VAX11 Assembler

Adar Nir
Grosman Rotem

1. Contents

2. Introduction

Nir Adar
Email: underwar@hotmail.com
Home Page: http://underwar.livedns.co.il

Rotem Grosman
Email: superrpg@t2.technion.ac.il

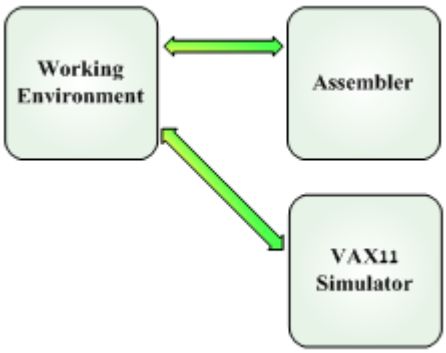
The following document describes the algorithms we are using for our project:
Writing an assembler & simulator to VAX11 computer.
This document contains only the algorithms we used to write the assembler.
We implemented the algorithms using C# as Windows application.

We would to thank our supervisor, **Gal Zion**, and also **Dr. Ilana David** from the Technion, for helping us with our project.

3. General

1. Project Modules

The project is divided to three main modules: Working Environment, Assembler, and simulator.
The working environment contains a source editor, and it uses the other modules to provide complete IDE for writing, compiling and testing programs for the VAX11.
The assembler and the simulator modules provide services to the environment.



2. Goals and Times

The project is divided to two parts: The first part is writing the environment and the assembler, and the second is writing the simulator.
The time for completing each part of the project is four months.

First part's Goals:

- Write the Environment
- Write the Assembler
- Generate List File in order to test the assembler.

Second part's Goals:

- Write the Simulator
- Debug…

3. The old simulator

The Technion is currently using VAX11 simulator written on 1989. It is a DOS program with many limitations.

Problems with the Old Simulator

- Dos Program
- Unfriendly Interface
- Can Handle only one file at time
- Memory Limitation
- BUGS...

Advantages of Our Simulator

- Windows Application
- Familiar Interface (Menus, Toolbars, Dialog Forms, Shortcuts, Environment Looks Like Visual Studio .Net)
- Can Handle Multiple Files at a Time
- No Memory Limitations

4. Settings Class

1. Description

Setting class is helping class that contains all the setting of VAX11 simulator, assembler and working environment.
When the environment is loaded, all the settings are read from the registry to this class, for use of all the other modules in the project.
The class has several sub classes that indeed to organize the settings.

2. Class Structure

Settings class contains three sub classes; each contains setting and constants that relevant to other module in the project:

- Assembler
- Simulator
- Environment

Assembler Settings:

Setting	Values	Meaning
GenerateLstFile	Yes/No	Should the assembler generate LST files while compiling the code?
LstFileAppend	Over/Append	When generating LST file, if there is already file with the same name, should we override it or append the new LST file to it?

Simulator Settings:

Setting	Values	Meaning
StartAddress	int	Address where to load the user's code

Environment Settings:

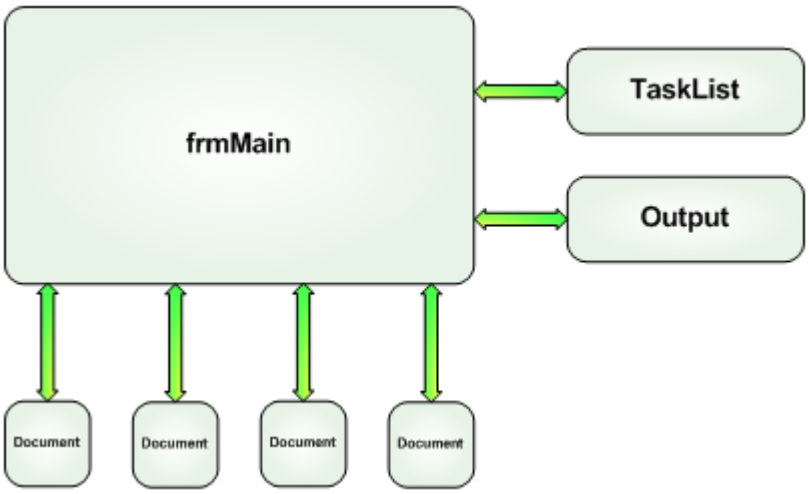
Setting	Values	Meaning
HighlightCode	Yes/No	Should the environment highlight the user's code?
EditorFont	Font	Font size and type that will be used by the environment

Environment Constants

Constant	Value	Meaning
FILE_FILTERS	string	Filters for file saving / loading
MESSAGEBOXS_TITLE	string	Define the title of the different message boxes of the environment

5. Working Environment

1. Overview



frmMain

- The main object. Application starts from this class.
- Contains the status bar, the menus, the toolbar, and the container.
- Displays the entire user interface.

TaskList

- Contains warnings and errors massages generated during compile.

Output

- Contains the compile massages.
- Displays messages during the compile process.

Documents

- Contains the source files of the user.
- Communicate with the main application via events.
- The application might contain multiply instances of this class at the same time.

2. Environment Classes

frmMain

This is the main class of the project. The class responsible to display the main application window and to manage the user interface.
It has no public methods/fields except its constructor.

