OneUp Wi-11 Simulator: Programmer's Guide

Generated by Doxygen 1.7.3

Wed Feb 23 2011 02:42:08

Contents

Chapter 1

Introduction

Primary author: Logan Coulson

Edited by: Andrew Canale

Doxygen: Andrew Groot

1.1 Introduction

The "Wi-11 Machine" is a theoretical 16-bit computer architecture. It has 8 general purpose registers, 3 condition code registers (CCRs), and a program counter (PC). The Wi-11 Simulator emulates the execution of this Wi-11 processor. However, programming for the Wi-11 Simulator is difficult as it only accepts encoded object files as input. To aid the user, this assembler translates assembly language into usable object files for the Wi-11 Simulator. Before alterations are made to the programming of this assembler, a basic understanding of the environment is required. In particular, one needs to understand the input and output formats, how the components interact, and the programming conventions used.

1.2 Input

1.2.1 Format

The input file provided to the assembler must have limitations on the maximum size of the symbol table, the .o file, and the literal table. When the user runs the program, they may add a -s# to the end of their command. If the user does, that number is the maximum number of symbols allowed. The maximum number of literals allowed is one half the maximum number of symbols specified by the user. Additionally, the number of source records/lines in the object file produced by the assembler is limited to twice the number of symbols specified by the user. The default is two thousand source records, one thousand symbols, and five hundred literals. Each line of the input file must conform to the following structural organization:

1.2.1.1 Labels

With the exception of .ORIG and EQU, labels are optional per line. Labels may only be defined from the beginning of a line until the first whitespace. Labels must not begin with an uppercase 'R', nor begin with a lowercase 'x', as these indicate a register or a hexadecimal number respectively. Labels are case-sensitive, and may contain only alphanumeric ASCII characters. The first character in a label must be alphabetic. If a label is defined, it can be used as a reference in place of any Operand in Operation/Pseudo-Operation calls, except within operands in the same line on which the label is defined. Additionally, if a label is being used to replace a register operand, it must be a number zero through seven. Labels start on the leftmost position of the text file, and continue until a whitespace is encountered. When a symbol is used as the last argument for ADD or AND instructions, it is always interpreted as an imm5 operand rather than a source register.

1.2.1.2 Whitespace

Whitespace is any number of spaces and tabs in a row. Outside of comments, whitespace must exist before an operation (but after a label, if present), between an operation and its operands, and after the last operand to the end of the line or the start of a comment.

1.2.1.3 Operations

The operation field must be preceded on the same line by whitespace, or a label followed by whitespace. Operation and pseudo-operation names must be uppercase letters. Pseudo-operations (for use only by the compiler) may take the place of an operation, and are specified by beginning with a period. A list of all operations and pseudo-operations can be found in the 'Instructions' section of this document. There must be whitespace between the operation field, and its operands.

1.2.1.4 Operands

Operands continue to whitespace/end of line, or a semicolon. The types and numbers of operands are restricted by the specific operation or pseudo-operation invoked. To see these limitations, refer to section 1.2.2. A comma must be present between any two operands. No whitespace is permitted between or within operands. This assembler supports the following operands:

Registers

- Registers are specified by an uppercase ‘R’, followed by the number of the specific register. For example, register one is written as ‘R1’. The registers available to the user range from R0 – R7.

Constants

- Constants must be either decimal or hexadecimal. Decimal constants used as an operand must be immediately preceded by a pound sign ‘::’. Hexadecimal constants are preceded by a lowercase ‘x’. (i.e. Decimal 10 is written as #10. Hexadecimal 2C is written as x2C). An imm5 operand must be in the range #-16..#15, or x0..x1F. An address operand must be in the range #0..#65535, or x0..xFFFF. Note that only the least significant 9 bits of this value are used in the machine code encoding. An index6 operand must be in the range #0...#63, or x0...x3F. A trapvect8 operand must be in the range #0..#255, or x0..xFF.

1.2.1.5 Comments

Comments start with a semi-colon ‘;’, and may only come after code present on the same line, if any code is present on the line.There must be whitespace between the operand and any comments. Alternatively, a comment may be present on a line by itself. If a comment spans multiple lines, a semicolon must be present at the beginning of every line.

1.2.1.6 Sample Input

```
; Should execute properly
; Example Program
Lab2EG   .ORIG   x30B0
count    .FILL   #4
Begin    LD      ACC,count       ;R1 <- 4
         LEA     R0,msg
loop     TRAP    x22             ;print "hi! "
         ADD     ACC,ACC,#-1     ;R1--
         BRP     loop
         JMP     Next
msg      .STRZ   "hi! "
Next     AND     R0,R0,x0        ;R0 <- 0
         NOT     R0,R0           ;R0 <- xFFFF
         ST      R0,Array        ;M[Array] <- xFFFF
         LEA     R5,Array
         LD      R6,=#100        ;R6 <= #100
         STR     R0,R5,#1        ;M[Array+1] <= xFFFF
         TRAP    x25
ACC      .EQU    #1
```

```
; ----- Scratch Space -----
Array     .BLKW   #3
          .FILL   x10
          .END    Begin
```

1.2.2 Instructions

This assembler must support many different functions. A basic understanding of the different functions is necessary. DR(Destination Register) is the location where the final value is stored. SR(Source Register) is the source of the numbers that are being manipulated by the operation. imm5(Immediate) is 5 bits, and it is sign extended to 16 bits when used. pgoffset9(Page Offset Nine) is used to form the last 7 bits for a memory access. The first 7 bits come from the PC(Program Counter). index6(Index Six) is used as a six bit number that is added to a register(BaseR) to determine an offset. In addition to the operations provided by the simulator, there are several pseudo ops that the assembler provides. A pseudo op is recognizable by the period '.' at the beginning of the command.

1.2.2.1 .ORIG

.ORIG x0-xFFFF

.ORIG must be the first non-comment record in the source program. The operand indicates the absolute address the program is to be stored in. If the operand is absent, the program may be stored anywhere. If there is no operand, the entire program must fit in one page of memory(512 16-bit words). It is also required to have a 6 character label, which is used to generate the header record in the .o file.

1.2.2.2 .END

.END x0-xFFFF

This operation indicates the end of the input program. The operand is optional, and it indicates where the program will start executing. If the operand is not present, it means that execution begins at the first address in the segment, and the assembler assumes an address of x0. There must be a .END command in every program.

1.2.2.3 .EQU

.EQU

.EQU equates the label to the operand, creating a constant. This constant can be any previously defined symbol or constant. .EQU requires a label and an operand to be present.

1.2.2.4 .FILL

.FILL #-32768-32768 or x0-xFFFF

.FILL defines a one word quantity, whose contents is the value of the operand. This operand can be in hexadecimal, decimal, or a label. If a label is used, its value must be within acceptable ranges (i.e. #-32768..xFFFF). This value is stored by the assembler in the word of memory that the .FILL occupies. Additionally, the assembler location counter is moved forward one word.

1.2.2.5 .STRZ

.STRZ "a string with escaped \"'s"

.STRZ defines a null-terminated block of words to hold a string of the characters in the operand field (in this case, the next 6 memory locations would hold the ASCII for "words\0"). ASCII representations are used to store these characters in memory.

1.2.2.6 .BLKW

.BLKW x1-xFFFF

.BLKW creates a block of storage. It occupies a number words of memory as indicated by the operand. The block of storage is not initialized.

1.2.2.7 ADD

ADD DR SR1 SR2 or ADD DR SR1 imm5

ADD takes the contents of SR1 and SR2 or imm5, adds them, and places the result in DR.

1.2.2.8 AND

AND DR SR1 SR2 or AND DR SR1 imm5

AND performs a logical 'and' between SR1 and either SR2 or imm5, and places the result in DR.

1.2.2.9 BRNZP

BRNZP pgoffset9

BRNZP checks to see if the CCR matches the NZP component of the branch command. If it does, it sets the PC to be a concatenation of bits [15:9] of the PC with bits [8:0] (pgoffset9) of the instruction. Any combination of N's, Z's, and P's are applicable here. If the character is present, the accompanying bit is set.

1.2.2.10 DBUG

DBUG

DBUG prints out the contents of all of the machine registers to the console.

1.2.2.11 JSR

JSR pgoffset9

JSR sets the PC to be a concatenation of bits [15:9] of the PC with bits [8:0] (pgoffset9) of the instruction. JSR also stores the current PC into R7, so that the RET instruction can restore the PC and cause execution to jump back to the instruction following the JSR.

