User-Guide
Projekt „Hypothetical Machinecode - Assembler and Simulator“

Projektgruppe: *Markus Hennecke*
Phasenverantwortlich: *Markus Hennecke*

14th September 2003

Contents

1 General Description

1.1 Introduction

1.2 Program Parts

2 The Simulated Hardware

2.1 Register and Wordwidth

The simulated Processor has got 16 bit Registers. There are five Registers: The accumulator, the helpregister, the indexregister, the instruction pointer (short IP) and the stack pointer (SP). Only

the accumulator can be used for arithmetic operations. The width of the addresses used is 16 bit too.

2.2 Memory Organization

Every memory cell holds a 16 bit word. This is different to common systems like personal computers. There are 64k words addressable making it 128kb. ASCII values are stored as 16 bit values as well, so no 8 bit access methods are known to the processor.

2.3 Input / Output

I/O is memory mapped. Reading or writing to a address reserved to I/O operations results in accessing the device attached to that address. See section 6.3 for further information.

3 Description of the Hypothetical Machinecode

3.1 Instruction Structure

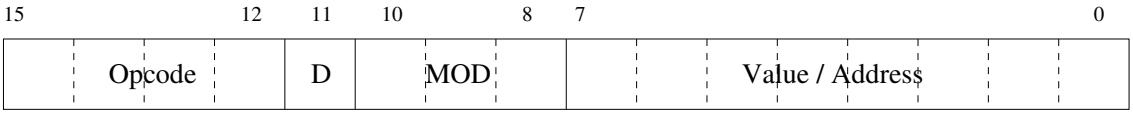

Figure 1: Structure of Instructions

OP-Code The code of an operation. See section 3.2.

D Mark for double-word instructions. The data is in the next word if this bit is set. An Exception are the operations on the stack where this bit is a part of the modification part.

MOD Modification part. With these bits addressing modes, stack- and indexoperations are selected.

Address / Value In this part the data belonging to the operation is stored.

3.2 Instructions

dual	Description of operation	Mnemonic
0000	Halt, end of program	HLT
0001	Load accu	LAD
0010	Store accu to address	SPI
0011	Add operand to accu	ADD
0100	Subtract operand from accu	SUB
0101	Multiply accu with operand	MUL
0110	Divide accu with operand	DIV
0111	Unconditional jump	JMP
1000	Jump if accu is equal zero	JEZ
1001	Jump if accu is greater or equal zero	JGZ
1010	Jump if accu is lower zero	JLZ
1011	Swap accu and help register	SWAP
1100	Stackinstructions, see section 3.2.1	*
1101	Jump to subroutine	JSR
1110	Return from subroutine	RET
1111	Indexinstruction, see section Indexinstructions	*

HLT Ends the program execution. Register and memory values are not changed. After setting the instruction pointer to another address the programm execution can be resumed.

LAD Loads the accumulator with the operand or the value described by the addressing mode.

SPI Stores the accumulator to the memory address described by the addressing mode.

ADD Adds the operand described by the addressing mode to the accumulator. This operation obeys signs. There is no information if an overflow occured.

SUB Subtracts the operand described by the addressing mode from the accumulator. This operation obeys signs. There is no information if an overflow occured.

MUL Multiplies the operand with the accumulator and stores the result in the accu. This operation obeys signs. There is no information if an overflow occured.

DIV Divides the accumulator with the operand and stores the result in the accu. This is an integer division and the remainder is stored in the SWAP Register. This operation obeys signs. There is no information if an overflow occured.

JMP Jumps to the given address.

JEZ Jumps to the given address if the accumulator is equal zero.

JGZ Jumps to the given address if the accumulator is greater or equal zero.

JLZ Jumps to the given address if the accumulator is lower zero.

SWAP Swaps the accumulator and the help register.

JSR Jumps to the given address. The address of the following instruction is pushed on the stack.

RET Pops an address from the stack and continues the program execution at that address.

The arithmetic instructions always hold their result in the accumulator.