The following points are related to the internal structure of this class.
In order to minimize dependencies between classes, we access other classes using covering functions. Doing so, we will be able to replace in the future parts of the GUI/Assembler/Simulator without major changes to the code.

Dealing with Task List

Function	Gets	Returns	Throws
AddTask	TaskType, Message, Line Number	Nothing	Nothing
ClearTasks	Nothing	Nothing	Nothing
updateTasksLinesNumber	FromLine, Offset	Nothing	Nothing

AddTask - Add task to the task list.
ClearTasks - Clear all tasks from the task list
updateTasksLinesNumber - Change all line numbers of tasks by offset, starting from given line number.

Event	Gets	Returns	Throws
OnTaskListClick	LineNumber	Nothing	Nothing

OnTaskListClick - Function to handle clicks on tasks list. Not to be called by the user, only by events.

Dealing with Output window

Function	Gets	Returns	Throws
ClearOutputWindow	Nothing	Nothing	Nothing
AppendToOutputWindow	StringtoAppend	Nothing	Nothing

ClearOutputWindow - Clears the output window
AppendToOutputWindow - Append text to output window

Dealing with Documents

Function	Gets	Returns	Throws
GetActiveDocumentIndex	Nothing	int	NoActivePageException
OnDocumentPosition	Point	Nothing	Nothing

GetActiveDocumentIndex - Get the page index of the active document
OnDocumentPosition - Update the status bar location when caret changes position

Interface - Helping Functions

Function	Gets	Returns	Throws
StatusBarMessage	string	Nothing	Nothing
GiveFocusToTheActiveDocument	Nothing	Nothing	Nothing
ArrangeControls	Nothing	Nothing	Nothing
ClearDocumentInfoStatusBar	Nothing	Nothing	Nothing

StatusBarMessage - Helping function for sending messages to status bar
GiveFocusToTheActiveDocument - Gives the focus to the active document
ArrangeControls - Update controls when the window size is changing
ClearDocumentInfoStatusBar - Clears document info form status bar

Files - Creating, loading, saving and closing

Function	Gets	Returns	Throws
CreateNewDocument	Nothing	Nothing	Nothing
DoOpen	Nothing	Nothing	Nothing
PrepareDocumentForClosing	frmToClose	bool	Nothing
DoSave	frmToSave	bool	Nothing
InterfaceDoSave	Nothing	bool	Nothing
InterfaceDoSaveAs	Nothing	bool	Nothing
InterfaceDoClose	Nothing	bool	Nothing
CloseAllDocuments	Nothing	bool	Nothing

CreateNewDocument - Create new document
DoOpen - Open assembly file
PrepareDocumentForClosing - Preparing document to be close. After calling to this function, we need to remove the page tab from the documents list.
DoSave - Save a document.
InterfaceDoSave - Interface Save - Actions to do when the user press on "Save" button
InterfaceDoSaveAs - Interface Save As - Actions to do when the user press on "Save As" button
InterfaceDoClose - Close the active document
CloseAllDocuments - Close all open documents

frmTaskList

This class is responsible to display the task list - it contains the errors and warning the user got during the code compilation.
The class raise event - *OnClickEvent*, when user double click on list item, announcing the code line that related to the clicked entry.

Public Functions:

Function	Gets	Returns	Throws
AddTask	TaskType, Message, Line Number	Nothing	Nothing
ClearTasks	Nothing	Nothing	Nothing
updateTasksLinesNumber	FromLine, Offset	Nothing	Nothing

AddTask - Add task to the task list.
ClearTasks - Clear all tasks from the task list
updateTasksLinesNumber - Change all line numbers of tasks by offset, starting from given line number.

frmCompileMessages

This class is responsible to display the output window - contains messages arrived during compile time.

Public Properties:

Output - Get/Set the text in the output window

frmEditor

This class represents a single document. It actually contains the document.
It responsible to display the document and giving interface functions for the main program to deal with it.
In the main program there may be many instances of this class.
The class raise event - *OnPositionChange* - when the caret position is changing.

Public Functions:

Function	Gets	Returns	Throws
Cut	Nothing	Nothing	Nothing
Copy	Nothing	Nothing	Nothing
Paste	Nothing	Nothing	Nothing
SelectAll	Nothing	Nothing	Nothing
FocusDocument	Nothing	Nothing	Nothing

Cut - Cut text
Copy - Copy text
Paste - Paste text
SelectAll - Select all text
FocusDocument - Give focus to the document

Public Properties:

bDocumentSaved - Did we save the current document
sFileName - File Name (including path)

Public Events:

OnPositionChange - Occurs when caret's position is changing

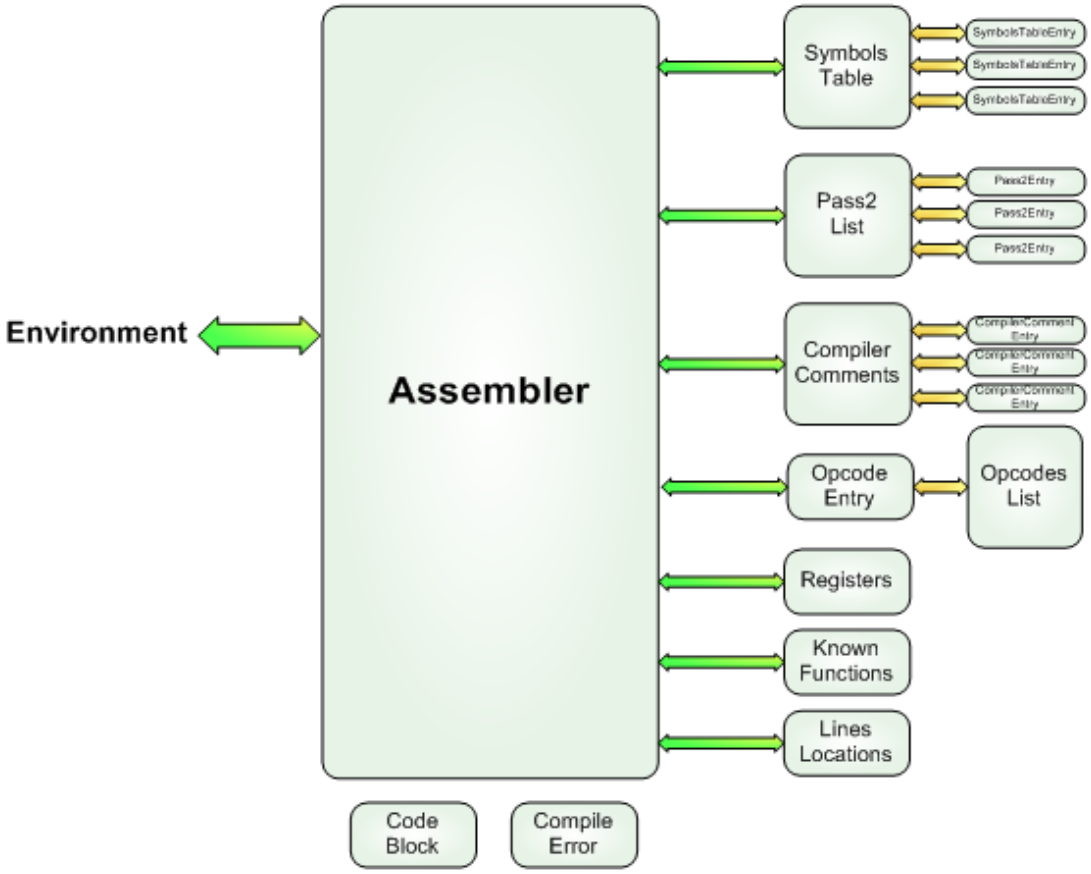
Private Functions:

Function	Gets	Returns	Throws
GetLocationOnRTB	RichTextBox	Point	Nothing

GetLocationOnRTB - Returns the line and the column of the caret.

6. Assembler's Class & Algorithms

1. Overview



Data Structures

- Several classes in the project contain VAX11 specific information.
  The information includes registers names, known functions addresses and opcodes names and operands.
- The idea is to split the source analyze process to few categories, making the analyzing of each category simple.

KnownFunctions, Registers, Opcode Table

- Registers class contains the VAX11 registers names.
- KnownFunctions class contains known operation system functions.
- The Opcode Table contains information about all the known opcodes: the opname of each command, its machine code, the number of operands the command needs to get, and also the expected type of each operand.
- Implemented using Hash Table to achieve O(1).

Classes

Assembler

- The main class of this module.
- The only class that communicate with the working environment
- Gets assembly source code and generate machine code
- Communicate with the environment using events to inform it when it found errors in the code.

SymbolsTable, SymbolsTableEntry

- VAX11 Assembler allows the users to use symbols - labels and constants, to make the code more readable.
- While analyzing the code, the compiler collects all the symbols it found in the Symbols Table.
- When the user uses a symbol, the compiler is using the symbols table to find the value of the symbol.
- Implemented using Hash Table to achieve O(1).

Pass2List, Pass2Entry

- The user may use symbols that will appear later in the code. In this case, the compiler cannot find the value that the symbol represents.
- In such cases, the compiler adds all the unknown symbols to Pass2List.
  After the first pass on the code, the compiler moves on this list, and checks if it can solve the symbols again.
- Implemented using Hash Table to achieve O(1).

CompilerComment, CompilerCommentEntry

- While compiling the user's code, the compiler might find syntax errors.
- CompilerComment class is class that maintains the list of all the compile errors.

Algorithm

The compiling process is divided to several parts:

- PreCompiler - In order to make the analyzing of the code more simple - we first do some preparing processing on the user's code. The PreCompiler cleans the user's comments, and clean white spaces from the beginning of each line, from its end and between the words.
- Pass 1 - First analyzing of the code. Creates symbols table, Creates table of the missing symbols for the second pass. Convert all the opnames and the operands to machine code.
- Pass 2 - After the first pass, the compiler moves again on all the expressions it had problems to translate in the first pass, and tries to translate it again.
  If it succeeds to translate all the expressions, then the compile process is over.

2. Data Structures

Opcodes Table

The opcodes table contains information about the different operands the assembler recognizes.
The opcodes table is constant list - it never changes and all its content is known.

Each entry in the table has the following fields:

Field	Type	Comments
OpName	string	Name of the operation
OpCode	int	Opcode of the operation
Operands	int	Number of expected operands
OpType	string	Operands types

The format of the OpType field is 2 characters per operand (max 6 operands).