1.2.2.12 JMP

JMP pgoffset9

JMP behaves the same as JSR, except that it does not store the current program counter into R7.

1.2.2.13 JSRR

JSRR BaseR index6

JSRR sets the PC equal to the contents of memory at location BaseR+index6. The original PC is stored in R7.

1.2.2.14 JMPR

JMPR BaseR index6

JMPR behaves the same as JSRR, except that it does not store the PC in R7.

1.2.2.15 LD

LD DR pgoffset9

LD loads DR with the contents of memory at the concatenation of [15:9] of the PC with bits [8:0] (pgoffset9) of the instruction.

1.2.2.16 LDI

LDI DR pgoffset9

LD loads DR with the contents of the memory address in the contents of memory at the address indicated by concatenation of [15:9] of the PC with bits [8:0] (pgoffset9) of the instruction.

1.2.2.17 LDR

LDR DR BaseR pgoffset9

DR is loaded with the operand at the address that is formed by adding the zero-extended six bit offset (index6) to the specified base register (BaseR). Thus, the index is always interpreted as a positive quantity (in the range #0-#63).

1.2.2.18 NOT

NOT DR SR

NOT inverts the bits of the SR and stores the results in DR.

1.2.2.19 RET

RET

RET copies the contents of R7 to the PC.

1.2.2.20 ST

ST SR pgoffset9

ST stores the contents of SR in memory at the concatenation of [15:9] of the PC with bits [8:0] (pgoffset9) of the instruction.

1.2.2.21 STI

STI SR pgoffset9

STI stores SR in the memory address in the contents of memory at the address indicated by concatenation of [15:9] of the PC with bits [8:0] (pgoffset9) of the instruction.

1.2.2.22 STR

STR SR BaseR index6

SR is saved at the address that is formed by adding the zero-extended six bit offset (index6) to the specified base register (BaseR). Thus, the index is always interpreted as a positive quantity (in the range 0-63).

1.2.2.23 TRAP

TRAP trapvect8

Depending on the trapvect8, trap does one of the following:

- x21 out Write the character in R0[7:0] to the console.
- x22 puts Write the null-terminated string pointed to by R0 to the console.
- x23 in Print a prompt on the screen and read a single character from the keyboard. The character is copied to the screen and its ASCII code is copied to R0. The high 8 bits of R0 are cleared.
- x25 halt Halt execution and print a message to the console.
- x31 outn Write the value of R0 to the console as a decimal integer.
- x33 inn Print a prompt on the screen and read a decimal number from the keyboard. The number is echoed to the screen and stored in R0. One may place specific requirements on the size, formatting, etc. of the input.
- x43 rnd Store a random number in R0.

1.3 Output

The assembler has two distinct outputs. The first outputs an object file usable by a simulator of the Wi-11 machine, and the second is a human readable listing for the user. The assembler will also display error codes on the console if necessary.

1.3.1 Object File

The object file consists of a header record, text records, and an end record as described in the Programmers guide for the OneUp Wi-11 simulator. These records are generated by the assembler from the program taken in as input.

1.3.2 Object File

The second output by the assembler is a more human readable output. It contains the source program, as well as its assembly. The current format for this data is (Address in hexadecimal) contents hexadecimal contents binary (line number) Label instruction operands. This allows for a easy to read output that has all of the necessary information in it.

1.3.2.1 Example

```
                               (   2) Lab2EG      .ORIG   x30B0
(30B0) 0004  0000000000000100 (   3) count       .FILL   #4
(30B1) 22B0  0010001010110000 (   4) Begin       LD      ACC,count       ;R1 <- 4
(30B2) E0B7  1110000010110111 (   5)             LEA     R0,msg
(30B3) F022  1111000000100010 (   6) loop        TRAP    x22             ;print "hi! "
(30B4) 127F  0001001001111111 (   7)             ADD     ACC,ACC,#-1     ;R1--
(30B5) 02B3  0000001010110011 (   8)             BRP     loop
(30B6) 40BC  0100000010111100 (   9)             JMP     Next
(30B7) 0068  0000000001101000 (  10) msg                 .STRZ   "hi! "
(30B8) 0069  0000000001101001 (  10)
(30B9) 0021  0000000000100001 (  10)
(30BA) 0020  0000000000100000 (  10)
(30BB) 0000  0000000000000000 (  10)
(30BC) 5020  0101000000100000 (  11) Next        AND     R0,R0,x0       ;R0 <- 0
(30BD) 9000  1001000000000000 (  12)             NOT     R0,R0          ;R0 <- xFFFF
(30BE) 30C3  0011000011000011 (  13)             ST      R0,Array       ;M[Array] <- xFFFF
(30BF) EAC3  1110101011000011 (  14)             LEA     R5,Array
(30C0) 2CC7  0010110011000111 (  15)             LD      R6,=#100       ;R6 <= #100
(30C1) 7141  0111000101000001 (  16)             STR     R0,R5,#1       ;M[Array+1] <= xFFFF
(30C2) F025  1111000000100101 (  17)             TRAP    x25
                               (  18) ACC                 .EQU    #1
(30C3)                         (  20) Array       .BLKW   #3
(30C6) 0010  0000000000010000 (  21)             .FILL   x10
(30C7) 0064  0000000001100100 ( lit) <100>
                               (  22)             .END    Begin
```

1.3.3 Errors

The Assembler can also output errors as included in the ResultCodes.h file. Additional errors can be added if needed, but adding superfluous errors should generally be avoided. Any error message is always returned, and the caller function either evaluates that result, or reports an error state.

1.4 Interaction

The interaction portion of this document describes how each function works, and what other functions that function calls.

1.4.1 Main

The main function uses the extractor class to generate the symbol tables. This is the first pass over the program. This assembler uses a 2-pass method. The second pass is made immediately after the first pass, and it is also done in the extractor class, using the line class. After that, it calls the result function to generate the users listing. It then uses the printer class to print the resulting .o file.

1.4.2 Extractor

The extractor uses the SymbolTable class to create a symbol table for the labels and literals as it parses through the input program. It also uses the line class to generate the rest of the .o file as it goes through.

1.4.3 Line

The line class is responsible for converting each line into the corresponding text record in the .o file. It uses the symbol table and word to do this.

1.4.4 Table

The symbol table keeps track of all of the symbols, and their values, by creating a map that maps each symbol to its value using the word class.

1.4.5 Printer

The printer is responsible for printing the output generated by the main. It uses both word and line to do this.

1.4.6 Word

Word is used to keep track of memory values, and to convert between hexadecimal, binary, and decimal.

1.5 Coding Conventions

The following is a list of coding conventions to follow while maintaining this project.

- Use comments to ensure that what you are doing is understandable.

- Specifications are stored as low in the dependency tree as possible.
- All interface .h files start with an i.
- Do not use namespaces in header files, as they will apply to oter components as well, which could cause problems.
- The first letter in each word in a method name is capitalized. All other letters in the name are lowercase
- All constants names are all-caps.
- The header files define the classes and functions, the .cpp files execute them.
- Private functions begin with a '_' character.
- Always use braces while using if, while, and for commands, even if they are not needed for what you were using it for.

1.6 Dependency Diagrams

In the pages to come, there are several dependancy diagrams that display the interactions of files and components. This section serves as a key for use in understanding their conventions.

- A **white** box indicates a class. A **marker** in the lower right corner of the box indicates that the class has base classes that are hidden. If the box has a **dashed** border this indicates virtual inheritance.
- A **solid** arrow indicates public inheritance.
- A **dashed** arrow indicates protected inheritance.
- A **dotted** arrow indicates private inheritance.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 3

Class Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

Namespace Documentation

6.1 Codes Namespace Reference

Values corresponding to the results of Wi-11 function calls.

Classes

- struct RESULT

  Holds error-reporting information.

Enumerations

- enum **ERROR** {
  ERROR_0, **SUCCESS**, **INV_LBL**, **LBL_WO_INST**,
  INV_INST, **STRZ_NOT_STR**, **END_OF_STR**, **STR_JUNK**,
  ARG_SIZE, **EMPTY_ARG**, **INV_REG**, **INV_CONST**,
  INV_ARG, **INV_HEX**, **INV_DEC**, **INV_BR**,
  NON_LD_LIT, **ORIG**, **ORIG2**, **ORIG_LBL**,
  ORIG_HEX, **REQ_LABEL**, **LBL_NOT_FOUND**, **REDEF_LBL**,
  MAX_S_SIZE, **MAX_L_SIZE**, **MAX_LENGTH**, **ABS_REL**,
  INV_IMM, **INV_IDX**, **PG_ERR**, **NO_END**,
  END_OB, **UNEXP_EOF**, **REL_PG_SIZE**, **MEM_FIT**,
  FILE_NOT_FOUND, **FILE_NOT_OPENED** }

6.1.1 Detailed Description

Values corresponding to the results of Wi-11 function calls. An enum is used for efficiency. The code can be returned up the collaboration hierarchy quickly so that, if necessary, the program can print an appropriate error message

Note

ResultDecoder can be used to do a look-up of the error message.