3.2.1 Stackinstructions

Following is a list of stack operations. These instructions are differentiated by the double-word- and the MOD-bits. Those operations not defined by these bits are illegal instructions. These instructions can be used for further extensions to the instruction set. Because of the double-word bit used to differentiat instructions there are no short versions of the stack instructions.

D+MOD	Description	Mnemonic
0001	Push the accumulator on the stack	PUSHAC
0010	Pop a value from the stack and store it in the accumulator	POPAC
0011	Add the two values pushed last on the stack	ADDST
0100	Subtract the two values pushed last on the stack	SUBST
0101	Multiply the two values pushed last on the stack	MULST
0110	Divide the two values pushed last on the stack	DIVST
0111	Store accumulator to the stack pointer	LSA
1000	Load accumulator with the address the stack pointer holds	LDS
1011	Push operand on the stack	PUSH
1100	Pop operand from stack	POP

Arithmetic operations on the stack Arithemtic operations using the stack for parameters are executed in the following order:

1. The last value pushed on the stack is the first operand.
2. The first value pushed on the stack is the second operand.
3. The operation is executed and the parameters are removed from the stack.
4. The result is pushed on the stack.

PUSHAC The accumulator is pushed on the stack. The stack pointer is decreased after the value is stored.

POPAC The stack pointer is increased by one and the value at that address is stored to the accumulator.

ADDST Adds the two last pushed elements on the stack. See the description of stack arithmetic

SUBST Subtracts the two last pushed elements on the stack. See the description of stack arithmetic

MULST Multiplies the two last pushed elements on the stack. See the description of stack arithmetic

DIVST Divides the two last pushed elements on the stack. See the description of stack arithmetic

LSA Transfers the value stored in the accumulator to the stack pointer.

LDS Transfers the address stored in the stack pointer to the accumulator.

PUSH Pushs a 16 bit value on the stack. Only direct addressing mode is possible with this instruction.

POP Pops the top value from the stack and stores it to the specified address. Only direct addressing mode is supported.

3.2.2 Indexinstructions

The following index instructions have the opcode 1111 and are selected via the MOD part. They have no parameters with exception of the LDX instruction because all operate the index register direct.

MOD	Short Description	Mnemonic
000	Load zero to the index register	CLI
001	Transfer the accumulator to the index register	LDI
010	Transfer the index register to the accumulator	LIA
011	Increment the index register by one	INK
100	Decrement the index register by one	DEK
101	Illegal Opcode	*
110	Illegal Opcode	*
111	Load value from an absolute address into the index register	LDX,@adresse

CLI Load zero to the index register.

LDI Transfer the accumulator to the index register.

LIA Transfer the index register to the accumulator.

INK Increments the index register by one.

DEK Decrements the index register by one.

LDX Load value from an absolute address into the index register. Only the direct addressing mode is possible. The instruction is always a double-word instruction.

3.2.3 Addressing Modes

Direct via Address Form: @address, MOD: 0xx.
The first bit of the MOD bits is zero. The remaining bits are the highest bits of the address. A single-word instruction has a addressrange of 10 bits. In a double-word instruction the remaining MOD bits are ignored.

Direct with Operator Form: #operator, MOD: 100.
Values from -128 to +127 are stored in a single-word instruction. For higher values a double-word instruction is used. Those bits not belonging to the maximum addressing width of the cpu are ignored.

Relative to the Instruction Pointer Form: address, MOD: 101
The address is relative to the instruction pointer. There are 8 bits usable for single-word instructions (-128 to 127). For greater offsets a double-word instruction will be used. Die Adresse wird relativ zum Those bits not belonging to the maximum addressing width of the cpu are ignored.

Indirect Address Form: (address), MOD: 110.
The real address is found via the address given as a parameter. There are 8 bits usable for single-word instructions (-128 to 127). For greater offsets a double-word instruction will be used. Die Adresse wird relativ zum Those bits not belonging to the maximum addressing width of the cpu are ignored.