1st character:

- a - address
- b - immediate value (of the 2nd type)
- r - read access
- w - write access
- p - privilege register

2nd character:

- 1 - byte
- 2 - word
- 4 - long
- 8 - quad

Known Procedures Table

The known procedures table contains the name and the simulator addresses of the known procedures of VAX11 - printf, malloc, exit, etc.
Each entry in the table has the following fields:

Field	Type	Comments
Name	string	Procedure's Name
Address	int	Address of the symbol

Known Registers Table

The known registers table contains the name and the numbers of the known registers of VAX11.
Each entry in the table has the following fields:

Field	Type	Comments
Name	string	Register's Name
Number	int	Register Number

Assembler Messages

Message Number	Message
0	Compile Succeed
1	Number expected
2	Unexpected end of line
3	Label already defined
4	Double label in line
5	Illegal addressing mode
6	Number is too big
7	Register expected
8	Symbol expected
9	Register is not allowed here
10	Unrecognized procedure
11	) expected
12	Unrecognized opcode name
13	Immediate addressing mode cannot be destination
14	] expected
15	Rn must be different than Rx
16	End of operand expected
17	End of line or comment expected
18	Immediate addressing mode is not allowed here
19	Unrecognized Directive
20	Defined symbol or number expected
21	" expected
22	Value (number or symbol) expected
23	, expected
24	PC cannot be used here
25	Displacement too big
26	Procedure not allowed here
27	Divide by zero
28	Undefined symbol
29	Program must begin with .TEXT
30	Privileged register cant be used here
31	Illegal expression
32	Expected: Privileged register

Known VAX11 Functions:

Function Name	Address
GETCHAR	0xffff
PUTCHAR	0xfffe
GETS	0xfffd
PUTS	0xfffc
SCANF	0xfffb
PRINTF	0xfffa
MALLOC	0xfff9
EXIT	0xfff8
FREE	0xfff7
UNGETCHAR	0xfff6

VAX11 Register List

Register Name	Register Number
R1	1
R2	2
R3	3
R4	4
R5	5
R6	6
R7	7
R8	8
R9	9
R10	10
R11	11
R12	12
R13	13
R14	14
R15	15
AP	12
FP	13
SP	14
PC	15
SCBB	17
IPL	18
SIRR	20
SISR	21
ICCS	24
NICR	25
ICR	26
OSC	30
RXCS	32
RXDB	33
TXCS	34
TXDB	35

VAX11 commands

Opname	Number of Operands	OpType	Opcode
ACBA	4	r1r1w1b2	0x9D
ACBB	4	r1r1w1b1	0x9D
ACBL	4	r4r4w4b2	0xF1
ACBW	4	r2r2w2b2	0x3D
ADDB2	2	r1w1	0x80
ADDB3	3	r1r1w1	0x81
ADDL2	2	r4w4	0xC0
ADDL3	3	r4r4w4	0xC1
ADDW2	2	r2w2	0xA0
ADDW3	3	r2r2w2	0xA1
ADWC	2	r4w4	0xD8
AOBLEQ	3	r4w4b1	0xF3
AOBLSS	3	r4w4b1	0xF2
ASHL	3	r1r4w4	0x78
BBC	3	r4a1b1	0xE1
BBCC	3	r4a1b1	0xE5
BBCS	3	r4a1b1	0xE3
BBS	3	r4a1b1	0xE0
BBSC	3	r4a1b1	0xE4
BBSS	3	r4a1b1	0xE2
BCC	1	b1	0x1E
BCS	1	b1	0x1F
BEQL	1	b1	0x13
BEQLU	1	b1	0x13
BGEQ	1	b1	0x18
BGEQU	1	b1	0x1E
BGTR	1	b1	0x14
BGTRU	1	b1	0x1A
BICB2	2	r1w1	0x8A
BICB3	3	r1r1w1	0x8B
BICL2	2	r4w4	0xCA
BICL3	3	r4r4w4	0xCB
BICPSW	1	r2	0xB9
BICW2	2	r2w2	0xAA
BICW3	3	r2r2w2	0xAB
BISB2	2	r1w1	0x88
BISB3	3	r1r1w1	0x89
BISL2	2	r4w4	0xC8
BISL3	3	r4r4w4	0xC9
BISPSW	1	r2	0xB8
BISW2	2	r2w2	0xA8
BISW3	3	r2r2w2	0xA9
BITB	2	r1w1	0x93
BITL	2	r4w4	0xD3
BITW	2	r2w2	0xB3
BLBC	2	r4b1	0xE9
BLBS	2	r4b1	0xE8