Chapter 7

Class Documentation

7.1 Codes::RESULT Struct Reference

Holds error-reporting information.

Public Member Functions

- **RESULT** (ERROR err, std::string inf="")

Public Attributes

- std::string **info**
- ERROR **msg**

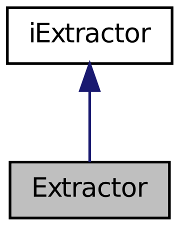
7.1.1 Detailed Description

Holds error-reporting information.

7.2 Extractor Class Reference

Implements the iExtractor interface.

Collaboration diagram for Extractor:



Public Member Functions

- Extractor (int size=SYMBOL_TABLE_MAX_SIZE)

  Gets the default max number of symbols, defaults on 1000.

- ∼Extractor ()

  Closes the input file, if necessary.

- bool **Open** (std::string filename)
- Codes::RESULT **GetSymbols** (SymbolTable &symbols)
- Word **GetLength** () const

Private Member Functions

- std::string _LineNumber (int pos)

  Creates a string "Line n", with n = pos.

Private Attributes

- std::ifstream _fileStream

  The input file.

- int _length

  Keep length after SymbolTable is filled.

- int _max_size

  The maximum size of the symbol table.

7.2.1 Detailed Description

Implements the iExtractor interface. This implementation is very nearly trivial. In this file the default maximum symbol table size is declared and store the run-time value of the max will be stored in the Extractor object. The private variable keeping track of the object file length is an int to facilitate the reporting of a file whose length has exceeded the upper boundary of memory.

7.2.2 Member Function Documentation

7.2.2.1 string Extractor::_LineNumber (int *pos*) `[private]`

Creates a string "Line n", with n = pos.

Parameters

`in`	*pos*	The line number.

Returns

A string as described above.

7.3 iExtractor Class Reference

Extracts symbols from a source file to be used on a second read.

Public Member Functions

- virtual bool Open (std::string filename)=0

  Opens the input file to be read.

- virtual Codes::RESULT GetSymbols (SymbolTable &symbols)=0

  Extracts the symbols and literals from the source file.

- virtual Word GetLength () const =0

7.3.1 Detailed Description

Extracts symbols from a source file to be used on a second read. This class provides a simple interface to fill a SymbolTable object and retrieve the size in memory this program would require.

7.3.2 Member Function Documentation

7.3.2.1 virtual bool iExtractor::Open (std::string *filename*) `[pure virtual]`

Opens the input file to be read.

Parameters

`in`	*filename*	The source file.

Returns

True iff the file was successfully opened.

7.3.2.2 virtual Codes::RESULT iExtractor::GetSymbols (SymbolTable & *symbols*) `[pure virtual]`

Extracts the symbols and literals from the source file.

Parameters

in:out]	symbols The object into which the symbols and literals will be stored.

Returns

SUCCESS iff the file syntax is sound, otherwise an appropriate error code.

This opeeration acts as the first of the two passes on the source file. Everything that can be checked on this first pass will, all other possible errors will be left until the second pass.

7.3.2.3 virtual Word iExtractor::GetLength ( ) const `[pure virtual]`

Precondition

GetSymbols() has already been successfully run.

Returns

The size of the space in memory needed to hold the source file.

7.4 iLine Class Reference

Stores information about a Line of Wi-11 assembly code.

Public Member Functions

- virtual Codes::RESULT ReadLine (std::string line)=0

    Parses and tokenizes a string.

- virtual std::string Label () const =0
- virtual int Literal () const =0
- virtual std::string Instruction () const =0
- virtual std::string operator[ ] (int index) const =0

    Provides access to the arguments of the instruction.

- virtual int Size () const =0
- virtual std::string ToString () const =0
- virtual bool HasLabel () const =0
- virtual bool IsPseudoOp () const =0
- virtual bool HasLiteral () const =0
- virtual bool IsComment () const =0

7.4.1 Detailed Description

Stores information about a Line of Wi-11 assembly code. This class defines an interface for pulling relevent data from a line of assembly code without having to handle whitespace or test for various syntactic properties.

7.4.2 Member Function Documentation

7.4.2.1 virtual Codes::RESULT iLine::ReadLine (std::string *line*) `[pure virtual]`

Parses and tokenizes a string.

Parameters

`in`	*line*	The line to be parsed.

Returns

SUCCESS iff the line is valid wi-11 assembly code, otherwise an appropriate error code.

7.4.2.2 virtual std::string iLine::Label ( ) const `[pure virtual]`

Precondition

HasLabel() returns true.

Returns

The label found at the beginning of the line.

7.4.2.3 virtual int iLine::Literal ( ) const `[pure virtual]`

Precondition

HasLiteral() returns true.

Returns

The literal found in the line.

7.4.2.4 virtual std::string iLine::Instruction ( ) const `[pure virtual]`

Returns

The instruction found in the line.

7.4.2.5 virtual std::string iLine::operator[] (int *index*) const `[pure virtual]`

Provides access to the arguments of the instruction.

Parameters

`in`	*index*	The index of the argument desired, starting at 0 for the first one.

Returns

Argument number "index".

7.4.2.6 virtual int iLine::Size ( ) const `[pure virtual]`

Returns

The number of arguments to the instruction.

7.4.2.7 virtual std::string iLine::ToString ( ) const `[pure virtual]`

Returns

The line of code as found.

7.4.2.8 virtual bool iLine::HasLabel ( ) const `[pure virtual]`

Returns

True iff a label was found in the line.

7.4.2.9 virtual bool iLine::IsPseudoOp ( ) const `[pure virtual]`

Returns

True iff the instruction is a pseudo-op (.ORIG, .END, .EQU, etc.).

7.4.2.10 virtual bool iLine::HasLiteral ( ) const `[pure virtual]`

Returns

True iff a literal was found in the line.

7.4.2.11 virtual bool iLine::IsComment ( ) const `[pure virtual]`

Returns

True iff he line only contained a comment or whitespace.

7.5 iPrinter Class Reference

Writes an object file and prints a listing to standard out.

Public Member Functions

- virtual Codes::RESULT Open (std::string infile, std::string outname)=0
  Opens the input and output files for reading and writing.
- virtual Codes::RESULT Print (SymbolTable &symbols, Word &file_length)=0
  Reads from the source file, writes an object file, and prints a listing.

7.5.1 Detailed Description

Writes an object file and prints a listing to standard out. This class defines a very simple interface by which the second pass of the two-pass algorithm is completed and the user is presented with useful information about the encoding process as well as an object file if the code is entirely valid.

7.5.2 Member Function Documentation

7.5.2.1 virtual Codes::RESULT iPrinter::Open (std::string *infile*, std::string *outname*) `[pure virtual]`

Opens the input and output files for reading and writing.

Parameters

`in`	*infile*	The name of the input file to be opened.
`in`	*outfile*	The name of the output file to be opened.

Returns

7.5.2.2 virtual Codes::RESULT iPrinter::Print (SymbolTable & *symbols*, Word & *file_length*) `[pure virtual]`

Reads from the source file, writes an object file, and prints a listing.

Parameters

`in`	*symbols*	A SymbolTable produced from a previous read of the input file.
`in`	*file_length*	The size the program should occupy in memory.

Returns

SUCCESS iff the object file could be created; otherwise an appropriate error message.

7.6 iSymbolTable Class Reference

Stores symbols and literals extracted from a source file.

Public Member Functions

- virtual void InsertLabel (std::string label, Word addr, bool relocate=false)=0

  Add a label to the table.

- virtual void InsertLiteral (int value, Word addr)=0

  Add a literal to the table.

- virtual bool Contains (std::string symbol) const =0
- virtual Word GetLabelAddr (std::string symbol) const =0

  Look up the address for a symbol.

- virtual Word GetLiteralAddr (int value) const =0

  Look up the address for a literal.

- virtual bool IsRelocatable (std::string label) const =0
- virtual const std::map< int, Word > * GetLiterals () const =0

  Aids the Printer in outputing literals.