Relative to the Indexregister Form: %address, MOD: 111.
The address is relative to the indexregister. There are 8 bits usable for single-word instructions (-128 to 127). For greater offsets a double-word instruction will be used. Die Adresse wird relativ zum Those bits not belonging to the maximum addressing width of the cpu are ignored.

3.2.4 Instructiontable

Mnemonic	Dir. Op.	Dir. Addr.	Relative	Indirect	Index	Mnemonic	Dir. Op.	Dir. Adr.	Relativ	Indirekt	Index
HLT	-	-	-	-	-	CLI	-	-	-	-	-
LAD	X	X	X	X	X	LDI	-	-	-	-	-
SPI	-	X	X	X	X	LIA	-	-	-	-	-
ADD	X	X	X	X	X	INK	-	-	-	-	-
SUB	X	X	X	X	X	DEK	-	-	-	-	-
MUL	X	X	X	X	X	LDX	-	X	-	-	-
DIV	X	X	X	X	X	PUSHAC	-	-	-	-	-
JMP	-	X	X	X	X	POPAC	-	-	-	-	-
JEZ	-	X	X	X	X	ADDST	-	-	-	-	-
JGZ	-	X	X	X	X	SUBST	-	-	-	-	-
JLZ	-	X	X	X	X	MULST	-	-	-	-	-
JSR	-	X	X	X	X	DIVST	-	-	-	-	-
RET	-	-	-	-	-	PUSH	X	-	-	-	-
LSA	-	-	-	-	-	POP	-	X	-	-	-
LDS	-	-	-	-	-	SWAP	-	-	-	-	-

4 The Assembler “as”

4.1 Command Line Parameter

Commandline options for the assembler “as”:
`as [options] sourcefile`

-h Shows a list of available options with a short description.

-l Enables listfile generation. This is a general switch. If the NOLIST meta instruction is used the listfile generation will be disabled. By default this filename of the listfile is derived from the source filename by substituting a “.asm” postfix with “.lst”. If “.asm” is not the fileending “.lst” is appended to the source filename.

-nl Disables listfile generation. Using this switch every occurence of LIST in the source is ignored.

-m number Sets the maximum passes done by the assembler. The default value is set to 99, possible values are 2 to 99.

-o filename Sets the filename of the objectfile. By default this filename is derived from the source filename by substituting a “.asm” postfix with “.obj”. If “.asm” is not the fileending “.obj” is appended to the source filename.

Parts in brackets are optional.

4.2 Input Format

The assembler reads lines with the following structure:
`[Label:]  [Mnemonic [, Argument]] [; Comment]`

Label A label starts with a character between “a” and “z”. The following characters can be either lower- or uppercase or digits. The colon (“:”) is strictly required after the label.

Mnemonic A mnemonic is either a valid instruction or meta instruction. The characters in this string have to be uppercase.

Argument The argument is separated from the mnemonic by a comma (“,”). See Addressing Modes for the format. As parameters are allowed symbolic names (e.g. labels) or numbers. Numbers are decimal only.

Comment The comment is started with a semicolon (“;”). Everything after this mark is ignored.

Every part in brackets is optional input.

4.3 META Instructions

The following instructions are defined to assign values to labels and to control listfile generation:

ORG Sets the instruction pointer to the absolute memory address. Following instructions are assembles relative to this address.

LIST From this line on all instructions are written to the listfile. See Listfile Format for more information.

NOLIST The following lines produce no output in the listfile.

PAGE Is listfile enabled a new page is started. (This instruction is without function as there are too many printers around)

EQUIVALENT Short: **EQU**. Assigns a value to the label. Format: label EQU value

WORD Writes the following value to memory. Only single values are supported.

TEXT Writes the string as ASCII to memory. See memory organization for more information.

4.4 Listfile Format

Output to the listfile is printed in the following format:
`Linenumber Address Bytecode Sourceline` The bytecode is broken in 16 bit blocks, each block written in 4 hex digits. If a line produces more than 2 blocks of bytecode only the first two blocks are displayed. This is the case for the TEXT metainstruction.

