
</tr>
<tr>
<td>loady, 22</td>
<td>storeix, 23</td>
</tr>
<tr>
<td>.ls, 7, 8, 43</td>
<td>storeixy, 23</td>
</tr>
<tr>
<td>It, 38</td>
<td>storeiy, 23</td>
</tr>
<tr>
<td>machine</td>
<td>storex, 23</td>
</tr>
<tr>
<td>- accumulator, 3, 4, 18, 41</td>
<td>storey, 23</td>
</tr>
<tr>
<td>- index, 3, 6, 22, 42</td>
<td>sub, 21, 29, 34, 40</td>
</tr>
<tr>
<td>- load-store, 3, 4, 25, 43</td>
<td>Unix, 1, 6, 7</td>
</tr>
<tr>
<td>- memory-memory, 3, 5, 30, 43</td>
<td>VAX, 3</td>
</tr>
<tr>
<td>- stack, 3, 5, 35, 44</td>
<td>Verilog, 2</td>
</tr>
<tr>
<td>- virtual, 1-3, 6-9, 11-15</td>
<td>VHDL, 2</td>
</tr>
<tr>
<td>McCabe, 47</td>
<td>virtual assembler, 1-3, 7-9, 11, 17</td>
</tr>
<tr>
<td>mem, 30</td>
<td>virtual machine, 1-3, 6-9, 11-15</td>
</tr>
<tr>
<td>Microsoft, 2</td>
<td>Windows NT, 2</td>
</tr>
<tr>
<td>MIPS, 1, 3</td>
<td>X11, 8</td>
</tr>
<tr>
<td>.mm, 7, 8, 43</td>
<td>xor, 21, 29, 34, 40</td>
</tr>
<tr>
<td>mod, 20, 28, 33, 38</td>
<td>yacc, 7, 9</td>
</tr>
<tr>
<td>Motorola, 3</td>
<td>zero, 25</td>
</tr>
<tr>
<td>mov, 28, 33</td>
<td></td>
</tr>
<tr>
<td>mul, 20, 28, 33, 38</td>
<td></td>
</tr>
<tr>
<td>mv, 28, 33</td>
<td></td>
</tr>
<tr>
<td>NCSS, 45-51</td>
<td></td>
</tr>
<tr>
<td>ne, 38</td>
<td></td>
</tr>
<tr>
<td>not, 20, 28, 33, 38</td>
<td></td>
</tr>
<tr>
<td>or, 21, 28, 33, 38</td>
<td></td>
</tr>
</tbody>
</table>
The DARK-Series of Virtual Machines

Technical Report
UMINF 00.15
ISSN 0348-0542

Ola Agren

Department of Computing Science, Umeå University, Sweden