OneUp Wi11 Software Suite

User's Guide

Primary Author: Andrew Canale

Contents

Introduction

This Wi11 Software Suite is a collection of three interactive components (the Simulator, Assembler, and Linker) which simulates the operation of the theoretical Wi11 microprocessor. The assembler takes properly formatted assembly language code in an ASCII text file, and generates an object file for use with the linker. The linker takes one or more object files OR assembly source files (exclusively one or the other per run), and can either generate a complete object file for use with the simulator, or load the object file directly into the simulator if the user wishes (there is a command-line flag which supports these different operation modes).

System Requirements

Operating System:	UNIX, Linux, Windows, or Mac OS (with a C++ assembler).
Processor:	Pentium 1 equivalent or higher.
Memory:	128MB or higher.
Hard Disk Space:	10MB or more.

Required Knowledge

A user of this software suite is expected to be familiar with assembly language, including the strategic use of registers, labels, literals, memory addresses, operations and pseudo-operations. The user is also expected to know how to use command or bash prompts (depending on the computer's operating system). A user who is inexperienced in any of these regards would likely have trouble with (and little use for) the Wi11 software suite.

Simulator Characteristics

This Wi11 simulator is designed as an emulator for the functions of the (theoretical) Wi11 computer architecture. It allows a user to load a compiled program and run it as if it were on an identically designed physical system containing 128kB of system memory. The memory is word-addressable, so each address of memory references two (2) bytes, giving 65,536 addressable memory locations. The simulator supports basic assembly language commands and has eight (8) general-purpose 16-bit

registers in addition to the Program Counter (PC) and three (3) Condition Code Registers (CCRs). The CCRs are denoted by N (negative), Z (zero), and P (positive), and are updated whenever a value is written to a general purpose register, with the exception of the JSR and JSRR operations (see Operation Definitions). The simulator executes an object file provided by the user and will produce error messages as necessary.

Preparing the Software Suite

In order to use this software suite, the user must have executables (compatible with the user's computer architecture and operating system) for the simulator, assembler, and linker in the same directory (henceforth referred to as the 'base directory'.) If there are no prepared executables available, the user must compile the Wi11 software suite from its C++ source. A makefile is included for the user's convenience, which directs the compiler to compile all components of the suite simultaneously. Compilers known to work for this suite include Microsoft Visual Studio 2008, and GCC (the GNU compiler package.)

Running Wi11

The Wi11 software suite is capable of accepting multiple kinds of input. Standard operation requires that all input files be assembly source files formatted for the assembler. However, assembled code (in object files) may be passed directly to Wi11, if the user specifies the proper mode flag at runtime (-n).For a comprehensive list of flags, and input layout, run the Wi11 suite with the '--help' flag.

Options (runtime flags)

-s#: This flag allows the user to specify a maximum size for the symbol table. This also has the effect of limiting the number of literals to half the maximum number of symbols, and also limiting the number of lines in the output file to twice the maximum number of symbols. However, with a sufficiently large –s specification, the only real ceiling is the xFFFF address.

-t: This flag tells the assembler to include the set of predefined trap codes, so that the user doesn't need to explicitly define them. If the user specifies ' –t' and then assigns a label in this list, the user's value is overwritten by this table. Here are the labels with their corresponding TRAP:

```
out = TRAP x21        puts = TRAP x22

in = TRAP x23         halt = TRAP x25

outn = TRAP x31       inn = TRAP x33

rnd = TRAP x43
```

-a: Only assemble the input files, do not link or execute.

-o: Assemble and link, creating a single object file named "outfile"

-ox: Assemble, link, and execute; create "outfile" as in the '-o' option.

-x: Execute a pre-linked object file. Skip assemble and linking steps.

-n: Specifies that the input files are already assembled.The preceding '-o'

or '-ox' argument describes what should be done.

Notes

- The '-ox' option is the same as using the '-o' option then the '-x' option.
- The only compounding of arguments allowed is the '-ox' option. All othersmust be printed separately.

Examples

```
wi11 -t -s2000 -a file1.s file2.s file3.s
```

This will create the object files that correspond to each file individually. They wi1l not be linked or executed.

```
wi11 -l -t -o prog.o file1.s file2.s file3.s
```

This will assemble and link file1.s, file2.s and file3.s into prog.o.

```
wi11 -ox prog.o -n file1.o file2.o file3.o
```

This will link file1.o file2.o and file3.o into prog.o and execute it.

