|  |
| --- |
|  |
| User’s Guide |
| Lab 2 Group BEERZ |
|  |
| **Elliot Schumacher** |
| **Brad Kline, Zach Smith, Evan Todd, and Ryan Powers** |
| **2/9/2011** |

|  |
| --- |
|  |

Contents

[Introduction 3](#_Toc286187881)

[Installing the Software 3](#_Toc286187882)

[Writing a Program 3](#_Toc286187883)

[Required Software 3](#_Toc286187884)

[Structure 3](#_Toc286187885)

[Header Record 3](#_Toc286187886)

[Relocatable Programs 4](#_Toc286187887)

[Text Records 4](#_Toc286187888)

[Label 4](#_Toc286187889)

[Instructions 4](#_Toc286187890)

[Operands 11](#_Toc286187891)

[Pseudo Operations 13](#_Toc286187892)

[Comments 14](#_Toc286187893)

[End Record 15](#_Toc286187894)

[Examples 15](#_Toc286187895)

[Compiling a Program 16](#_Toc286187896)

[Commands 16](#_Toc286187897)

[Output 17](#_Toc286187898)

[Object Data 17](#_Toc286187899)

[Listing Data 18](#_Toc286187900)

[Debugging Error Messages 20](#_Toc286187901)

# Introduction

This user guide outlines the abilities and requirements of the WI11 assembler. This program takes in a list of assembly files, which all must follow the requirements listed below, and outputs a list of object files, all of which largely follow the requirements of the object files for the WI11 simulator (see the output section for exceptions). This user guide assumes that the reader has general knowledge about assembly language, and also understands the object file format that is produced.

# Installing the Software

The WI11 Assembler can run on any platform that supports Java Runtime Environment. The assembler does not require much disk space; it takes up approximately 2 MB of space, and minimal memory.

After unzipping the program in the desired directory, the program must be compiled. If the directory path is where the program was unzipped, the instruction would be as follows;

> javac Common/\*.java Assembler/\*.java

Equivalently:

> build-asm

To run the program, see the section on compiling a program.

# Writing a Program

## Required Software

In addition to the installation of the assembler (discussed in the section above), you can use any basic word processing software such as Emacs, Notepad, or Notepad++, to write an assembler program. When saving, the file must have the extension \*.asm, marking an assembly file.

## Structure

To write a program for this assembler, you must format your file as follows. If you chose to omit a Label or the operation code does not take up all the spaces allotted below, leave those spaces blank. The numbers in the second row indicates the position of a character in a record. There can be white space between the individual arguments in a record, but otherwise the format needs to strictly be followed.

The beginning of a comment is noted with a semicolon (;). Anything after a semicolon is ignored by the Assembler. Please see the

## Header Record

A header record must be the first record in your program, although you may have multiple lines of comments before the header record. You must have a label for the header record; this serves as your segment name. After three blank spaces, you should have the pseudo-operation code for the beginning of a program, .ORIG. The operand for this record is optional. If you wish to specify an absolute address at which the program should be loaded from, this number needs to be in hexadecimal, and needs to be in the range of x0 to xFFFF. If the operand is omitted, the program is considered to be relocatable (see the section below on relocatable programs for further details).

### Relocatable Programs

A program is considered relocatable if the programmer does not specify a location in memory where the program should be loaded. This allows more flexibility in allotting a properly sized block in memory for the program; this is done by the program and not by the user.

## Text Records

Text records are the commands you wish the program to execute. The various components are described in further detail below.

There can be no more than 200 records in any assembly program.

### Label

Labels can be used to refer to a specific record, memory location, or value in your program. You can have a label on any record, except for the end record. A label can be composed of up to 6 alphanumeric characters (no spaces), but cannot begin with “R” (upper case) or “x” (lower case). Both upper case and lower case characters can be used in labels. Additionally, the labels are case sensitive.

There can be no more than 100 labels in any assembly program.

### Instructions

There are three categories of instructions, described in further detail below; Arithmetic, Loading and Storing, and Flow of Control instructions.

The table below gives a quick overview of the instructions available. The stuff between the brackets is to be replaced by actual values. Additionally, all values that are numbers can be replaced by symbols whose values are within the ranges specified by the requirements of the instructions.

|  |  |  |
| --- | --- | --- |
| Instruction | Operand(s) | Example |
| ADD | [destination register], [source register 1], [source register 2] **or**  [destination register], [source register 1], [immediate] | R1, R1, R3 **or**  R1, R1, x45 |
| AND | [destination register], [source register 1], [source register 2] **or**  [destination register], [source register 1], [immediate] | R4, R3, R4 **or**  R4, R3, #0 |
| BR[x] | [address] | x4509 |
| DBUG |  |  |
| JSR | [address] | x3000 |
| JMP | [address] | x2000 |
| JSRR | [base register], [index] | R1, #12 |
| JMPR | [base register], [index] | R5, x43 |
| LD | [destination register], [address] | R1, x3002 |
| LDI | [destination register], [address] | R1, x200 |
| LDR | [destination register], [base register], [index] | R1, R4, x23 |
| LEA | [destination register], [address] | R1, x4000 |
| NOT | [destination register], [source register] | R1, R4 |
| RET |  |  |
| ST | [source register], [address] | R5, x204F |
| STI | [source register], [address] | R3, x10FF |
| STR | [source register], [base register], [index] | R1, R4, x45 |
| TRAP | [trap vector] | x23 |

#### Arithmetic Instructions

There are three Arithmetic instructions available in the assembler; ADD, AND, and NOT. All of these instructions modify the CCRs, which are used for Flow of Control operations. The CCR is modified according to whether the result of the operation is positive, negative, or zero.