4.5 Objectfile Format

The objectfile is written with the following scheme: `Address: Bytecode ; Comment` The comment is a simplified readable description of the source line and optional. The bytecode is broken in 16 bit blocks, each block written in 4 hex digits. There is no restriction in the number of blocks containing the bytecode.

5 The GUI

5.1 Overview

The GUI consists of two parts. First the Editor which is invoked at program start. Second the Debugger which can be invoked from the Editor or after the successful assembling of a sourcefile.

5.2 Command Line Parameter

The Program itself does not accept any command line parameters. In fact these are ignored. The QT toolkit does accept command line parameters. See the QT documentation for further information

5.3 Menu Entries

5.3.1 File Menu

New If in edit mode creates a new empty document. This menu entry is not available in debug mode.

Open if in edit mode opens an existing document. In debug mode it opens an existing object file.

Save Only available in edit mode. Saves the current document. If the document is not associated to a filename the user is asked for a filename.

Save As Only available in edit mode. Saves the current document, the user is asked for a filename.

Quit Exits the program.

5.3.2 Edit Menu

Only available in edit mode.

Undo Undos the last opereration

Redo Redos the last undone operation.

Cut Cuts the selected text.

Copy Copies the selected text.

Paste Pastes a previously cutted or copied text.

Find Opens a dialogbox where the text to find can be typed in.

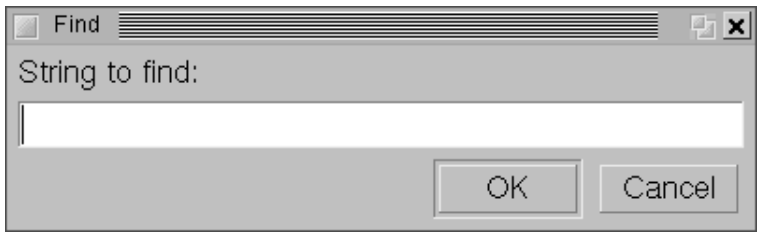

Figure 2: *Find Dialogbox*

5.3.3 Assembler Menu

Only available in edit mode.

Switch to Debugger Switches to the debugger.

Assemble Assembles the current file. If it is not saved it is saved now. A dialogbox with the results of the assemble is displayed and if no errors were found the user is asked to switch to the debugger.

Options Opens a dialogbox with the available options of the program.

Path to the as binary Sets the path to the assembler. The Filebrowser button can be used to choose it with a file dialog.

Generate Listfile If checked listfile generation is allowed.

Max. Number of Passes Sets the maximal number of passes.

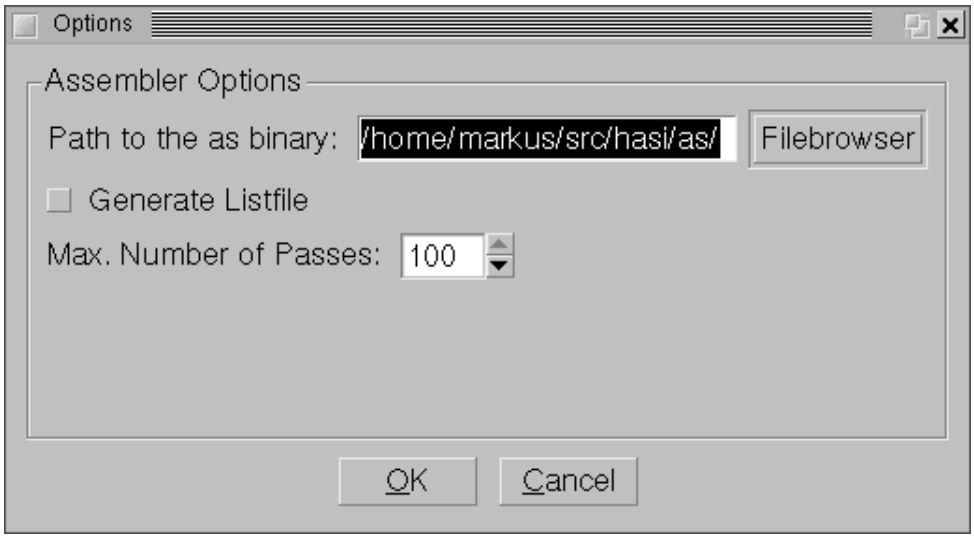

Figure 3: *Options*

5.3.4 Help Menu

Contents Opens the helpwindow.