Assembly Input Syntax

The input (source) file must conform to certain standards in order to be understood and parsed properly. The most important two requirements for any code segment are the presence of a .ORIG pseudo-operation on the first non-comment line, and the presence of a .END pseudo-operation on the last line. Review the requirements for these in the pseudo-operation definitions section of this guide. Every line of the input file must conform to the following structural organization:

```
[Label] <Whitespace><Operation Field><Whitespace><Operands> [Whitespace] [Comment]
```

In this input syntax, items within brackets are optional, while items within carrots are mandatory. Omit the brackets and carrots specified here when calling the Wi-11 software suite.

Label

Labels are mandatory for .EQU and .ORIG, but optional for other operation/pseudo-operations. Labels are not permitted for the .END, .ENT, and .EXT pseudo-operations. Labels are case-sensitive, may only contain alphanumeric ASCII characters, and .ORIG labels may not exceed 6 characters in length. Labels must not begin with an uppercase ‘R’, nor begin with a lowercase ‘x’. When a label is defined, it can be used as a reference symbol in place of an operand in operation/pseudo-operation calls (except within operands on the same line in which the label is defined, and where prohibited by the specific operation/pseudo-operation invoked.)

Whitespace

Any number and combination of spaces and tabs are treated collectively as 'whitespace'. Outside of comments, whitespace may only 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 start of a comment. The assembler will still function properly even if a comment is started directly following the last operand, without whitespace.

Example: `ADD R1,R2,R3; this is a comment`

Operation Field

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 in All-uppercase letters. Pseudo-operations (used exclusively by the assembler) 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 "Operation and Pseudo-Operation Definitions" section of this guide.

Operand(s)

Operands follow whitespace after the instruction, and are finished by whitespace, end of line, or a semicolon. The types and numbers of operands are restricted by the specific operation or pseudo-operation invoked. A comma must be present between any two operands. No whitespace is permitted within operands, or between operands and commas. There are several types of operands supported by this assembler:

Registers

Registers are specified by an uppercase 'R', followed by the number of the specific register. For example, register 1 is written as 'R1'. The registers available to the user range from R0 to R7. Labels with the values 0-7 may be used as registers for the first operand where applicable.

Constants

Constants must be either decimal or hexadecimal. Decimal constants used as an operand must be immediately preceded by a pound sign '#'. Hexadecimal constants, in a similar fashion, must be preceded by a lowercase 'x'. Additionally, any alpha-characters used in hexadecimal must be uppercase. (example: Decimal 28 is written as '#28'. The hexadecimal equivalent of this is written as 'x1C'.)

Labels

Labels (defined elsewhere in the code) may be used in place of explicit operands in most situations.

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. 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.

Sample Assembly Input Syntax

This code segment is provided by Professor Paul Sivilotti of The Ohio State University.

```
; 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
```

Assembly Operation Definitions

ADD **DR,SR1,SR2**

DR : Destination Register, where the sum of SR1 and SR2 will be stored.

SR1 : Source Register 1, contains the first number to be summed.

SR2 : Source Register 2, contains the second number to be summed.

ADD **DR,SR1,imm5**

DR : Destination Register, where the sum of SR1 and imm5 will be stored.

SR1 : Source Register 1, contains the first number to be summed.

imm5 : an immediate number, decimal or hexadecimal, which is limited by its 5 bit 2's complement representation; meaning that it can range from decimal #-16 to #15, or

from hexadecimal x0 to x1F. Note that if the user wishes to use hexadecimal here, the assembler will interpret it in the equivalent (decimal) 2's complement representation.

AND **DR,SR1,SR2**

DR : Destination Register, where the bitwise AND of SR1 and SR2 will be stored.

SR1 : Source Register 1, contains the first sequence of bits to be ANDed into DR.

SR2 : Source Register 2, contains the second sequence of bits to be ANDed into DR.

AND **DR,SR1,imm5**

DR : Destination Register, where the bitwise AND of SR1 and imm5 will be stored.

SR1 : Source Register 1, contains the first sequence of bits to be ANDed into DR.

imm5 : an immediate number, decimal or hexadecimal, (again, limited by its 5 bit 2's complement representation, as explained above in ADD.)

BRx **pgoffset9**

There are several options for this Break operation. The 'x' must be replaced by any combination of 'N' for negative, 'Z' for zero, and/or 'P' for positive. Here are several possible permutations:

BR : Break never, so essentially a No OP.

BRN : Break if condition register has negative bit flagged.

BRNZ : Break if condition register has either negative or zero bits flagged.

BRNP : Break if condition register has either negative or positive bits flagged.

BRNZP : Break if condition register has any of negative, zero, or positive bits flagged. (This is effectively always branches.)