##### Add

The ADD instruction performs the addition arithmetic operation on two specified arguments, and stores the result in a specified register. The instruction can either add two registers together, or add a register and an immediate operand (for further information on immediate operands, please see the Operands section). They would be formed as follows;

ADD DR, SR1, SR2 ;Adds two registers together.

ADD DR,SR1,imm5;Adds a register with a immediate operand.

DR represents the location where the result of the operation is stored; SR1 and SR2 represent the registers where the arguments are stored, and imm5 represents an immediate operand.

Example: To add registers 1 and 2 together, and store the result in register 4, the record should be

ADD R4, R1, R2 ;R1+R2=R4

Example: To add register 1 with xA0, and store the result in register 1, the record would be

ADD R1,R1,xA0 ;R1+xA0=R1

##### And

The AND instruction performs a bitwise and operation on two specified arguments, and stores the result in the specified register. The instruction can either and two registers, or and a register with an immediate operand (see Operands section for further information). The instructions would be formed as follows;

AND DR,SR1,SR2 ;Ands two registers together

AND DR,SR1,imm5;Ands a register with an immediate.

DR represents the location where the result of the operation is stored; SR1 and SR2 represent the registers where the arguments are stored, and imm5 represents an immediate operand.

Example: To and registers 4 and 5 together, and store the result in register 0, the record would be

AND R0, R4, R5 ;R4 and R5 = R0

Example: To and register 1 with x0, and store the result in register 1, the record would be

AND R1,R1,x0;R1 and x0 = R1

##### Not

The NOT instruction performs a bitwise not operation on a register, then stores the result in a specified register. The instructions would be formed as follows;

NOT DR,SR

DR represents the location where the result of the operation is to be stored, and SR represents the register where the argument should be stored.

Example: To not register 1, and store the result in register 6, the record would be

NOT R6,R1 ;~R1 = R6

#### Loading and Storing Instructions

There are seven load and store instructions available for use. Four instructions, LD, LDR, LDI, and LEA load data from memory and store it in a given register. Three instructions, ST, STR, and STI, write data from the given register to a given memory location. Only instructions that load data from memory change the CCRs; they are changed depending on whether the data loaded is negative, positive, or zero. CCRs are used for Flow of Control instructions; please see the section on Flow of Control for further information.

The instructions are categorized by their addressing mode; the differences between these modes are explained in their respective sections.

##### Immediate Addressing

There is one Immediate Addressing operation, Load Effective Address (instruction code LEA). This instruction concatenates bits 15 through 9 of the Program Counter with bits 8 through 0 specified in the instruction. The result is stored in the register specified by the programmer. The instruction is formed as follows;

LEA DR,addr

DR represents the destination register where the result is to be stored, and the addr represents the address to be concatenated with the Program Counter.

Example: If the Program Counter is x3000, to load address x31A0 (or concatenate pgoffset x1A0 with the Program Counter) in to register 4, the instruction would be as follows;

LEA R4,x1A0 ; Loads address x31A0 into register 4

Example: If the Program Counter is x4200, to concatenate the PC with address Addr1 (a label that represents value x7094), and store the result in register 3 , the instruction would be as follows.

LEA R3, Addr1 ; Loads address x4294 in to register 3

##### Direct Addressing

There are two Direct Addressing operations, Load (LD) and Store (ST). In Direct Addressing, the address where data is loaded or stored is specified in the instruction. The address is formed by concatenating bits 15 through 9 of the Program Counter with bits 8 through 0 of the instruction. In the Load instruction, data is loaded from the specified memory location and stored in the given register. In the Store instruction, data is stored in the specified memory location, the source being the specified register. The instruction format is as follows;

LD DR,addr

ST SR,addr

DR represents the destination register where the result is to be stored. SR represents the source register where the data to be stored is located. The addr field represents the address that should be used to form the address that data is to be loaded or stored.