7.6.1 Detailed Description

Stores symbols and literals extracted from a source file. This class defines an interface for storing various mappings vital to the two-pass algorithm used in this assembler.

7.6.2 Member Function Documentation

7.6.2.1 virtual void iSymbolTable::InsertLabel (std::string *label*, Word *addr*, bool *relocate* = `false`) `[pure virtual]`

Add a label to the table.

Parameters

`in`	*label*	The label to the stored.
`in`	*addr*	The address or value to associate it with.
`in`	*relocate*	Whether or not the program is relocatable.

Precondition

Contains(label) returns false.

7.6.2.2 virtual void iSymbolTable::InsertLiteral (int *value*, Word *addr*) [pure virtual]

Add a literal to the table.

Parameters

in	*value*	The value to be stored.
in	*addr*	The address whether the literal will be stored.

7.6.2.3 virtual bool iSymbolTable::Contains (std::string *symbol*) const [pure virtual]

Parameters

in	*symbol*	The symbol to look for.

Returns

True iff "symbol" is in the table.

7.6.2.4 virtual Word iSymbolTable::GetLabelAddr (std::string *symbol*) const [pure virtual]

Look up the address for a symbol.

Parameters

in	*symbol*	The symbol to look up.

Precondition

Contains(symbol) returns true.

Returns

The address of the symbol.

7.6.2.5 virtual Word iSymbolTable::GetLiteralAddr (int *value*) const [pure virtual]

Look up the address for a literal.

Parameters

in	*value*	The literal value to look up.

Returns

The address of the literal.

A literal should always be present as it will be found where it was declared. Any errors must be assembler related, not user related.

7.6.2.6 virtual bool iSymbolTable::IsRelocatable (std::string *label*) const `[pure virtual]`

Parameters

`in`	*label*	The label to look up.

Precondition

Contains(label) returns true.

Returns

True iff the symbol is relocatable.

7.6.2.7 virtual const std::map<int, Word>* iSymbolTable::GetLiterals ( ) const `[pure virtual]`

Aids the Printer in outputing literals.

Returns

An iterator from the literal values to their addresses.

7.7 iWord Class Reference

Defines a "word" of data on the Wi-11 Machine.

Public Member Functions

- virtual int ToInt () const =0

  "To non-negative Integer"

- virtual int ToInt2Complement () const =0

  "To Integer as 2's Complement"

- virtual std::string ToStr () const =0

  "To String"

- virtual std::string ToHex () const =0

  "To Hexadecimal"

- virtual bool FromInt (int value)=0

  "From Integer"

- virtual bool FromStr (const std::string &str)=0

  "From String"

- virtual bool FromHex (const std::string &str)=0

  "From Hexadecimal"

- virtual Word Add (const iWord &w) const =0

  Adds two words.

- virtual Word operator+ (const iWord &w) const =0

  A standard addition operator.

- virtual Word Subtract (const iWord &w) const =0

  Subtracts two words.

- virtual Word operator- (const iWord &w) const =0

  A standard subtraction operator.

- virtual Word And (const iWord &w) const =0

  "And"s the bits of two words.

- virtual Word Or (const iWord &w) const =0

  "Or"s the bits of two words.

- virtual Word Not () const =0

  "Not"s the bits of a word.

- virtual void Copy (const iWord &w)=0

  Copies a word.

- virtual Word & operator= (const Word &w)=0

  A standard assignment operator.

- virtual iWord & operator++ ()=0

    A standard pre-increment operator.

- virtual iWord & operator++ (int)=0

    A standard post-increment operator.

- virtual bool operator[ ] (const int i) const =0

    An accessor to the 'i'th bit of the value.

- virtual void SetBit (const int i, bool)=0

    Sets the 'i'th bit of the value.

7.7.1 Detailed Description

Defines a "word" of data on the Wi-11 Machine. The methods present in this interface are meant to mimic the functionality of the Wi-11 machine, allowing for simplified execution of the instructions therein. As the size of a "word" depends on the architecture, classes implementing this interface should define the word length to be 16 bits in length.

7.7.2 Member Function Documentation

7.7.2.1 virtual int iWord::ToInt ( ) const `[pure virtual]`

"To non-negative Integer"

Postcondition

The value of the word is not changed.

Returns

The bits of the word interpreted as a positive integer value.

7.7.2.2 virtual int iWord::ToInt2Complement ( ) const `[pure virtual]`

"To Integer as 2's Complement"

Postcondition

The value of the word is not changed.

Returns

The bits of the word interpreted as a signed (2's complement) integer value.

7.7.2.3 virtual std::string iWord::ToStr ( ) const [pure virtual]

"To String"

Postcondition

The value of the word is not changed.

Returns

16 characters: each either a 1 or 0

Examples:

If the object holds a (2's comp.) value 4: "0000000000000100"
If the object holds a (2's comp.) value -1: "1111111111111111"

7.7.2.4 virtual std::string iWord::ToHex ( ) const [pure virtual]

"To Hexadecimal"

Postcondition

The value of the word is not changed.

Returns

"0x" + <4 characters in the range [0-9],[A-F]>

Examples:

If the object holds (2's comp.) value 8: "0x0008"
If the object holds (2's comp.) value -2: "0xFFFE"

7.7.2.5 virtual bool iWord::FromInt (int *value*) [pure virtual]

"From Integer"

Parameters

`in`	*value*	The value to be stored into the word.

Postcondition

"value" is not changed.

Returns

True if and only if "value" can be represented in 16 bits

When this function returns "False", the value of the word is unchanged.

Otherwise, the word now holds the value "value".

7.7.2.6 virtual bool iWord::FromStr (const std::string & *str*) `[pure virtual]`

"From String"

Parameters

`in`	*str*	A string of characters meant to represent a "word" to be stored.

Postcondition

"str" is not changed.

Returns

True if and only if "str" is well-formed (as defined in #toStr()).

When this function returns "False", the value of the word is unchanged.

Otherwise, the word now holds the value "str".

7.7.2.7 virtual bool iWord::FromHex (const std::string & *str*) `[pure virtual]`

"From Hexadecimal"

Parameters

`in`	*str*	A string of characters meant to represent a "word" to be stored.

Postcondition

"str" is not changed.

Returns

True if and only if "str" is well-formed (as defined in #toHex()).

When this function returns "False", the value of the word is unchanged.

Otherwise, the word now holds the value "str".

7.7.2.8 virtual Word iWord::Add (const iWord & *w*) const `[pure virtual]`

Adds two words.

Parameters

`in`	*w*	A word value to be added.

Postcondition

Both "w" and the calling object do not change.

Returns

A new "Word" object containing result of adding "w" and the calling object.

Note

The addition is carried out with no regard to logical overflow.

7.7.2.9 virtual Word iWord::operator+ (const iWord & *w*) const `[pure virtual]`

A standard addition operator.

Note

"result = p + w" is equivalent to "result = p.Add(w)".

7.7.2.10 virtual Word iWord::Subtract (const iWord & *w*) const `[pure virtual]`

Subtracts two words.

Parameters

`in`	*w*	A word value to be subtracted.

Postcondition

Both "w" and the calling object do not change.

Returns

A new "Word" object containing the result of subtracting "w" from the calling object.

Note

The subtraction is carried out with no regard for logical overflow.

7.7.2.11 virtual Word iWord::operator- (const iWord & *w*) const `[pure virtual]`

A standard subtraction operator.

Note

"result = p - w" is equivalent to "result = p.Subtract(w)".

7.7.2.12 virtual Word iWord::And (const iWord & *w*) const `[pure virtual]`

"And"s the bits of two words.

Parameters

`in`	*w*	A word value to be "and"ed.

Postcondition

Both "w" and the calling object do not change.

Returns

A new "Word" object containing the result of performing a bit-wise and on "w" and the calling object.

7.7.2.13 virtual Word iWord::Or (const iWord & *w*) const `[pure virtual]`

"Or"s the bits of two words.

Parameters

`in`	*w*	A word value to be "or"ed.

Postcondition

Both "w" and the calling object do not change.

Returns

A new "Word" object containing the result of performing a bit-wise or on "w" and the calling object.

7.7.2.14 virtual Word iWord::Not ( ) const `[pure virtual]`

"Not"s the bits of a word.

Postcondition

The calling object do not change.

Returns

A new "Word" object containing the result of performing a bit-wise not on the calling object.

7.7.2.15 virtual void iWord::Copy (const iWord & *w*) `[pure virtual]`

Copies a word.

Parameters

`out`	*w*	The value to be copied.

Postcondition

The caller equals that parameter.