BLEQ	1	b1	0x15
BLEQU	1	b1	0x1B
BLSS	1	b1	0x19
BLSSU	1	b1	0x1F
BNEQ	1	b1	0x12
BNEQU	1	b1	0x12
BPT	0		0x03
BRB	1	b1	0x11
BRW	1	b2	0x31
BSBB	1	b1	0x10
BSBW	1	b2	0x30
BVC	1	b1	0x1C
BVS	1	b1	0x1D
CALLG	2	a1a1	0xFA
CALLS	2	r4a1	0xFB
CASEB	3	r1r1r1	0x8F
CASEL	3	r4r4r4	0xCF
CASEW	3	r2r2r2	0xAF
CLRB	1	w1	0x94
CLRL	1	w4	0xD4
CLRW	1	w2	0xB4
CMPB	2	r1r1	0x91
CMPC3	3	r2a1a1	0x29
CMPC5	5	r2a1r1r2a1	0x2D
CMPL	2	r4r4	0xD1
CMPW	2	r2r2	0xB1
CVTBL	2	r1w4	0x98
CVTBW	2	r1w2	0x99
CVTLB	2	r4w1	0xF6
CVTLW	2	r4w2	0xF7
CVTWB	2	r2w1	0x33
CVTWL	2	r2w4	0x32
DECB	1	w1	0x97
DECL	1	w4	0xD7
DECW	1	w2	0xB7
DIVB2	2	r1w1	0x86
DIVB3	3	r1r1w1	0x87
DIVL2	2	r4w4	0xC6
DIVL3	3	r4r4w4	0xC7
DIVW2	2	r2w2	0xA6
DIVW3	3	r2r2w2	0xA7
EDIV	4	r4r4w4w4	0x7B
HALT	0		0x00
INCB	1	m1	0x96
INCL	1	m4	0xD6
INCW	1	m2	0xB6
INSQUE	2	a1a1	0x0E
JCC	1	b2	0x31031F
JCS	1	b2	0x31031E
JEQL	1	b2	0x310312
JEQLU	1	b2	0x310312
JGEQ	1	b2	0x310319

JGEQU	1	b2	0x31031F
JGTR	1	b2	0x310315
JGTRU	1	b2	0x31031B
JLBC	2	r4b2	0x3103E8
JLBS	2	r4b2	0x3103E9
JLEQ	1	b2	0x310314
JLEQU	1	b2	0x31031A
JLSS	1	b2	0x310318
JLSSU	1	b2	0x31031E
JMP	1	a1	0x17
JNEQ	1	b2	0x310313
JNEQU	1	b2	0x310313
JSB	1	a1	0x16
JVC	1	b2	0x31031D
JVS	1	b2	0x31031C
LOCC	3	r1r2a1	0x3A
MATCHC	4	r2a1r2a1	0x39
MCOMB	2	r1w1	0x92
MCOML	2	r4w4	0xD2
MCOMW	2	r2w2	0xB2
MFPR	2	p4w4	0xDB
MNEGB	2	r1w1	0x8E
MNEGL	2	r4w4	0xCE
MNEGW	2	r2w2	0xAE
MOVAB	2	a1w1	0x9E
MOVAL	2	a4w4	0xDE
MOVAW	2	a2w2	0x3E
MOVB	2	r1w1	0x90
MOVC3	3	r2a1a1	0x28
MOVC5	5	r2a1r1r2a1	0x2C
MOVL	2	r4w4	0xD0
MOVPSL	1	w4	0xDC
MOVTC	6	r2a1r1a1r2a1	0x2E
MOVTUC	6	r2a1r1a1r2a1	0x2F
MOVW	2	r2w2	0xB0
MOVZBL	2	r1w4	0x9A
MOVZBW	2	r1w2	0x9B
MOVZWL	2	r2w4	0x3C
MTPR	2	r4p4	0xDA
MULB2	2	r1w1	0x84
MULB3	3	r1r1w1	0x85
MULL2	2	r4w4	0xC4
MULL3	3	r4r4w4	0xC5
MULW2	2	r2r2	0xA4
MULW3	3	r2r2w2	0xA5
NOP	0		0x01
POPR	1	r2	0xBA
PUSHAB	1	a1	0x9F
PUSHAL	1	a4	0xDF
PUSHAW	1	a2	0x3F
PUSHL	1	r4	0xDD
PUSHR	1	r2	0xBB

REI	0		0x02
REMQUE	2	a1w4	0x0F
RET	0		0x04
ROTL	3	r1r4w4	0x9C
RSB	0		0x05
SBWC	2	r4w4	0xD9
SCANC	4	r2a1a1r1	0x2A
SKPC	3	r1r2a1	0x3B
SOBGEQ	2	w4b1	0xF4
SOBGTR	2	w4b1	0xF5
SPANC	4	r2a1a1r1	0x2B
SUBB2	2	r1w1	0x82
SUBB3	3	r1r1w1	0x83
SUBL2	2	r4w4	0xC2
SUBL3	3	r4r4w4	0xC3
SUBW2	2	r2w2	0xA2
SUBW3	3	r2r2w2	0xA3
TSTB	1	r1	0x95
TSTL	1	r4	0xD5
TSTW	1	r2	0xB5
XORB2	2	r1w1	0x8C
XORB3	3	r1r1w1	0x8D
XORL2	2	r4w4	0xCC
XORL3	3	r4r4w4	0xCD
XORW2	2	r2w2	0xAC
XORW3	3	r2r2w2	0xAD

3. Assembler Class and Internal Data Structures

Assembler Class Interface

The following table presents the public methods of the assembler class:

Operation	Gets	Returns	Throws
Constructor	Nothing	Nothing	Nothing
GetSymbolsTable	Nothing	Code's symbols table	Nothing
GetCompileMessages	Nothing	Compile Comments	Nothing
CompileCode	Assembly Code	Machine Code	CompileError
GetMachineCode	Nothing	Machine Code	Nothing
GetLSTFile	Nothing	The LST File	Nothing

Constructor - Initialize the assembler.
CompileCode - Gets code and compile it. This is the main function of this module.
GetSymbolsTable - Returns the symbols table generated during the compiling process.
GetCompileMessages - Returns the compiling errors.
GetMachineCode - Gets the machine code that the assembler created.
GetLSTFile - Generate and create LST file for the compiled code. LST file is file that mix the source code with the machine code, allowing the user to see the compiler output.