Example: If the value of the Program Counter is x2130, to store the data in register 4 at address x20E0, the instruction would be as follows;

ST R4,xE0; Contents of R4 stored at address x20E0

Example: If the value of the Program Counter is x39, to load the data from address x10E into register 1 (label Value represents value x10E), the instruction would be as follows;

LD R1,Value; Contents of x10E are loaded to R1

##### Indirect Addressing

There are two Indirect Addressing operations, Load Indirect (LDI) and Store Indirect (STI). In Indirect Address mode, the address where data is loaded or stored is formed by loading the address from the memory location specified in the instruction. The memory location where the address is stored is formed by concatenating bits 15 through 9 for the Program Counter with bits 8 through 0 of the instruction. For the Load Indirect instruction, data is loaded from the address stored in the specified memory location, and is loaded into the given register. For the Store Indirect instruction, data is stored at the address stored in the given memory location, the source of the data being the specified register. The instruction format is as follows;

LDI DR,addr

STI SR,addr

DR represents the destination register where the result is to be stored. SR represents the source register where the data to be stored is located. The addr field represents the memory location where the address for the data to be loaded or stored is located.

Example: If the value of the Program Counter is x3000, to store the data in register 1 at the address stored in location x31FF, the instruction would be as follows;

STI R1,x1FF

Example: If the value of the Program Counter is x14, to load data from the address stored at location xA, and store it in register 3, the instruction would be formed as follows;

LDI R3,xA

##### Register Indexed Addressing

There are two Register-Indexed Addressing operations, Load Register-relative (LDR) and Store Register-relative (STR). In Register indexed addressing mode, the address where the data is stored or loaded is formed by adding a zero-extended six bit offset (index6) to a given base register. The instruction format is as follows;

LDR DR,BR,index6

STR SR,BR,index6

DR represents the destination register where the result is to be stored. SR represents the source register where the data to be stored is located. BR represents the base register that is used to form the store or load location. The index6 represents the integer to be added to the base to form the store or load location.

Example: To store the data in register 2 at the address formed by register 3 (as the base register) and by xFF, the instruction would be as follows;

STR R2,R3,xFF

Example: To load the data stored at the location formed by register 1 (as the base register) and by INDEX1 (a label representing x40) into register 2, the instruction would be as follows;

LDR R2,R1,INDEX1

#### Flow of Control Instructions

Flow of Control instructions do several things; allow the program to jump to a specific record in a program, to control input and output, and allow the program to use subroutines. The instructions are described in more detail below.

##### Branches

A branch is a conditional instruction that changes the program counter to the given address in the operand if the conditions specified are true. The address that the program branches to is formed by concatenating bits 15 through 9 of the program counter with bits 8 through 0 of the address in the operand.

The branch instruction depends on what the conditions are for the branch to occur. After “BR”, the CCR conditions should be listed in order of the CCRs; N (Negative), Z (Zero), and P (Positive) can all be added after BR, and must be added in that order. The CCRs of the machine are changed after any Arithmetic or Load instruction, depending on the result of the command.

Example: If the program counter is equal to x3100, to branch if the CCRs are set to either negative or positive to x3156, the instruction would be as follows;

BRNP x3156

Example: If the program counter is equal to x10, to branch unconditionally to x40, the instruction would be as follows;

BRNZP x40

The instruction x0000, which is a never branch, is also known as a no-op. This instruction does not execute anything, yet the program counter is incremented.

##### Traps

A Trap instruction controls several specific functions of the machine. The instructions for these are formed by the Trap instruction code (TRAP), followed by the trap vector for the function desired.

The table below is a quick summary of the traps available.

|  |  |  |
| --- | --- | --- |
| Name | Number | Description |
| IN | x23 | Prompts the user to enter a character, which is stored in register 0. |
| INN | x33 | Prompts the user to enter a decimal number, which is stored in register 0. |
| OUT | x21 | Writes character stored in register 0 to the console. |
| PUTS | x22 | Writes string whose first character is pointed to by register 0 to the console. |
| OUTN | x31 | Writes the decimal value of register 0 to the console. |
| HALT | x25 | Ends execution of the program. |
| RND | x43 | Generates a random number and stores it in register 0. |

A symbol that equals the trap vector integer can replace a constant trap vector.

###### Input

There are two forms of Input instructions; one prompts the user for a character (IN, x23), the other prompts the user for a decimal integer (INN, x33). The data input by the user is stored in register 0 (for a character, the ASCII code is stored in the register).

The CCRs are changed according to the data input by the user.

Example: To prompt the user for a character to be stored in register 0, the instruction would be as follows;

TRAP x33

###### Output

There are three forms of Output instructions; one outputs a character from register 0 (OUT, x21), another prints out a string whose starting point is the address stored in register 0 (PUTS, x22), and the last prints out a decimal integer stored in register 0 (OUTN, x31).

Example: To print out the integer stored in register0, the instruction would be as follows;

TRAP x31

Example: To print out the string “Test”, if the first character’s address is stored in register 0 (and the symbol PUTS equals x22), the instruction would be as follows;

PUTS .EQU x22

TRAP PUTS

###### Halt

This instruction halts the execution of the program (HALT, x25).

Example: To halt a program, the instruction would be as follows;

TRAP x25

###### Random

This instruction generates a random number, and then stores it in register 0 (RND, x43).

The CCRs are changed according to the number generated.