Equivalent to the assignment "caller = parameter".

7.7.2.16 virtual Word& iWord::operator= (const Word & *w*) `[pure virtual]`

A standard assignment operator.

Parameters

`in`	*w*	The value to be copied.

Returns

A copy of the parameter.

The return value and parameter here must be declared as "Word"s as C++ does not work well with polymorphic assignment operators.

7.7.2.17 virtual iWord& iWord::operator++ ( ) `[pure virtual]`

A standard pre-increment operator.

Returns

A reference to itself.

The object increments its value BEFORE the execution of the current line.

7.7.2.18 virtual iWord& iWord::operator++ (int) [pure virtual]

A standard post-increment operator.

Returns

A reference to itself.

The object increments its value AFTER the execution of the current line.

7.7.2.19 virtual bool iWord::operator[] (const int *i*) const [pure virtual]

An accessor to the 'i'th bit of the value.

Parameters

in	*i*	The index of the bit in question.

Precondition

The index must be less than the size of a word, ie. 16.

Returns

True <=> 1, False <=> 0.

The number of the bits starts at zero and rises into the more significant bits.

Examples:

If the object holds a value of 4 (0...100 in binary): num[2] = 1.
If it holds a value of 1 (0...001 in binary): num[0] = 1.
If it holds a negative value (Starting with a 1 in 2's complement): num[15] = 1.

7.7.2.20 virtual void iWord::SetBit (const int *i*, bool) [pure virtual]

Sets the 'i'th bit of the value.

Parameters

`in`	*i*	The index of the bit in question.

Precondition

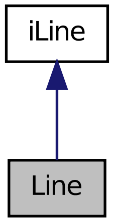
The index must be less than the size of a word, ie. 16.

Works in a similar way to ::operator[ ] but sets the bit instead of determining if it is set.

7.8 Line Class Reference

Implements the iLine interface.

Collaboration diagram for Line:



Public Member Functions

- Codes::RESULT **ReadLine** (std::string line)
- std::string **Label** () const
- std::string **Instruction** () const
- std::string **operator[ ]** (int index) const
- int **Size** () const
- int **Literal** () const
- std::string **ToString** () const
- bool **HasLabel** () const
- bool **IsPseudoOp** () const
- bool **HasLiteral** () const
- bool **IsComment** () const

Private Member Functions

- bool _IsWS (char ch) const

  Tests a character for whitespace.

- std::string _GetNext (std::string &str) const

  Get the next whitespace-free sub-string.

- Codes::RESULT _CheckArgs ()

  Tests the arguments extracted from the line for validity.

Private Attributes

- std::string _label

  Label, if any. Empty string otherwise.

- std::string _inst

  Instruction, if any. Empty string otherwise.

- std::vector< std::string > _args

  Holds each argument to the instruction, if any.

- std::string _code

  Holds a copy of the line of code.

- int _literal

  Holds a literal value, if any.

- bool _hasLabel

  True iff line contained a label.

- bool _hasLiteral

  True iff line contained a literal.

- bool _comment

  True iff line was a comment.

7.8.1 Detailed Description

Implements the iLine interface. This class store redundant amounts of information separated in different ways to allow the client to recreate and reformat the original text with ease.

7.8.2 Member Function Documentation

7.8.2.1 bool Line::_IsWS (char *ch*) const [private]

Tests a character for whitespace.

Parameters

in	*ch*	The character to be tested.

Returns

True iff ch is a space or a tab.

7.8.2.2 string Line::_GetNext (std::string & *str*) const [private]

Get the next whitespace-free sub-string.

Parameters

in	*str*	The string from which to obtain the substring.

Returns

The first whitespace-free substring of str.

If str contains only whitespace, the empty string is returned.

7.8.2.3 RESULT Line::_CheckArgs ( ) [private]

Tests the arguments extracted from the line for validity.

Returns

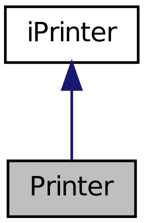
SUCCESS if the arugments are valid; an appropriate error otherwise.

Here, valid arguments are as defined in the Operands sections.

7.9 Printer Class Reference

Implements the iPrinter interface.

Collaboration diagram for Printer:



Public Member Functions

- ~Printer ()

  Closes the input and output files, if necessary.

- Codes::RESULT **Open** (std::string infile, std::string outfile)
- Codes::RESULT **Print** (SymbolTable &symbols, Word &file_length)

Private Member Functions

- void _SetBits (Word &w, int value, int index)

  Set 0 or more bits of a word.

- Codes::RESULT _IsReg (std::string reg)

  Tests a string as a valid register.

- int _RegNum (std::string reg)

  Get the index of register from a string.

- void _SetBits (std::string reg, Word &initial_mem, int &bit_offset)

  Sets the bits of initial_mem for registers.

- Word _ParseWord (const std::string &op, const SymbolTable &symbols)
- bool _Check9 (Word value, Word PC)

  Checks a Pgoffset9 value.

- bool _Check6 (Word value)

  Checks a 6-bit index.

- bool _Check5 (Word value)

  Checks a 5-bit immediate value.

- void _PreError (const std::string &line)

  Prints to the console to make error messages more readable.

- void _LineListing (const Word &current_address, const Word &value, const Line &current_line, const int &pos)

  Prints the listing entry for a single non-pseudo-op instruction.

- std::string _InFileData (const int line_number, const Line &current_line)

  Prints to the console to make listings more readable.

Private Attributes

- std::ifstream _inStream

  The input file.

- std::ofstream _outStream

  The output file.

7.9.1 Detailed Description

Implements the iPrinter interface. This implementation shows the complexity of a Printer component. Much of the redundancy inherent in the concept of this class is accounted for through the use of its many private functions. However, the method chosen for returning error messages makes it very difficult to truly optimize this code. Therefore, even much of the code specific to each type of operation is just a variant of another component. The handling of different instructions is done through a while loop containing a large if-elseif-elseif... statement. The components are separated by their argument pattern in an attempt to compactify the code as much as possible while still maintaining its correctness and readability.

7.9.2 Member Function Documentation

7.9.2.1 void Printer::_SetBits (Word & *w,* int *value,* int *index*) `[private]`

Set 0 or more bits of a word.

Parameters

	in:out]	w The word whose bits are to be set.
`in`	*value*	An integer corresponding to a bit mask.
`in`	*index*	The starting position of bit-setting.

Precondition

"index" is less than the size of the word.
"value" can be contained in |size of word| - "index" bits.

Note

This is essentially a localized bit-wise or.

Examples

Word=8, value=1, index=0 => Word=9.
Word=1F, value=14, index=11 => Word=E01F.

7.9.2.2 RESULT Printer::_IsReg (std::string *reg*) `[private]`

Tests a string as a valid register.

Parameters

`in`	*reg*	The string to be tested.

Returns

SUCCESS if reg is in the form RX where x is a number from 0 to 7. Otherwise, an appropriate error is returned.

7.9.2.3 int Printer::_RegNum (std::string *reg*) `[private]`

Get the index of register from a string.

Parameters

`in`	*reg*	The register string.

Precondition

reg is a valid register.

Returns

The index of register alluded to by reg

7.9.2.4 void Printer::_SetBits (std::string *reg*, Word & *initial_mem*, int & *bit_offset*) `[private]`

Sets the bits of initial_mem for registers.

Parameters

`in`	*reg*	The register string.
`in`	*initial_mem*	The Word object to be changed.
`in`	*bit_offset*	The location of the first bit to be set (leftmost being 15). Passed by reference for cascade effect.

Precondition

bit_offset > 2

Postcondition

initial_mem has bits [bit_offset:bit_offset-2] appropriately set

7.9.2.5 Word Printer::_ParseWord (const std::string & *op*, const SymbolTable & *symbols*) `[private]`

Get a 16 value from a string.

Parameters

`in`	*op*	The string to be parsed.
`in`	*symbols*	A table of symbols in case it is necessary.

Returns

The value contained in op, whether it is a constant or a symbol. If the symbol is not defined that will be handled outside of this function.

7.9.2.6 bool Printer::_Check9 (Word *value*, Word *PC*) [private]

Checks a Pgoffset9 value.

Parameters

in	*value*	The value to be checked.
in	*PC*	The value of the PC for value to be checked against.

Returns

True iff the highest order 7 bits of value and PC are the same.

7.9.2.7 bool Printer::_Check6 (Word *value*) [private]

Checks a 6-bit index.

Parameters

in	*value*	The value to be checked.

Returns

True iff value can be expressed in 6 bits.