Index Opens the helpwindow. Starting at the indexpage.

About Shows an about dialogbox.

5.4 Edit Mode

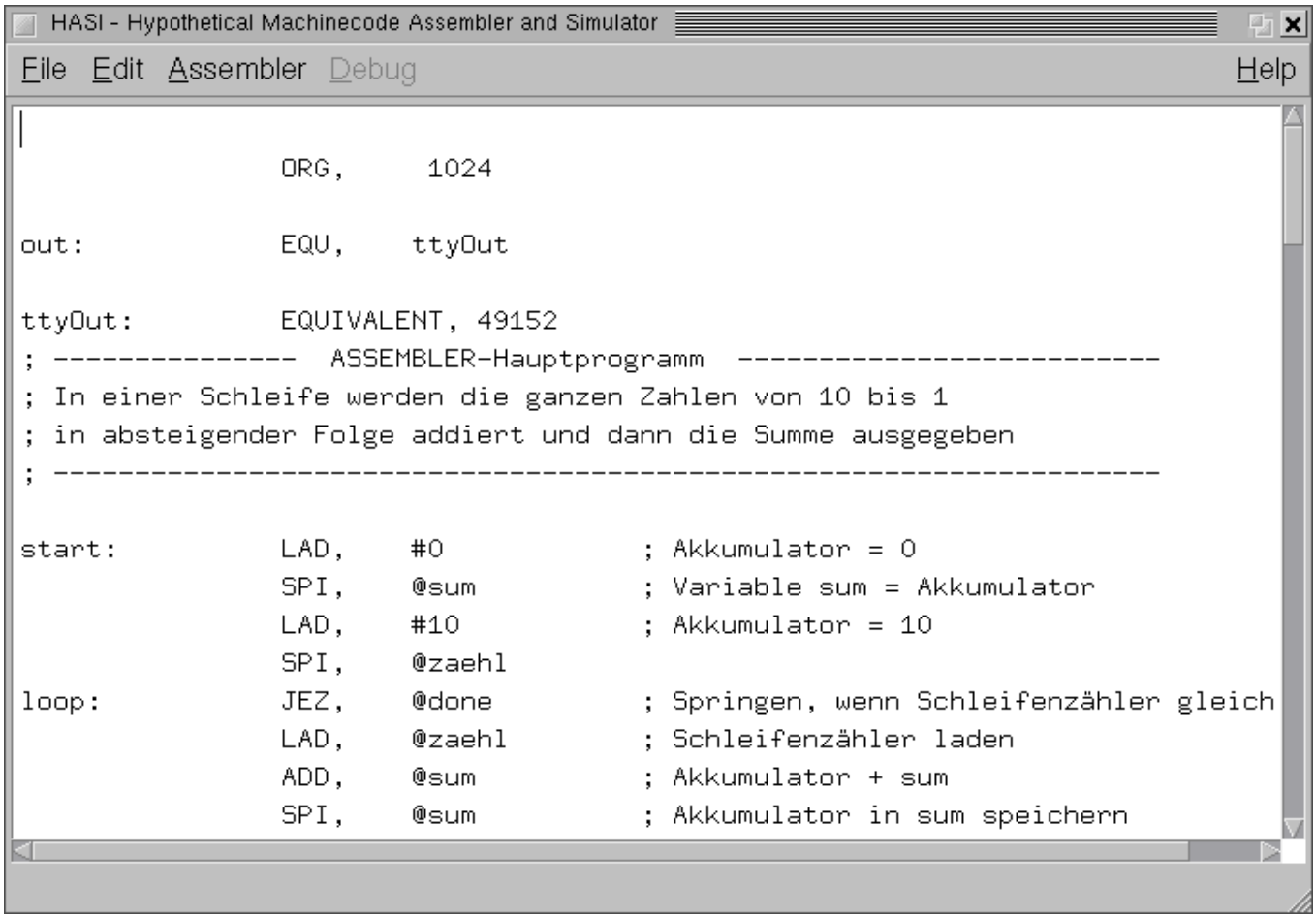