##### Jumps

There are two types of jumps; jumps to subroutines (JSR, JSRR) and regular jumps (JMP, JMPR). The difference between the two is that the jumps to subroutine instructions copy the address of the Program Counter into register 7. Otherwise, JSR and JMP are the same instruction, as are JSRR and JMPR.

In the JSR and JMP instruction, the destination address is formed by concatenating bits 15 through 9 of the Program Counter with bits 8 through 0 of the instruction. The instruction format is as follows;

JSR addr

JMP addr

The addr operand represents either the symbol or constant used as an addresss.

In the JSRR and JMPR instruction, the destination address is formed by by adding a zero-extended six bit offset (index6) to the given base register. The instruction format is as follows;

JSRR BR, index6

JMPR BR, index6

BR indicated the base register, which is added to index6, to calculate the desired destination address.

Example: To jump to the address formed by the addition of register 1 and x20, the instruction would be as follows;

JMPR R1, x20

##### Returns

The return instruction (RET) copies the contents of register 7 into the Program Counter. This allows a program to return from a subroutine call.

### Operands

There are several different forms of operands that can be used in the operand fields of records; Registers, Constants, Offsets, Immediates, Addresses, Indexes, Symbols, and Literals. To see which forms of operands can be used with a specific instruction, please refer to the section above discussing the requirements for each one.

#### Registers

In the machine, there are eight 16-bit registers. To refer to a register, use a capital ‘R’ followed by the number of the register. Thus, the range of registers possible is R0 to R7.

#### Constants

If an instruction can use constants in their operands, you can use either hexadecimal or decimal numbers. Hexadecimal numbers can be in the range of x0 and xFFFF, and must have a lowercase ‘x’ before the number. Decimal numbers can be in the range of -32,768 to 32,767, and must have pound sign ‘#’ before the negative sign (if present) and the numbers.

#### Immediates (imm5)

If an instruction can have Immediates in their operands, the Immediate must be in the range of -16 to 15 if in decimal, or in the range of x0 to x1F if hexadecimal. Decimal numbers must have the pound sign ‘#’ before the digits (and negative sign). Hexadecimal numbers must have a lowercase ‘x’ before the digits.

#### Addresses (addr)

If an instruction uses Addresses, they must be in the range of x0 to xFFFF (or 0 to 65535 in decimal). When used in an instruction, only the last nine bits (in binary) are used. Decimal numbers must have the pound sign ‘#’ before the digits (and negative sign). Hexadecimal numbers must have a lowercase ‘x’ before the digits.

If the address provided is not in the same memory page as the program counter, the assembler shows an error message.

#### Indexes (index6)

If an instruction uses Indexes, they can be in the range of x0 to xFF (or 0 to 255 in decimal). Decimal numbers must have the pound sign ‘#’ before the digits (and negative sign). Hexadecimal numbers must have a lowercase ‘x’ before the digits.

#### Symbols

Any of the above operands can be replaced by a symbol in a program. A relocatable symbol, or a symbol whose value changes depending on the memory location of the program, can be used only with Branch, Jump Subroutine, Jump, Load, Load Immediate, Load Effective Address, Store, and Store Immediate. Otherwise, an absolute symbol is required, whose value does not change based on the memory location of the program.

Symbols can be used in place of a register name. If done, the absolute symbol must be equal to a value between 0 and 7.

Example: To add the value of registers 1 and 2, and store the result in register 1. The symbol “avalue” represents register 1, and the symbol “bvalue” represents register 2.

avalue .EQU x1

bvalue .EQU x2

ADD avalue, bvalue, avalue ; avalue + bvalue = avalue

Symbols can also be used in place of Immediates, Indexes, and Trap Vectors. These must be absolute (most likely using the .EQU function), and must be in the range of the operand it is replacing. When a symbol is used as the last argument in the ADD or AND instructions, it is always assumed to be an Immediate operand, rather than a register.

There can be no more than 100 symbols in any program.

#### Literals

Literals can only be used with the Load (LD) instruction. If a literal is present, the assembler assigns a location in memory for the literal, then places the value indicated in the reserved memory, and finally uses that address in the instruction. To use a literal, place an equal sign ‘=’ in front of a constant. Thus, the value of a literal must be between -32,768 and 32,767 for decimal numbers, and between x0 and xFFFF for hexadecimal numbers.

There can be no more than 50 literals in any assembly program.

### Pseudo Operations

Pseudo Operations are instructions that either store a specified value (or values beginning) at a location, or configure storage. There are six pseudo operations (the instructions for the start and end of a program are discussed in a separate section).

The table below is a quick overview of the available pseudo operations.