CodeBlock (Class)

In order to pass code of blocks from the different functions to the main compile process, we use CodeBlock objects, that contains the machine code we want to pass and its size. CodeBlock is sub-class of Assembler class.

Class Properties:

Property	Type	Comments	Gets / Sets
MachineCode	Byte[]	MachineCode	Get / Set
Size	int	Size of code	Get

MachineCode - Contains machine code, as array of bytes
Size - Machine Code size, in bytes.

Methods:

Operation	Gets	Returns	Throws
Operator+	CodeBlock	CodeBlock	Nothing
Byte Casting	Byte	CodeBlock	Nothing
Int Casting	Int	CodeBlock	Nothing
Constructor 1	Nothing	Nothing	Nothing
Constructor 2	Number, BlockSize	Nothing	IllegalBlockSize
ChangeCodeblock	Position, CodeBlock	CodeBlock	IndexOutOfRange

Constructor 1 - Creates an empty CodeBlock
Constructor 2 - Creates CodeBlock that contains Number, in size of BlockSize
Example: CodeBlock(15,4) → 0F 00 00 00
Operator+ - Overloaded operator +. Allowing us to sum several BlockCode, to create bigger CodeBlock that contains them.
Byte Casting - Allows casting from byte to CodeBlock
Int Casting - Allows casting from int to CodeBlock
ChangeCodeblock - Changes given CodeBlock

Opcode Entry (Class)

When the program wants to get information from the opcodes table, it creates object from Opcode Entry, which contains the request information from the table.

Class Fields: Same as those of the opcodes table.

Class Methods:

Operation	Gets	Returns	Throws
Constructor	OpName	Nothing	UnrecognizedCommand

Constructor - Gets opname and gets all the information related to that opname.

Class Properties

Property	Type	Comments	Gets / Sets
OpCode	int	Opcode of the operation	Get
Operands	int	Number of expected operands	Get
OpType	string	Operands types	Get

Symbols Table (Class)

The symbols table contains information about all the labels defined in the source file. The information is entered to the table during pass 1 of the assembler. Each entry in the table has the following fields:

Field	Type	Comments
Name	string	Symbol's Name
Value	int	Address of the symbol
Type	enum	Constant / Label
Line	int	The line number the symbol defined at

Class Methods:

Operation	Gets	Returns	Throws
Constructor	Nothing	Nothing	Nothing
AddEntry	New symbol table entry	Nothing	LabelAlreadyExists
ResetTable	Nothing	Nothing	Nothing
SymbolValue	string - name	int - value	unknown symbol

Constructor - Creates new symbol table.
AddEntry - Add new symbol to the table.
ResetTable - Resets all symbols table entries.
SymbolValue - Gets symbols and returns its value.

LinesLocations (Class)

For debug (breakpoints) support and for making the LST file, the assembler saves the starting address of each line in the code.

Class Methods:

Operation	Gets	Returns	Throws
Constructor	Nothing	Nothing	Nothing
ResetLines	Nothing	Nothing	Nothing
AddLine	Line, Address	Nothing	Nothing
GetStartingAddress	Line Number	Starting Address	NoSuchLine

Constructor - Initialize the lines list.
AddEntry - Add new entry to the list.
ResetLines - Resets all list entries.
GetStartingAddress - Gets line and returns its starting address.

CompilerComment (Class)

When returning compile messages, the assembler returns array of CompilerComment objects.

Class fields:

Field	Type	Comments
MessageNumber	int	Number of message
Line	int	The relevant line number (or -1)

Pass2List (Class)

We create the list on Pass1. The list is being used on Pass2.
The list contains the locations that pass2 need to change and some information about the needed changes.

Class fields:

Field	Type	Comments
Where	int	LC - Location where the update need to take place
Size	short int	How many bytes are allowed to be change
Expression	string	The all line of the expression
negative	boolean	True if the number is allowed to be negative

4. Assembler Algorithms

General

When we create assembler object, the constructor is responsible to make it usable.
After we use the object to compile code, we can use *GetSymbolsTable*() and *GetCompileMessages*() to get information about the compiling process.
The *CompileCode*() function starts the analyzing of the user's code. During its execution, it calls to *DoPass1*() and *DoPass2*() that do the compiling process.
To get the compiler's results, we use *GetMachineCode*() and *GetLSTFile*().

Constructor

Gets: Nothing
Returns: Nothing
Throws: Nothing
Description: Initialize assembler object

Algorithm:

1) **Reset** Symbols Table
2) **Reset** Compile Messages
3) **Reset** Machine Code
4) **Reset** Pass2 List

GetSymbolsTable

Gets: Nothing
Returns: Symbols Table
Throws: Nothing
Description: Returns the assembler symbols table

Algorithm:

1) **Return** the internal Symbols Table.

GetCompileMessages

Gets: Nothing
Returns: Compile Comments
Throws: Nothing
Description: Returns the compile time messages

Algorithm:

1) **Return** the internal list of complier comments.

GetLstFile

Gets: Nothing
Returns: The LST File
Throws: Nothing
Description: Using the symbols table, the assembly code and the machine code, the function generate LST file for the code and returns it.