7.9.2.8 bool Printer::_Check5 (Word *value*) [private]

Checks a 5-bit immediate value.

Parameters

in	*value*	The value to be checked.

Returns

True iff value can be expressed in 5 bits.

7.9.2.9 void Printer::_PreError (const std::string & *line*) [private]

Prints to the console to make error messages more readable.

Parameters

in	*line*	The line in which the error was found.

Shows the user the line with the error and formats the output to make error messages more readable.

7.9.2.10 void Printer::_LineListing (const Word & *current_address*, const Word & *value*, const Line & *current_line*, const int & *pos*) `[private]`

Prints the listing entry for a single non-pseudo-op instruction.

Parameters

`in`	*current_-address*	The address of the instruction.
`in`	*value*	The value to be store at current_address.
`in`	*current_line*	The line that has been translated.
`in`	*pos*	The current line number in the input file.

7.9.2.11 string Printer::_InFileData (const int *line_number*, const Line & *current_line*) `[private]`

Prints to the console to make listings more readable.

Parameters

`in`	*line_number*	The current line in the file.
`in`	*current_line*	The line that has been translated.

7.10 ResultDecoder Class Reference

Finds the messages associated with a given result code.

Public Member Functions

- ResultDecoder ()

  Generates the code-to-message mappings.

- std::string Find (const Codes::RESULT &result) const

  Looks up a result code.

Private Attributes

- std::map< Codes::ERROR, std::string > _codes

  Maps a result code to, in every case but SUCCESS, an error message.

7.10.1 Detailed Description

Finds the messages associated with a given result code.

7.10.2 Member Function Documentation

7.10.2.1 string ResultDecoder::Find (const Codes::RESULT & *result*) const

Looks up a result code.

Parameters

`in`	*result*	The result code to look up.

Returns

The messages associated with "result".

7.10.3 Member Data Documentation

7.10.3.1 std::map<Codes::ERROR, std::string> ResultDecoder::_codes `[private]`

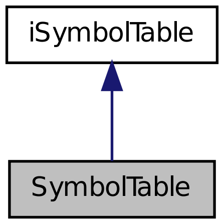
Maps a result code to, in every case but SUCCESS, an error message.

It is static because the result code messages should be available from anywhere.

7.11 SymbolTable Class Reference

Implements the iSymbolTable interface.

Collaboration diagram for SymbolTable:



Public Member Functions

- void **InsertLabel** (std::string label, Word addr, bool relocate=false)
- void **InsertLiteral** (int value, Word addr)
- bool **Contains** (std::string symbol) const
- Word **GetLabelAddr** (std::string symbol) const
- Word **GetLiteralAddr** (int value) const
- bool **IsRelocatable** (std::string label) const
- int **LabelCount** () const
- int **LiteralCount** () const
- const std::map< int, Word > ∗ **GetLiterals** () const

Private Attributes

- std::map< std::string, Word > _symbols

  Store symbol to address mappings.

- std::map< int, Word > _literals

  Store literal value to address mappings.

- std::map< std::string, bool > _relocatable

  Store relocation information by string.

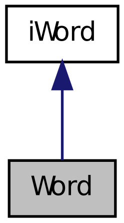
7.11.1 Detailed Description

Implements the iSymbolTable interface. This implementation keeps mappings of the different symbols and literal values to their address, as well as a mapping of symbol to its "relocatable" status. The purpose of this class is primarily to isolate the use of some of the more fragile code and provide the client with a tool rather than a list of components to pass into every function.

7.12 Word Class Reference

Implements iWord.

Collaboration diagram for Word:



Public Member Functions

- Word ()

  Sets a new Word's value to 0.

- Word (int i)

  Sets a new Word's value to "i".

- int **ToInt** () const
- int **ToInt2Complement** () const
- std::string **ToStr** () const
- std::string **ToHex** () const
- std::string ToHexAbbr () const

Same as ToHex() but with a different format.

- bool **FromInt** (int value)
- bool **FromStr** (const std::string &str)
- bool **FromHex** (const std::string &str)
- bool FromHexAbbr (const std::string &str)

  Same as FromHex() but with a different format.

- Word **Add** (const iWord &w) const
- Word **operator+** (const iWord &w) const
- Word **Subtract** (const iWord &w) const
- Word **operator-** (const iWord &w) const
- Word **And** (const iWord &w) const
- Word **Or** (const iWord &w) const
- Word **Not** () const
- void **Copy** (const iWord &w)
- Word & **operator=** (const Word &w)
- iWord & **operator++** ()
- iWord & **operator++** (int)
- bool **operator[ ]** (const int i) const
- void **SetBit** (const int, bool)

Static Public Attributes

- static const int **MAX_SIZE** = 0xFFFF

Private Member Functions

- bool _HasBit (int) const

  Tests for powers of two in binary representation.

Private Attributes

- unsigned short _value

  Used to store the "word" of data.

7.12.1 Detailed Description

Implements iWord.

7.12.2 Constructor & Destructor Documentation

7.12.2.1 Word::Word (int *i*)

Sets a new Word's value to "i".

Parameters

`in`	*i*	The value for the new Word to hold.

Precondition

"i" must fit within 16 bits.

7.12.3 Member Function Documentation

7.12.3.1 bool Word::_HasBit (int *i*) const `[private]`

Tests for powers of two in binary representation.

Parameters

i	The index of the digit desired from the binary representation of _word.

Returns

True if and only if the 'i'th bit is 1.

The indexing of the bits works as defined in #operator[ ]().

7.12.3.2 string Word::ToHexAbbr ( ) const

Same as ToHex() but with a different format.

The format in question is the same as ToHex() but without leading the "0x" or leading zeros.

Examples

A value of 4 would give "4".
A value of 17 would give "11".
A vallue of -2 would give "FFFE".

7.12.3.3 bool Word::FromHexAbbr (const std::string & *str*)

Same as FromHex() but with a different format.

Precondition

The hex string cannot exceed the values a Word can hold.

The format this function uses is the same as ToHexAbbr().

7.12.4 Member Data Documentation

7.12.4.1 unsigned short **Word::_value** `[private]`

Used to store the "word" of data.

The type "unsigned short" was chosen because in c++, shorts are 16bits (the same size as our words) and having it unsigned allows for easy "reading" as a positive int or a 2's complement int.

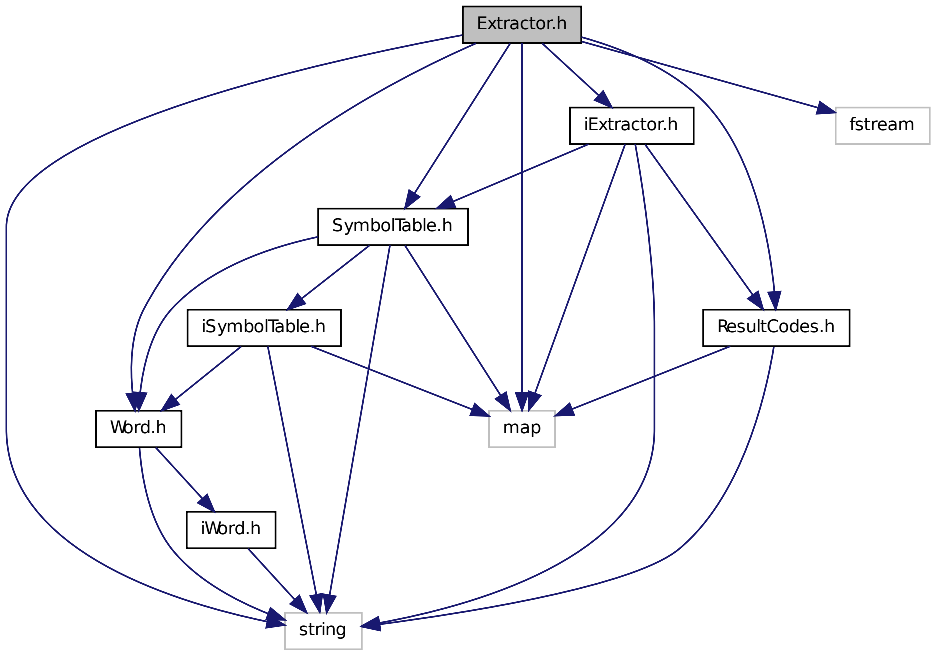
Chapter 8

File Documentation

8.1 Extractor.h File Reference

Definition of the private data for the "Extractor" class.

Include dependency graph for Extractor.h:



Classes

- class Extractor

  Implements the iExtractor interface.