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction | Example Operands | Label Necessary? | Description |
| .ORIG | x3000 | Yes | Tells the assembler where the program is to be stored (operand not necessary). |
| .END | x3001 | Not allowed | Denotes where the program should begin executing (operand not necessary). |
| .EQU | #56 | Yes | Equates the symbol with the value in the operand field. |
| .FILL | x30 | No | Fills the memory location of the program counter with the value in the operand field. |
| .STRZ | “Testdata” | No | Assigns a word of memory for each character in the string, followed by a null character. |
| .BLKW | x10 | No | Creates a block of words; the size is specified in the operand. |
| .EXT | Symbol, BEERZ | Not allowed | Defines a list of symbols that may be referenced by this segment, but are not defined in this segment. |
| .ENT | Symbol, BEERZ | Not allowed | Defines a list of symbols that must be defined in this segment, and may be referenced by other segments. |

#### .EQU

This instruction equates a symbol (given in the label field) with a value given in the operand field. This is analogous to a constant in other programming languages. The value can be a previously defined label or a constant. If a constant, it can be written as a decimal (with the pound sign (#) before it) or as a hexadecimal number (with a lower case ‘x’ before it). This method does not allocate memory.

If the value of the symbol is equated to another symbol, that symbol must be defined earlier in the program.

Example: To set ‘const1’ equal to the value #32, the instruction would be as follows;

const1 .EQU #32

Example: To set ‘const2’ equal to the value represented by ‘const1’, the instruction would be as follows;

const2 .EQU const1

#### .FILL

This instruction creates a word (of memory), that holds the operand specified by the user. The operand can be in either hexadecimal (with a lower case ‘x’ preceding the number) or in decimal (with a pound sign ‘#’ preceding the number). Hexadecimal numbers must be in the range of x0 to xFFFF, and decimals must be in the range of -32,768 and 32,767. Thus, if a symbol is used as the operand, the value of that would need to be in the range of -32,768 to xFFFF. You can give a .FILL record a label if you wish, to refer back to the specific location in memory where the word is stored. However, a label is not required.

Example: To define #-342, the instruction would be as follows;

.FILL #-342 ;Reserves memory location with contents -342

Example: To define x45, and label it with the string “hexnum”, the record would be as follows;

hexnum .FILL x45

#### .STRZ

This instruction creates a block of words (of memory) to hold a string of characters, which should be in the operand field enclosed in quotation marks. The last character is followed by a null word. Thus, the .STRZ function uses one plus the length of the string words in memory. A label is optional; if used, it refers to the memory location where the first character is stored.

Example: To store the string “Test1” in memory, and link the first character’s location with the label string, the record would be as follows;

string .STRZ “Test1”

#### .BLKW

This instruction creates a block of storage in memory. The operand of this instruction is the number of words to be set aside, in hexadecimal. The number of words needs to be between 0 and 32,767 (operand must be positive). In addition to using a constant in the operand field, a previously defined symbol can also be used; the data it contains must follow the previously mentioned guidelines for constants. This block does not contain any data after created. A label is optional; if used, it refers to the memory location of the first word in the block of memory.

Example: To create a block of x20 words, and label the first word “blkw12”, the instruction would be as follows;

blkw12 .BLKW x20

#### .EXT

This instruction creates a list of symbols that are external to this program segment. This means that these symbols may be referenced by this program, but are defined in a separate program. This essentially tells the assembler to accept the symbol(s) provided in the operand(s). The number of symbols in the operand does not have a limit, though each operand must be a symbol. There also may not be a label to this pseudo-op.

Example: To define the symbol BEERZ as an external symbol (not defined in this segment, but referenced here), the instruction would be as follows:

.EXT BEERZ

#### .ENT

This instruction defines a list of symbols that must be defined in this segment, and may be referenced in other segments. These serve as entry points into this segment. Like .EXT, there is no limit on the number of operands, but they must all be symbols. There also may not be a label for this pseudo-op.

Example: To define the symbol BEERZ as an entry point (can be referred to in other segments externally, and must be defined in this segment), the following instruction would be used:

.ENT BEERZ

### Comments

Comments can be formed in two ways; a full line comment or a partial line comment. If the first character of a line is a semicolon (;), then that record is considered to be a comment, and is ignored by the Assembler. Otherwise, a comment needs to follow a properly formed record, and should start after the operands field. Again, the comment should begin with a semicolon (;), and the assembler ignores any information after the semicolon in the same line.

Example: To insert a comment after a fill instruction, the instruction would be as follows;

exampl .FILL xFF; Any data after the semicolon will be ignored.

## End Record

The end record needs to be the last record in your program. The record should be formed with .END in the operation field. Optionally, a hex integer starting location operand may be specified. If none is given, then the program begins execution at the first address in the segment. If a hex integer between x0 and xFFFF is given, then the program begins to execute at that address.

Additionally, there cannot be any label in this record.

Example: To begin execution at address x300A, the End Record would be formed as follows;

.END x300A

## Examples

#### Relocatable Program

The following program is an example of a relocatable program.

Test5 .ORIG

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] <- x####

LEA R5,Array

LD R6,=#100 ;R6 <= #100

STR R0,R5,#1 ;M[Array+1] <= x####

TRAP x25

ACC .EQU #1

; ----- Scratch Space -----

Array .BLKW #3

.FILL x10

.END Begin

As seen in the output produced (shown in the output section), the assembler marks the records that need to be changed when a memory address is assigned to the program.