Algorithm:

1) RetFile ← Empty Block
2) Add header line to RetFile: Address, Machine Code, Line and Line Source
3) **For** Line = 1 **to** TotalNumberOfLines
4) Use LinesLocationTable to find the machine code and the address that releated to the current line.
5) Add the current line with all the relevant fields to RetFile.

GetMachineCode

Gets: Nothing
Returns: Machine Code
Throws: Nothing
Description: Returns the machine code generated by the compiler

Algorithm:

1) **Return** the machine code.

CompileCode

Gets: Assembly Code
Returns: Nothing
Throws: CompileError
Description: Compile the code, returns the machine code

Algorithm:

1) **Reset** all internal data structures.
2) **Save** original code, for future use
3) **If** there is no code, **then return** empty block.
4) **Try**:
5) sCleanCode ← PreCompiler(Assembly Code)
6) TempBlockCode ← DoPass1(sCleanCode)
7) MachineCode ← DoPass2(TempBlockCode)
8) **Catch:** (Compile Errors)
9) Create CompileMessages array contains the errors.
10) **Throw** CompileError

PreCompiler

Gets: Assembly Code
Returns: Clean Assembly Code
Throws: Nothing
Description: Clean the code from comments and spaces
Comment: Be careful not to clean empty lines, so line number will stay untouched.

Algorithm:

1) **While** not end of source
2) CurLine ← Read Line
3) **While** not end of line
4) **If** found '#' on line, **then** cut from it till end of line.
5) Clean spaces between the words.
6) Add the line to the block we return
7) **Return** ReturnedBlock

DoPass1

Gets: CodeBlock
Returns: CodeBlock
Throws: CompileError
Description: Move first time on the code, build symbols table

Algorithm:

```
 1)  LC ← 0
 2)  LineNumber ← 0
 3)  ErrFlag ← false
 4)  FinalCode ← Empty Block
 5)  CurWord ← ReadNextWord(LinePointer)
 6)  If  CurWord is not ".TEXT" then
 7)          add CompileError("Program must begin with .TEXT",
                  LineNumber)
 8)          Throw CompileError
 9)  While not reach end of code
10)          LineNumber ← LineNumber +1
11)          LinePointer ← 0
12)          LabelDefined  ←  false
13)          Call AddToLinesList(LineNumber, LC)
14)          If reached end of line, then continue
15)          CurWord ← ReadNextWord(LinePointer)
16)          While CurWord ends with ":" (label) do
17)                  If the LabelDefined = True then
18)                          add CompileError("Double label in line",
                                  LineNumber)
19)                          ErrFlag ← true
20)                  Else
21)                          LabelDefined  ←  true
22)                          Try
23)                                  UpdateSymbolsTable(CurWord, LC, Lbl)
24)                          Catch (LabelAlreadyExists)
25)                                  add CompileError("Label already defined",
                                          LineNumber)
26)                                  ErrFlag ← true
27)                  CurWord ← ReadNextWord(LinePointer)
                     (Now lets start the real job ☺)
28)          If CurWord starts with "." then
29)                  Try
30)                          CodeBlock ← AnalyzeDirective(CurWord,
                                                  LC, LinePointer)
31)                          FinalCode ← FinalCode + CodeBlock
32)                          LC ← LC+CodeBlock.Size
33)                  Catch (CompileError)
```

```
34)                         add CompileError(CompileError.Error,
                            LineNumber)
35)                         ErrFlag ← true
36)         Else
37)             Try
                                CodeBlock ← AnalyzeCommand(CurWord,
38)                                         LC,LinePointer)
39)                         FinalCode ← FinalCode + CodeBlock
40)                         LC ← LC+CodeBlock.Size
41)             Catch (CompileError)
42)                         add CompileError(CompileError.Error,
                            LineNumber)
43)                         ErrFlag ← true
44)         If line not ended then
45)             add CompileError("End of line or comment expected",
                            LineNumber)
46)             ErrFlag ← true
47)     If ErrFlag = true then throw CompileError
48)     Return FinalCode
```

AnalyzeCommand

Gets: WordToAnalyze , LC
Returns: CodeBlock
Throws: CompileError (including error field)
Description: Analyze the command that found at WordToAnalyze,
Returns the machine opcode of the operation in the code block.
Analyze the operands and add it to the block.

Algorithm:

```
1)  CurCommand ← New OpcodeEntry(WordToAnalyze)
    (Might throw exception)
2)  ReturnedCodeBlock.MachineCode ← CurCommand.OpCode
3)  For Counter = 0 to CurCommands.Operands-1 do
        CurOperand ← FetchOperand(CurCommand.OpType[2*Counter],
4)                              CurCommand.OpType[2*Counter+1]-'0')
        (Might throw CompileError)
5)      ReturnedCodeBlock ← ReturnedCodeBlock +CurOperand
6)      LC ← LC+CodeBlock.Size
7)      If Counter = CurCommand.Operands then break
8)      If ReadNextChar(WordToAnalzye) is not ',' then
9)          Throw CompileError(", expected")
```

FetchOperand

Gets: Expected Operand Type (OpType), Expected Operand Size (OpLen), LC, IsPrivilegedCommand
Returns: CodeBlock
Throws: CompileError (including error field)
Description: Analyze the operand where the given LC is. Returns the machine code for the operand.