Figure 4: *The Editor*

5.5 Debug Mode

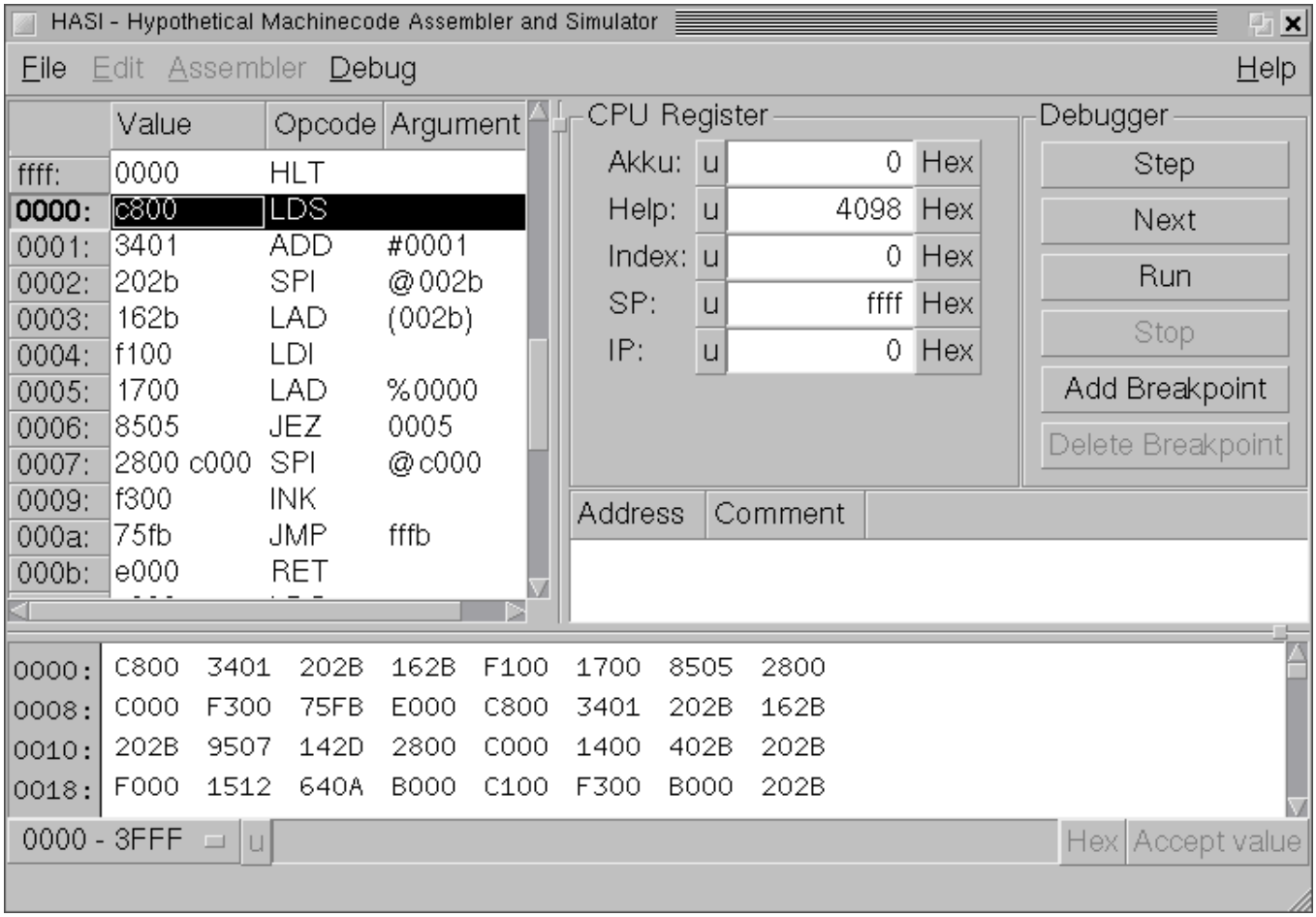

Figure 5: *The Debugger*

In debug mode there are several elements on the screen.