#### Non-Relocatable Program

The following program is an example of a program that is not relocatable.

Test1 .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

# Compiling a Program

## Commands

Before executing this command, please make sure that your current directory is where the file you wish to assemble is located.

The basic command for compiling an assembly program with this assembler is as follows.;

>java Main (inputfile) (outputfile) [options]

The input and output files are necessary, while any options specified are not necessary to run the program. The placeholders (inputfile) and (outputfile) should be replaced with the assembly program and output object file. The placeholder [options] shows where any additional flags would go; in this case, only the listing flag is available.

To view the possible flags for the program, omit two files and substitute “--help”.

To see a source code listing for the program, use the “-l” flag in the option field. This causes the listing to be displayed on the console screen.

Example: To compile file “test.asm” into an object file “test.obj”, and display a listing for the file, the command would be as follows;

>java Main test.asm test.obj -l

## Output

There are two types of output that the file produces; a listing and an object file. The listing file can only be viewed in the console, while the object data is written to a file.

### Object Data

The object data file is a format that a loader would read in to use to execute this program. The assembler compiles the assembly file input into it, and outputs the machine code for the program. The output differs depending on whether the program is relocatable or not. If the program is not relocatable, the object is in the standard format of the WI11 simulator. If it is relocatable, then the assembler places a marker after the records that need to be changed when the program is assigned a position in memory.

#### Examples

These examples are the object files produced by the two input file examples in that section.

##### Relocatable

This is the object file produced by from the assembly file in the Input Examples section;

HTest5 00000018

T00000004

T00012200M0

T0002e007M0

T0003f022

T0004127f

T00050203M0

T0006480cM0

T00070068

T00080069

T00090021

T000a0020

T000b0000

T000c5020

T000d9000

T000e3013M0

T000fea13M0

T00102c17M0

T00117141

T0012f025

T00160010

T00170064

E0001

As shown in the above file, the “M0” flag is placed after any record that needs to have a memory location defined in it.

##### Non-Relocatable

This is the object file produced from the assembly file shown in the Input Examples section;

HTest1 30b00018

T30b00004

T30b122b0

T30b2e0b7

T30b3f022

T30b4127f

T30b502b3

T30b648bc

T30b70068

T30b80069

T30b90021

T30ba0020

T30bb0000

T30bc5020

T30bd9000

T30be30c3

T30bfeac3

T30c02cc7

T30c17141

T30c2f025

T30c60010

T30c70064

E30b1

As shown in the above file, there are no extra markers after memory reference records, unlike the Relocatable example.

### Listing Data

When the assembler is passed the “-l” flag to produce a listing file for the program, it prints one out in the console. The listing file has several columns; the hexadecimal address and hexadecimal instruction, the label (if any) for the records, the instructions, the operands, and any comments.

#### Examples

These listings were produced from the two example assembly files shown in the input section.

##### Relocatable

Test5 .ORIG

00000004 count .FILL #4

00012200 Begin LD ACC,count ;R1 <- 4

0002e007 LEA R0,msg

0003f022 loop TRAP x22 ;print "hi! "

0004127f ADD ACC,ACC,#-1 ;R1--

00050203 BRP loop

0006480c JMP Next

00070068 msg .STRZ "hi! "

00080069

00090021

000a0020

000b0000

000c5020 Next AND R0,R0,x0 ;R0 <- 0

000d9000 NOT R0,R0 ;R0 <- xFFFF

000e3013 ST R0,Array ;M[Array] <- x####

000fea13 LEA R5,Array

00102c17 LD R6,=#100 ;R6 <= #100

00117141 STR R0,R5,#1 ;M[Array+1] <= x####

0012f025 TRAP x25

ACC .EQU #1

Array .BLKW #3

00160010 .FILL x10

.END Begin

00170064

Notice that several of the pseudo operations do not have a memory location; either they do not need one or one is generated at run time.

##### Non-Relocatable

Test1 .ORIG x30B0

30b00004 count .FILL #4

30b122b0 Begin LD ACC,count ;R1 <- 4

30b2e0b7 LEA R0,msg

30b3f022 loop TRAP x22 ;print "hi! "

30b4127f ADD ACC,ACC,#-1 ;R1--

30b502b3 BRP loop

30b648bc JMP Next

30b70068 msg .STRZ "hi! "

30b80069

30b90021

30ba0020

30bb0000

30bc5020 Next AND R0,R0,x0 ;R0 <- 0

30bd9000 NOT R0,R0 ;R0 <- xFFFF

30be30c3 ST R0,Array ;M[Array] <- xFFFF

30bfeac3 LEA R5,Array

30c02cc7 LD R6,=#100 ;R6 <= #100

30c17141 STR R0,R5,#1 ;M[Array+1] <= xFFFF

30c2f025 TRAP x25

ACC .EQU #1

Array .BLKW #3

30c60010 .FILL x10

.END Begin

30c70064

Notice that there are several memory locations that have no listed content in them after the .STRZ instruction. The memory location where the record is located only holds the first character of the string; any further characters and a null memory location follow that initial memory address.