AnalyzeDirective

Gets: WordToAnalyze, LC
Returns: CodeBlock
Throws: CompileError
Description: Analyze the directive from WordToAnalyze.

Algorithm:

```
1)  TempBlock ← Empty CodeBlock
2)  WordToAnalyze ← MakeLower(WordToAnalyze)
3)  If WordToAnalyze = ".data" or ".text" or ".org" then Return TempBlock
4)  If WordToAnalyze = ".space" then
5)          TempNum ← ReadNextNumber(LinePointer)
6)          Return CodeBlock(0,TempNum)
7)  If WordToAnalyze = ".set" then
8)          TempName ← ReadNextWord(LinePointer)
9)          If TempName is empty then throw CompileError("symbol expected")
10)         If ReadNextChar isn't "," then throw CompileError (", expected")
11)         SymbolTable.AddEntry(TempName,CalcExpersion(WordToAnalyze))
12)         Return TempBlock
13) If WordToAnalyze = ".ascii" then
14)         Return CodeBlock(AnalyzeString(WordToAnalyze))
15) If WordToAnalyze = ".asciz" then
16)         Return CodeBlock(AnalyzeString(WordToAnalyze) + ascii(0))
17) If WordToAnalyze = ".byte" then
18)          BlockSize ← 8
19) Else if WordToAnalyze = ".word" then
20)          BlockSize ← 16
21) Else if WordToAnalyze = ".int" then
22)          BlockSize ← 16
23) Else if WordToAnalyze = ".long" then
24)          BlockSize ← 32
25) Else if WordToAnalyze = ".quad" then
26)          BlockSize ← 64
27) Else
28)         Throw CompileError ("Unrecognized Directive")
```

```
29)  Do
30)       TempNum ← CalcNextExpersion(LinePointer)
31)       TempNum ← TempNum modulo 2^BlockSize
32)       TempBlock ← TempBlock +CodeBlock(TempNum,BlockSize/8)
33)  While (ReadNextChar(WordToAnalyze) = ",")
34)  Return TempBlock
```

ReadNextNumber

Gets: Expression
Returns: A number.
Throws: CompileError("Number expected")
Description: The function reads the next word and convert it to number
The function convert hex/decimal numbers.
If the next word cannot be converted then the function throw exception.

ReadNextWord

Gets: Expression
Returns: A string
Throws: Nothing
Description: The function returns the next word.
Word may contain only big letters, small letters and numbers.

ReadNextChar

Gets: Expression
Returns: A char
Throws: Nothing
Description: The function returns the next char on Expression that is not white spaces.

CalcExpression

Gets: Expression
Returns: A number.
Throws: CompileErrors (Defined symbol or number expected, Value expected, number too big)
Description: This function Gets string with expression contains labels, numbers and operators and solves it. If the string doesn't contains ecuation, it throws exception.
If one of the labels isn't defined, it throw exception, and it
is the caller responsability to add the expression to Pass2List.
If the user tries to divide by zero, it throws exception.
If one of the numbers in the equation is too big, it throws an exception.

AnalyzeString

Gets: Expression
Returns: A string
Throws: CompileError(""\"" expected")
Description: The function read string from "\"" to the first "\"" that don't have before him slash "\\" and replace the codes like "\n" to their meaning. If there is not "\"" at the end or at the beginning of the string then the function throws CompileError(""\"" expected").

DoPass2

Gets: CodeBlock
Returns: CodeBlock
Throws: CompileError
Description: Second pass on the code, generate final machine code
We assume in this point that all the definitions of the constants
(Those who need to be in the symbol table) already defined
on AnalyzeDirective, and now we need to update only expressions
with the constants that need update on the memory

Algorithm:

```
 1)  For each record on Pass2List do
 2)      Try
 3)          value<-- CalcNextExpression (Pass2List.Expression)
 4)      Catch ()
 5)          add CompileError(unknown symbol,LineNumber)
 6)          wascatch<--true
 7)          continue //foreach
 8)      If overRange(value,Pass2List.negative,Pass2List.size) then
 9)          If Pass2List.negative
10)              add CompileError(number too big,LineNumber)
11)          else
12)              add CompileError(displacement too big,LineNumber)
13)      else ChangeCodeblock
             (Pass2List.where,CodeBlock(value,Pass2List.size))
14)  If was catch then Throw ("unknown symbol")
```

overRange

Gets: value, negative allowed, num of bytes
Returns: Boolean
Throws: nothing
Description this function check if the value is over the allowed range that determine from negativity and number of bytes
Algorithm:

1) **If** negative allowed and abs(value)< 2^(8*num bytes) **then** return **True**
2) **If** negative not allowed and abs(value)<2^(8*num bytes-1) return **True**

- **return False**

7. Bibliography

Books

- C# and .NET Technology, Izhak Gerber, 2002

Summaries

- C# for C++ programmers - Nir Adar - http://underwar.livedns.co.il/
- Regular Expression - A.M. Kuchling
- Learning to Use Regular Expressions, Dario F. Gomes, 2001, http://ibm.com/developer/
- C# Regular Expressions - Brad Merrill, 2001, http://www.oreilly.com/

Open Sources

- Magic - The User Interface Library for .Net - Manual Pages, 2003, http://www.dotnetmagic.com