In the upper left corner is a listing with disassembled instructions around the instruction pointer.

To the right of this the CPU registers are shown. These can be switched from signed to unsigned and from hex to decimal display. The values are editable, the edited value is taken after the user pressed return.

Far to the right are the buttons located to control the debugger. These are similar to the menu entries of the Debug menu.

Step Steps the next instruction.

Next Steps over the next instruction.

Run Runs the program from the current position.

Stop Stops the execution of a running program.

Add Breakpoint Adds a breakpoint. The Breakpoint is shown in the list below these buttons. The user is able to add a comment to each added breakpoint.

Delete Breakpoint Removes a selected breakpoint from the list.

To the lower left is a listing with a representation of the current memory. The memory is displayed in maximal 16k, dividing the available memory into 4 equal sized parts. If a memory cell is selected it can be edited similar to the CPU Registers.

In the lower Right is a representation of the memory located above the current stack pointer. The last 20 positions are shown in this view. This is read only.

6 The Interpreter

The interpreter is a standalone program with no gui.

6.1 Command Line Parameter

Commandline options for the interpreter "hasi-nogui":
`hasi-nogui [options] objectfile`

-h Shows a list of available options with a short description.

-d Adds debug output to the console by printing out the registers after each instruction.

-s number Sets the start address. This defaults to 0.

Parts in brackets are optional. These parameters are not handled in the gui version.

6.2 Objectfile Format

See Objectfile Format in the description of the assembler "as".

6.3 Builtin I/O

6.3.1 Example - Output to the console

An example class is included into the project. This class handles output to the console by printing out everything written to address 49152. The declaration can be used for further extensions:

```
#include "Memory.hpp"

namespace Interpreter {

  class IOTtyOut : public Memory
  {
  public:
    // Con and Destructor:
    IOTtyOut(const unsigned int start = 0);
    IOTtyOut(const IOTtyOut &);
    virtual ~IOTtyOut();

    // The read and write functions reimplemented for IOTtyOut
    virtual bool write(const unsigned int addr, const short val);
    virtual bool read(const unsigned int addr, short &) const;
  };
}
```

The implementation is very simple:

```
#include "IOTtyOut.hpp"

namespace Interpreter {

  // IOTtyOut::IOTtyOut(int)
  // Initializes the IOTtyOut. The content of the IOTtyOut is undefined.
  IOTtyOut::IOTtyOut(const unsigned int start)
    : Memory(1, start)
  { }

  // IOTtyOut::IOTtyOut(const IOTtyOut &)
  // Copies the memory. The content is copied 1:1. If there was no memory
  // in the argument new memory will be created.
  IOTtyOut::IOTtyOut(const IOTtyOut &m)
    : Memory(1, m.start)
  { }

  // IOTtyOut::~IOTtyOut()
  // Frees the IOTtyOut.
  IOTtyOut::~IOTtyOut()
  { }

  // bool IOTtyOut::write(const unsigned int, const short)
  // Writes a ASCII char to a tty. This is only done if the address is equal
  // to start. True or false will be returned according to the success of
  // the operation.
  bool IOTtyOut::write(const unsigned int addr, const short val)
  {
    if (addr == start) {
      std::cout << static_cast<char>((val & 0xff));
      return true;
    }
    return false;
  }

  // bool IOTtyOut::read(const unsigned int, short &)
  // False will be returned and the value is set to -1 (0xffff).
  bool IOTtyOut::read(const unsigned int addr, short &val) const
  {
    val = -1; // don't produce warnings on unused vars...
    return false;
  }
}
```

Such a simple class is really easy to implement.