# Debugging Error Messages

Note: the pound sign (#) represents a number that varies in different situations, and any word between brackets [] indicates the same.

The error messages are sorted alphabetically, with periods coming first.

##### “.BLKW requires a positive operand.”

**Cause**: The program specifies that there should be a negative number of words set aside in memory.

**Solution**: Verify that the value of the operand specified in the instruction is equal to a positive value.

##### “.BLKW requires one operand. *#* were given.”

**Cause:** There is more than one operand in the .BLKW instruction.

**Solution:** Remove all but one operand in this record. Also, make sure the one operand remaining is a symbol or constant.

##### ".ENT operand [operand] must be a valid symbol name."

Cause: A .ENT operation has an invalid operand.

Solution: Verify that all .ENT instructions have valid symbols as operands.

##### ".ENT requires at least one operand."

**Cause**: A .ENT instruction has no operands.

**Solution**: Ensure that all .ENT instructions have at least one operand. Recall that .ENT operands must be symbols.

##### ".EXT operand [operand] must be a valid symbol name."

Cause: A .EXT operation has an invalid operand.

Solution: Verify that all .EXT instructions have valid symbols as operands.

##### ".EXT requires at least one operand."

**Cause**: A .EXT instruction has no operands.

**Solution**: Ensure that all .EXT instructions have at least one operand. Recall that .EXT operands must be symbols.

##### “.EQU operand must be a symbol or a constant value.”

**Cause**: The operand of an .EQU instruction can only be a symbol or a constant value. No other values can be used.

**Solution**: Verify that a symbol or constant is in the operand field. For further information on this instruction, see the section on it in the Instruction section.

##### “.FILL requires one operand. *#* were given.”

**Cause**: You can only have one operand in a .FILL instruction. Any more than that is prohibited.

**Solution**: Remove all but one operand in the instruction. See the section on the .FILL pseudo-operation in the Instruction section for further information.

##### “.ORIG instruction must be first non-comment line.”

**Cause**: The first record (that is not a comment) in an assembly program must be the header record, containing the .ORIG instruction.

**Solution**: Place .ORIG instruction at the beginning of the file, or after comments. For further information on header records, see the section on those in the input section.

##### “.ORIG may have a maximum of one operand; *#* of operands were given.”

**Cause**: There is more than one operand in the .ORIG instruction.

**Solution**: The only operand permitted in the .ORIG instruction is a hexadecimal address if the programmer wishes to specify one. Any more operands generates an error.

##### “.STRZ requires one operand. *#* were given.”

**Cause**: More than one operand is given in the .STRZ instruction.

**Solution**: Remove all but one operand in the .STRZ instruction.

##### “Cannot alias a symbol to itself”

**Cause:** The program attempts to define a symbol as the value of the same symbol.

**Solution:** Change to operand of the symbol to a constant or another defined symbol.

##### “Cannot parse # as hex value”

**Cause:** There is a number specified in the program as a hexadecimal number that cannot be parsed as such.

**Solution:** Verify that the hexadecimal number in the record indicated is formatted correctly.

##### “Could not find definition for operation [operation] with matching operands.”

**Cause:** There was a unknown instruction used in the assembly file. This could mean that the instruction itself is not defined, or the instruction does not have the proper operands.

**Solution:** Verify that the record noted only uses standard instructions, and that the proper operands are listed as well.

##### “Detected string operand with unclosed quotation mark.”

**Cause:** The operand in the .STRZ instruction is missing a quotation mark.

**Solution:** Format the string in the operand properly by placing a quotation mark on both sides of the string.

##### “Failed to assemble program due to an IO error.”

**Cause**: This error is likely caused by the assembly file not being where it was specified to be in the program call.

**Solution**: Verify that your file is not named differently, or that your directory path is wrong and needs to be changed to the location of the assembly file.

##### “Incorrect operand type for .STRZ operation.”

**Cause**: The operand in the .STRZ instruction is not a string.

**Solution**: Change the operand in the instruction to be a string, surrounded by quotation marks.

##### “Incorrect operand type for operand [value]”

**Cause:** The operand specified does not meet the requirements for that type of operand.

**Solution:** Check the requirements of the instruction’s operand(s) in the instruction section.

##### “Incorrect operands for .FILL operation.”

**Cause**: There is an operand in the .FILL instruction that is out of range.

**Solution**: The .FILL operation requires a hexadecimal or decimal constant between the ranges of x0 to xFFFF or -32,768 and 32,767. Also, make sure that the proper sign precedes the number; a lower case ‘x’ needs to be before a hexadecimal number, and the pound sign “#” must be before a decimal number.