BRZ : Break if condition register has zero bit flagged.

BRZP : Break if condition register has zero or positive bit flagged

BRP : Break if condition register has positive bit flagged.

pgoffset9 should contain the page offset value (in hexadecimal, ranging from 0 – 1FF) which the machine should branch to if any of the specified N/Z/P condition code register(s) are flagged.

DBUG

Manually debugs the machine, causing it to output register and memory contents. No operands allowed.

JSR **addr**

'Jump to Subroutine' at the memory location specified by the addr operand. This flow-control operation stores the current program counter into register 7, so that it's easy to return to the proper location. The 'RET' operation is the complementary return operation for this.

addr : can only range from #0 to #65535 or x0 to xFFFF because of the hardcoded size of a memory page. A corollary to this is that any label used in place of the addr operand must also lie within that same numerical range.

JMP **addr**

A simple 'Jump' to another address in the code within the same page. However, this jump does not store the program counter, so it should only be used in situations where the program doesn't need to return to its previous location.

JSRR **BR,index6**

'Jump to Subroutine Register Relative', which acts the same as JSR with one exception: the address operand is split between the base register, and the 6-bit index6.

BR : the 'Base Register', which must contain the zero-extended 9 most significant bits of the memory location to where the user would like to jump.

index6: an immediate decimal or hexadecimal number which is combined with the base register to get the full 16 bit address where the program must jump to.

JMPR **BR,index6**

'Jump Register Relative', which behaves just like JSRR, with the exception that it does not store the program counter into register 7 (just like JMP).

LD **DR,addr**

'Load', which retrieves the contents stored at addr, and copies them into the destination register. A unique feature of LD is the ability to use Literals in place of addr. Literals are specified by an equal '=' sign, followed by a constant within the range #-32768 to #32767 in decimal, or x0 to xFFFF in hexadecimal. The assembler will place the value indicated in the literal into a new location which comes after the last programmer-

defined location. The assembler will then replace the literal operand with the address at which the value was stored.

LDI **DR,addr**

'Load Indirect', which looks at the contents of addr, which holds the address where the desired contents reside. Those contents are then copied into the destination register.

LDR **DR,BR,index6**

'Load Register Relative', which computes the load address by combining the (most significant) zero-extended 9 bits stored in the base register, with the least significant 6 bits specified explicitly by index6. The contents located at this load address are copied into the destination register.

LEA **DR,addr**

'Load Effective Address', which copies the upper 8 bits of the program counter, along with the lower 8 bits of addr, into the Destination Register.

NOT **DR,SR**

DR : Destination Register, which will store the bitwise NOT of the Source Register.

SR : Source Register, which contains the bits to be inverted.

RET

'Return', changes the Program Counter back to whatever was stored in R7 by JSR or JSRR.

ST **SR,addr**

SR : Source Register, which contains the bits to be stored into memory.

addr : memory address at which the contents of the Source Register are to be stored.

STI **SR,addr**

'Store Indirect', which looks at the location specified by addr, and finds there the address at which the source register is to be stored.

STR **SR,BR,index6**

'Store register-indexed', which forms the store address by adding the zero-extended 6 bit offset index6 to the base register. The index is always interpreted as positive (#0 to #63).

TRAP **trapvect8**

This TRAP instruction has several possible hexadecimal-valued operands, with varying effects. Here is a list of all trapvect8 possibilities:

x21 : writes the character in R0 (from bits 7 to 0) to the console

x22 : write the null-terminated string pointed to by R0 to the console.

x23 : Read a single character from the keyboard, and the corresponding ASCII code is copied into R0.

x25 : Halt execution and print end message to the console.

x31 : write the value of R0 to the console as a decimal integer.

x33 : Read a decimal number from the keyboard, and store it in R0.

Note*:* The user need not use the hex values explicitly if the '-t' flag is used, but may instead use the symbols specified in the '-t' part of the operation modes section in this guide.

Assembly Pseudo-Operation Definitions

.ORIG **addr**

.ORIG must be the first non-comment record in the source. The addr operand is used to specify the first address at which the program is supposed to be loaded. Omitting the addr signifies that the code is designed to be relocatable. There must be a label on .ORIG, which is the name of the segment. Relocatable programs must fit within one page of memory (512 16-bit 'words')

.END **beginaddr**

.END signifies the end of the input program. beginaddr specifies the address at which execution is supposed to begin. The address beginaddr must be present within the assembly code. .ORIG <= beginaddr < (.ORIG + Segment Length). If beginaddr is omitted, execution will begin at the load location specified by .ORIG. Anything after the .END record is ignored by the assembler.