Defines

- #define **SYMBOL_TABLE_MAX_SIZE** 1000

8.1.1 Detailed Description

Definition of the private data for the "Extractor" class.

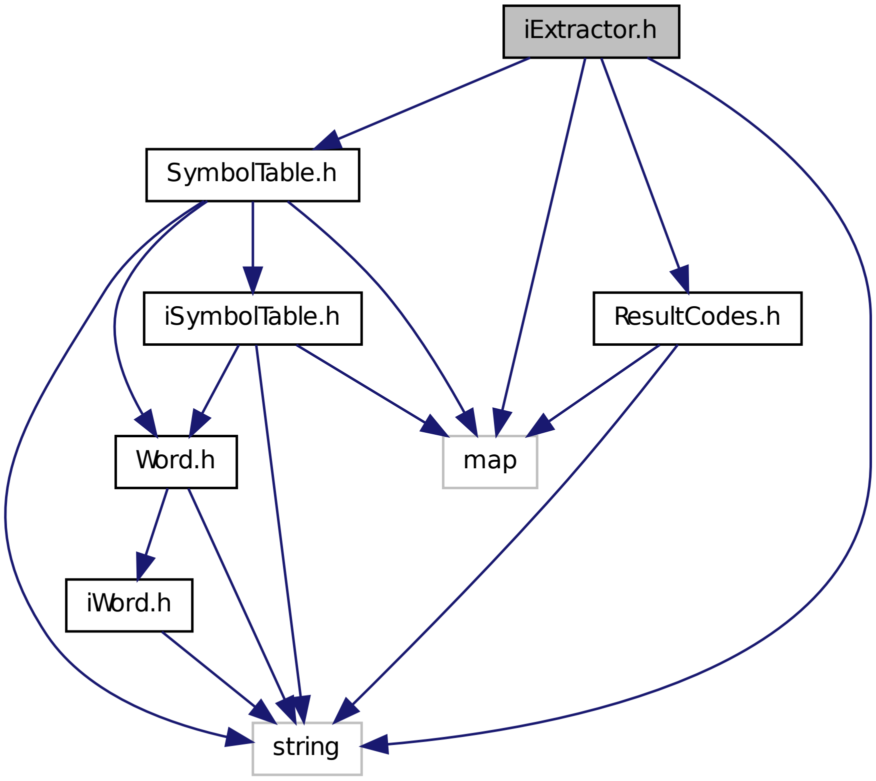
Author

Andrew Groot

8.2 iExtractor.h File Reference

Definition of the Extractor for the Wi-11 assembler.

Include dependency graph for iExtractor.h:



Classes

- class iExtractor

    Extracts symbols from a source file to be used on a second read.

8.2.1 Detailed Description

Definition of the Extractor for the Wi-11 assembler.

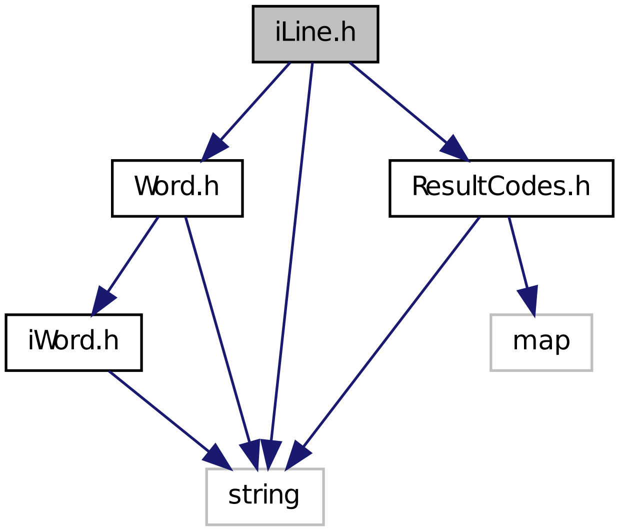
Author

Andrew Groot

8.3 iLine.h File Reference

Definition of a "Line" of Wi-11 assembly code.

Include dependency graph for iLine.h:



Classes

- class iLine

  Stores information about a Line of Wi-11 assembly code.

8.3.1 Detailed Description

Definition of a "Line" of Wi-11 assembly code.

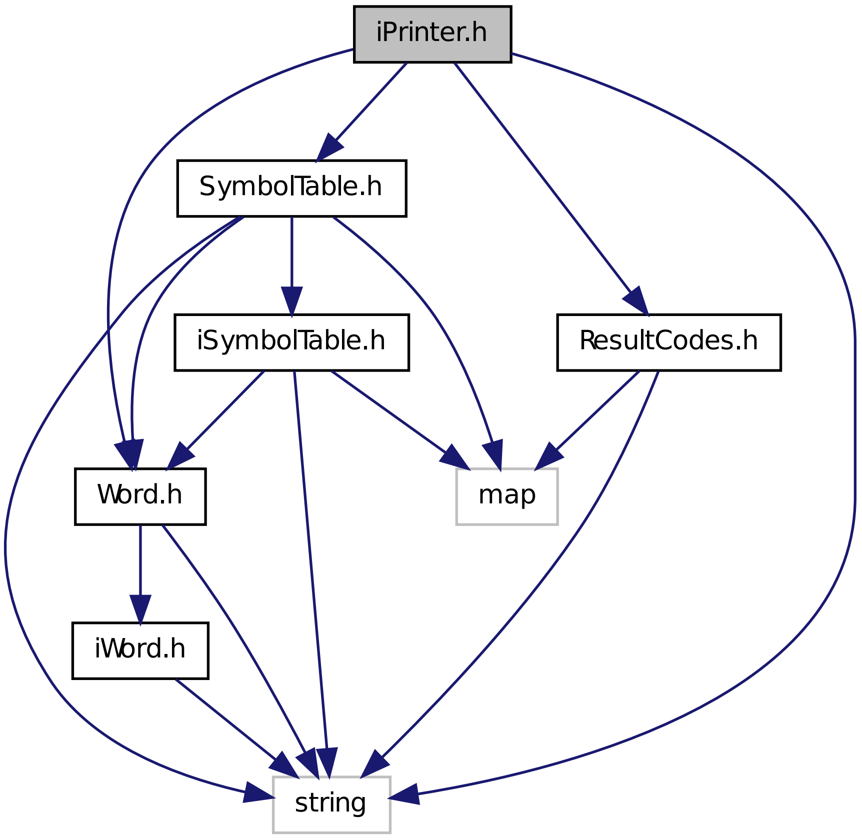
Author

Andrew Groot

8.4 iPrinter.h File Reference

Definition of the output of the Wi-11 assembler.

Include dependency graph for iPrinter.h:



Classes

- class iPrinter

  Writes an object file and prints a listing to standard out.

8.4.1 Detailed Description

Definition of the output of the Wi-11 assembler.

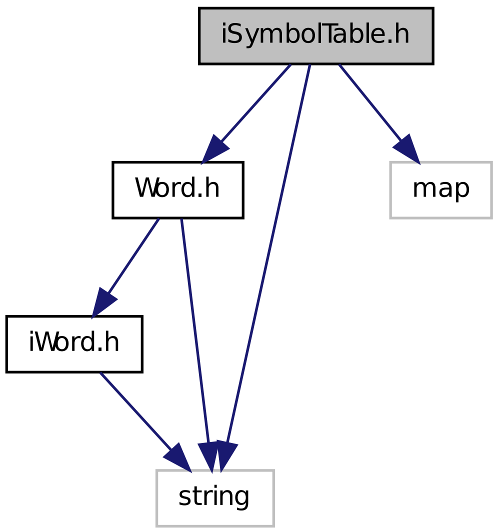
Author

Andrew Groot

8.5 iSymbolTable.h File Reference

Definition of the symbol table for the Wi-11 assembler.

Include dependency graph for iSymbolTable.h:



Classes

- class iSymbolTable

  Stores symbols and literals extracted from a source file.

8.5.1 Detailed Description

Definition of the symbol table for the Wi-11 assembler.

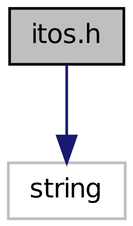
Author

Andrew Groot

8.6 itos.h File Reference

Utility function: int to std::string.

Include dependency graph for itos.h:



Functions

- std::string **itos** (long int number)

8.6.1 Detailed Description

Utility function: int to std::string.

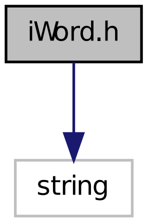
Author

Joshua Green

8.7 iWord.h File Reference

Definition of a "word" of data.