##### “Incorrect spacing.”

**Cause**: This error is caused by there not being the proper space between label, operation, and operand in a record.

**Solution**: Verify that you have the record space correctly. For information on how spacing should be formatted, see the Input Structure section.

##### “Incorrect Usage of .EQU. Requires a label and one operand.”

**Cause**: There is more than one operand in the .EQU instruction.

**Solution**: In an .EQU instruction, only one label and one operand can be given. If there are any more than that, an error is generated. See the section on the .EQU pseudo-operation for further information.

##### “Invalid string operand”

**Cause:** A string in the operand of an instruction is improperly formed.

**Solution:** Check that the operand is a string that is surrounded by quotation marks. For further information of the requirements on strings, see the .STRZ instruction section.

##### “Literal value # is out of bounds”

**Cause**: The literal value specified in the operand field of a load instruction is out of bounds.

**Solution**: Verify that the value in the operand is between decimal -32,768 and 32,767 (or x0 to xFFFF in hexadecimal).

##### “Multiple .END instructions are not allowed.”

**Cause**: There is more than one .END instruction in the assembly program.

**Solution:** Remove all but one .END instruction. Also, make sure that the .END instruction is the last instruction in the file.

##### “Multiple .ORIG instructions are not allowed.”

**Cause**: You can only have one .ORIG instruction per program. The program detects more than one.

**Solution**: Remove all but the first .ORIG instruction. For further information on Header records, see the Header Record section.

##### "No label allowed on .ENT instruction."

**Cause**: A .ENT instruction has a label.

**Solution**: Ensure that all .ENT instructions do not have a label.

##### "No label allowed on .EXT instruction."

**Cause**: A .EXT instruction has a label.

**Solution**: Ensure that all .EXT instructions do not have a label.

##### “Operand # out of bounds”

**Cause:** A number specified in the noted record is out of the range of the numbers accepted.

**Solution:** Verify that the numbers in the noted record are within range.

##### “Operand of .ORIG must be an immediate value.”

**Cause**: An immediate is not in the operand position of the .ORIG instruction.

**Solution**: The operand of the .ORIG instruction must be an immediate value, not a label or other form of value. See the section on Immediates (in the Input section) for further information.

##### “Operand value # is out of bounds”

**Cause:** The program is attempting to define a symbol that has a value outside of the bounds of possible numbers.

**Solution:** Verify that the operand of the symbol is within in bounds.

##### “Program exceeds limit for maximum number of literals.”

**Cause:** There are more than 50 literals defined in the assembly program.

**Solution:** As the limit on literals in a program is 50, remove as many as needed to be under or at the limit.

##### “Program exceeds limit for maximum number of source records.”

**Cause:** There are more than 200 source records in the assembly program.

**Solution:**  Reduce the size of the program so that it includes no more than 200 source records.

##### “Program exceeds limit for maximum number of symbols.”

**Cause:** There are more than 100 symbols defined in the assembly program.

**Solution**: As the limit on symbols in a program is 100, remove as many as needed to reach the limit.

##### “Program is missing .ORIG and/or .END instructions.”

**Cause:** There are no starting or ending records in the program.

**Solution:** Make sure that there is one header record at the beginning of the program, and one end record at the end of the program.

##### “Program loads into memory outside of the addressable range.”

**Cause:** The program has records that are loaded into memory addresses out of the range of memory available. In other words, the last instructions (or all) in the program are to be defined improperly in a memory location of above xFFFF.

**Solution**: Make the program relocatable, or change the initial memory address to fit within the bounds of memory.

##### “Program spans multiple memory pages. Relocate or shrink the program to fit inside one memory page.”

**Cause:** The program compiled spans more than one page of memory.

**Solution:** Reduce the size of the file to fit the memory requirements.

##### “Relocatable symbol [value] cannot be used in a non-relocatable operand”

**Cause:** The program attempts to use a relocatable symbol where a non-relocatable symbol (or absolute) is required.

**Solution:** Check that the instruction and its operands meet the requirements specified in the instruction section.

##### “Symbol redefinition is not allowed.”

**Cause**: This error is caused when the assembly file tries to define a symbol more than once, while it can only be defined once.

**Solution**: Rename the symbols that are used more than once, or remove records defining the symbols extra times.

##### “Symbol referenced in operand not defined.”

**Cause**: There is a reference to a symbol that is not defined in the program.

**Solution**: If a symbol is specified in an operand, it must be defined somewhere in the program. Otherwise, the program will not compile.

##### "Undefined .ENT symbol [symbol]."

**Cause**: There is a symbol defined in a .ENT operand that is not defined in this segment.

**Solution**: Define a symbol with the name [symbol] in the program segment.

##### “Undefined symbol [symbol]”

**Cause:** The program attempts to equate a symbol with an undefined symbol.

**Solution:** Define the symbol in the operand of this record, or change the value of the operand to another symbol or constant.

##### “Undefined symbol [value]”

**Cause:** The program attempts to use a symbol that has not been defined in the program.

**Symbol:** Either define or change the symbol that was specified in the instruction.