.EQU **n**

'Equates'. All .EQU instructions must have a label, which is 'equated' with the constant n. The operand field can be a previously defined symbol (label) or a constant. For example,

if the user writes ‘seven .EQU 7’, then any time the label ‘seven’ is used, it will be understood by the assembler to be 7.

.FILL **n**

.FILL defines a one-word quantity whose content is the value of the operand n, which must be either decimal or hexadecimal. Decimal values must range from #-32768 to #32767. Hexadecimal values must range from x0 to xFFFF. A corollary to this is that any symbol used as an operand for .FILL must also be within the accepted numerical range.

.STRZ **“…”**

.STRZ is used to define a block of words which hold the characters of the null-terminated string. The string operand must be within quotation marks. The number of blocks/words occupied is one more than the number of characters in the string (because of the null termination).

Example: `Label .STRZ “test” ;`

Now, using ‘Label’ in the code points to the address of the first character in the string. This example would require 5 blocks in memory.

.BLKW **n**

.BLKW sets up a block of storage, with n specifying the number of words. n must be at least x1, and at most xFFFF (or the equivalent values in decimal). This instruction moves the assembler location counter forward the corresponding number of words, in a way that allocates that space for the user’s use. The block of storage is not initialized by this pseudo-operation.

The following pseudo-operations are handled exclusively by the linker component. *Labels are not allowed to be defined*on either of these pseudo-operations. Multiple instances of .ENT and .EXT are permitted.

.ENT **Entry_name**

.ENT takes at least one symbol as an operand (multiple symbols are separated only by a comma), that must be defined in the current segment, and are permitted to be referenced by other segments (files) with which this segment will be linked.

.EXT **External_name**

.EXT takes at least one symbol as an operand (multiple symbols separated by a comma), that may be referenced legitimately by this segment, but are not defined in this segment. The symbols must be defined in some other segment with which this segment will be linked.

.MAIN **n**

.MAIN, which is mandatory when linking multiple files, allows the user to specify the overall beginning load address for the linked segments. The letter 'H' in the header record of the produced object file is changed to 'M' to signify that .MAIN is used. Only one instance of .MAIN may be present throughout all of the linked segments.

Runtime Messages

"Absolute value in instruction that requires a relative."

This message will appear when the user improperly uses an absolute value in an instruction which expects a relative value. For example, this can happen when a program is designed to be relocatable by specifying .ORIG x0, but in subsequent instructions the user uses memory addresses which are not within the code segment. (i.e. x3000, when the segment only spans addresses x0 to x200)

"Argument is the empty string (misplaced comma?)."

This message appears when the user has inadvertently typed two (or more?) commas in the operand field, which confuses the assembler with an empty string as input.

"Argument to ".END" instruction is outside declared memory."

This message will appear if the assembler encounters a .END pseudo-operation with an invalid address at which to begin execution.

"Argument to ".ORIG" not hex."

This message will appear if the user attempts to enter a decimal number (with a # sign, otherwise the assembler would assume that the decimal number is hex instead. i.e. 2020 decimal would be misinterpreted as 2020 hex = 8224 decimal). This message can also appear if there are any characters outside the range of 0-9 or A-F.

"Argument to ".STRZ" is not a string."

This message informs the user that the operand for a .STRZ instruction is not a string. This could appear if the quotation marks on either end of the string are omitted, or if the user attempts to use a label as the operand.

"Attempt to redefine label."

This message will appear if the user attempts to define the same label more than once.

"Constant value out of range."

This message alerts the user that an invalid constant was encountered. The user should ensure that the constant entered is within the limits of the 5,6, or 9 bit representation; depending on the instruction.

"End of string not found."

This message will appear when the closing quotation mark is omitted on a .STRZ instruction.

“Error: file … could not be opened.”

This appears when the assembler encounters an IO error when trying to use the specified file.

"Error: Undefined Result Code Enumeration"

This appears when the assembler encounters an error for which there is no written explanation. Under normal operation, the user should not encounter this error.

“Error: Usage: wi11-asm <input_file><output_file> [-t] [-s<max_size>]”

This error is caused by invalid syntax when calling the assembler, with the input/output files, and any flags as arguments separated only by a single space each. The ‘-t’ and/or ‘-s’ flags must come after the input and output files, but the order of those two flags is irrelevant.

"Extra characters found after end of string."

This message will appear when the user has non-whitespace character(s) after the closing quotation of a .STRZ instruction, which is also outside of the acceptable comment area.

"File could not be opened."

This message will appear if the assembler is prevented from reading the specified file, possibly because of some IO error, the file is in use by some other software, or the console does not have sufficient permission to read the file.