Include dependency graph for iWord.h:



Classes

- class iWord

  Defines a "word" of data on the Wi-11 Machine.

8.7.1 Detailed Description

Definition of a "word" of data.

Author

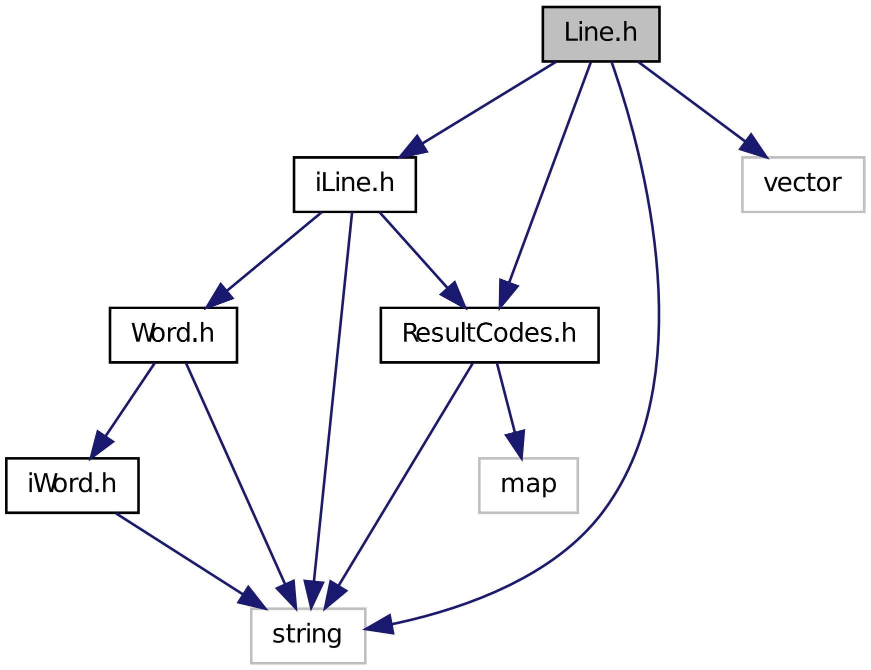
Joshua Green
Andrew Groot

Defines the operations and signatures by which a "word" class should operate. The signatures, while intended to be coded to the interface, are done as to this as C++ allows.

8.8 Line.h File Reference

Definition of the private data for the "Line" class.

Include dependency graph for Line.h:



Classes

- class Line

  Implements the iLine interface.

8.8.1 Detailed Description

Definition of the private data for the "Line" class.

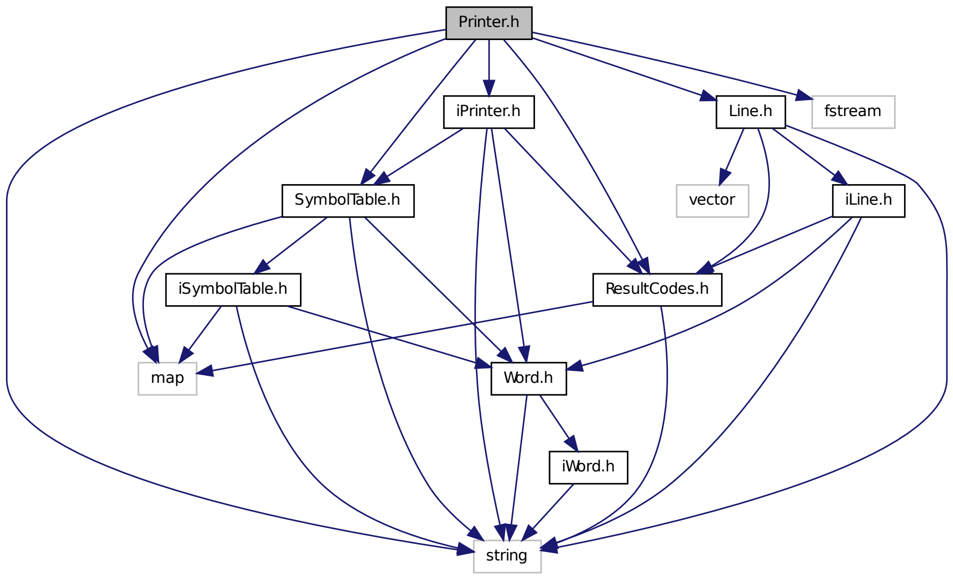
Author

Andrew Groot

8.9 Printer.h File Reference

Definition of the private data for the "Printer" class.

Include dependency graph for Printer.h:



Classes

- class Printer

  Implements the iPrinter interface.

8.9.1 Detailed Description

Definition of the private data for the "Printer" class.

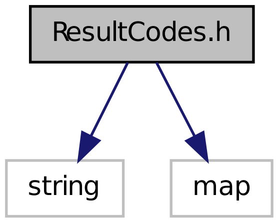
Author

Andrew Groot

8.10 ResultCodes.h File Reference

Definition of the Wi-11 assemblers error messages.

Include dependency graph for ResultCodes.h:



Classes

- struct Codes::RESULT

  Holds error-reporting information.

- class ResultDecoder

  Finds the messages associated with a given result code.

Namespaces

- namespace Codes

  Values corresponding to the results of Wi-11 function calls.

Enumerations

- enum **ERROR** {
  ERROR_0, **SUCCESS**, **INV_LBL**, **LBL_WO_INST**,
  INV_INST, **STRZ_NOT_STR**, **END_OF_STR**, **STR_JUNK**,
  ARG_SIZE, **EMPTY_ARG**, **INV_REG**, **INV_CONST**,

INV_ARG, **INV_HEX**, **INV_DEC**, **INV_BR**,

NON_LD_LIT, **ORIG**, **ORIG2**, **ORIG_LBL**,

ORIG_HEX, **REQ_LABEL**, **LBL_NOT_FOUND**, **REDEF_LBL**,

MAX_S_SIZE, **MAX_L_SIZE**, **MAX_LENGTH**, **ABS_REL**,

INV_IMM, **INV_IDX**, **PG_ERR**, **NO_END**,

END_OB, **UNEXP_EOF**, **REL_PG_SIZE**, **MEM_FIT**,

FILE_NOT_FOUND, **FILE_NOT_OPENED** }

8.10.1 Detailed Description

Definition of the Wi-11 assemblers error messages.

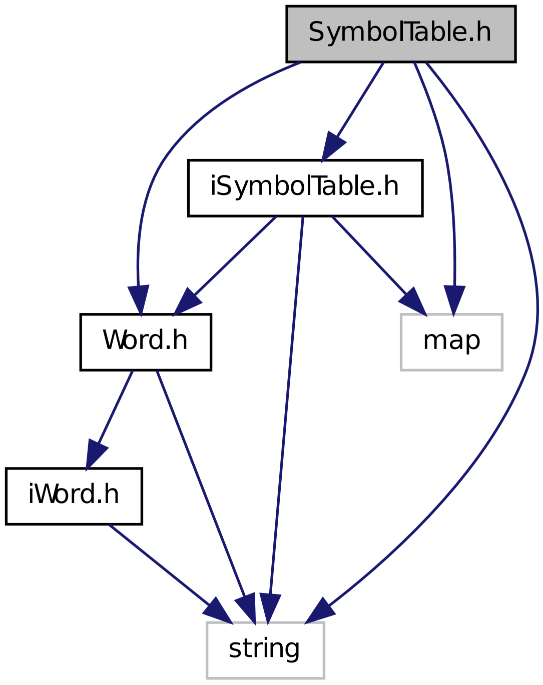
Author

Joshua Green
Andrew Groot

8.11 SymbolTable.h File Reference

Definition of the private data for the "SymbolTable" class.

Include dependency graph for SymbolTable.h:



Classes

- class SymbolTable

    Implements the iSymbolTable interface.

8.11.1 Detailed Description

Definition of the private data for the "SymbolTable" class.

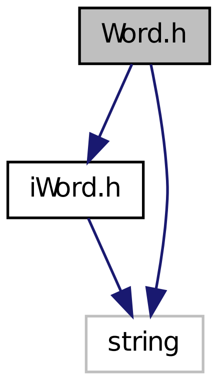
Author

Andrew Groot

8.12 Word.h File Reference

Definition of private data for the "Word" class.

Include dependency graph for Word.h:



Classes

- class Word

  Implements iWord.

Defines

- #define **WORD_SIZE** 16

8.12.1 Detailed Description

Definition of private data for the "Word" class.

Author

Joshua Green
Andrew Groot

Index