"File has no end record."

This message will appear if the input file does not contain a .END pseudo-operation at the end of the file.

"First non-comment line should contain ".ORIG" instruction."

This message will appear if the .ORIG pseudo-operation is missing or misplaced.

"File not found."

This message appears if the assembler is unable to locate the specified file within the same folder as the assembler executable itself. The user should ensure that the filenames were typed correctly, and that the file is not in a different directory of the computer’s file system.

"Immediate value not expressible in 5 bits."

This message will appear when the user attempts to use an immediate constant which is not within the range of #-16 to #15, or x0 to x1FF.

"Index value not expressible in 6 bits."

This message will appear if the user enters an index6 operand which is not within the range of x0 to x3F (#0 to #63).

"Instruction not recognized."

This message appears when the assembler encounters an invalid operation or pseudo-operation. The user should ensure that the instruction is documented in this guide, and that there are no spelling mistakes present.

"Instruction requires label."

This message appears when an instruction which requires a label, is left without one. The user should ensure that there is a valid label on .ORIG, .EQU, and .STRZ instructions.

"Invalid argument."

This message will appear with improper use/placement of labels as operands. (as in, the label used is not used in a location which supports labels.)

"Invalid CCR mask for branch instruction."

This message will appear if the user uses any characters other than 'N', 'Z', or 'P' following the BR of a BRx instruction.

"Invalid decimal following '#'."

This message will appear when a positive or negative decimal representation contains anything other than digits 0-9. Negative decimal numbers have the negative sign in between the pound sign and the first digit. i.e. #-9

"Invalid hex following 'x'."

This message will appear when a hexadecimal representation is malformed. The user should ensure that all characters are uppercase, alphanumeric, and not letters G-Z. i.e. only digits 0-9, and letters A-F are acceptable.

"Invalid maximum symbol table size. Usage: ... [-s<max_size>]"

This error is caused by the user entering an invalid (i.e. < 1) maximum symbol table size. Specify a larger maximum symbol table size. There is no hardcoded maximum symbol table size.

"Invalid number of arguments."

This message will appear when there are too few or too many operands for a given instruction. Ensure that the number and expected type of operands is observant of the specifications in this guide.

"Label is not followed by an instruction."

This message informs the user of a syntax error in which a label is specified, but the operation or pseudo-operation has been omitted. The user should ensure that the input syntax conforms to the proper specifications.

"Label not found. ((Forward reference to .FILL label?Case-sensitvity issue?))"

This message appears when the user attempts to use an undefined label. The user should ensure that the proper upper or lowercase label is used in every instance in which it's used.

"Label starting with 'R' or 'x'."

This message alerts the user that an invalid label was specified in the source code. Labels are not permitted to start with 'R' or 'x', in order to ensure that there is no confusion between labels and registers or hexadecimal numbers.

"Literals may only be used with the LD instruction."

This message will appear if the user attempts to use a literal anywhere except in an LD instruction.

"Maximum number of literals reached. ((Alter with '-s'?))"

This message will appear when the index of literals is completely full. Try specifying a larger number of symbols to increase the maximum number of literals supported.

"Maximum number of symbols reached. ((Alter with '-s'?))"

This message will appear when the index of symbols is completely full, and it is no longer possible to define more within the execution constraints.

"Maximum object file size reached. ((Alter with –s?))"

This message will appear if the number of records in the assembler-generated object file exceeds the limit set by (two multiplied by) the number of symbols.

"Non-existent register as argument."

This message will appear if the user attempts to use a register less than R0, or more than R7.

".ORIG" label longer than six characters."

This message appears when the label on .ORIG exceeds 6 characters.

"Page Error: Address references a different page."

This error will appear when the user tries to use an address which is present on a different page of memory. A complete segment of code must fit within one page of memory (which is xFFFF lines.)

"Second .ORIG instruction found."

This message will appear if there are multiple .ORIG instructions specified. The user needs to have exactly one .ORIG instruction in the source code.

"Successful."

This message means that the compiler has finished execution without any issues.

"Unexpected end of file."

This message will appear if the input file is corrupt or incomplete. Such situations include the assembler reaching the end of the file while in the process of parsing an instruction and its operands.

Changes to User's Guide from Lab 2 Documentation

- Additions to 'Assembly Pseudo-Operations Definitions' for Pseudo-Operations used by the linker (.ENT and .EXT)
- Additions to 'Runtime Messages' for messages from the linker.
- New 'Running Wi11' section to explain operation modes of the entire Wi11 software suite.