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

|  |
| --- |
|  |

Contents

[Introduction 3](#_Toc285796549)

[Installing the Software 3](#_Toc285796550)

[Writing a Program 3](#_Toc285796551)

[Required Software 3](#_Toc285796552)

[Structure 3](#_Toc285796553)

[Header Record 3](#_Toc285796554)

[Relocatable Programs 3](#_Toc285796555)

[Text Records 3](#_Toc285796556)

[Labels 4](#_Toc285796557)

[Instructions 4](#_Toc285796558)

[Operands 7](#_Toc285796559)

[Pseudo Operations 8](#_Toc285796560)

[Comments 9](#_Toc285796561)

[End Record 9](#_Toc285796562)

[Compiling a Program 10](#_Toc285796563)

[Commands 10](#_Toc285796564)

[Output 10](#_Toc285796565)

[Debugging Error Messages 10](#_Toc285796566)

# Introduction

# Installing the Software

# 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 will serve 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 coder 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 will be done by the program and not by the user.

## Text Records

### Labels

Labels can be used to refer to a specific record 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.

### Instructions

#### 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 will perform the addition arithmetic operation on two specified arguments, and store 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 will be 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 will perform a bitwise and operation on two specified arguments, and store the result in the specified register. The instruction can either and two registers, or and a register with a 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 will be 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 will perform a bitwise not operation on a register, then store 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 will 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 will be categorized by their addressing mode; the differences between these modes will be 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 (pgoffset9) 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 will 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 (a pgoffset9). 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 will 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 will 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 will 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 will be stored. SR represents the source register where the data to be stored is located. BR represents the base register that will be 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

### Operands

There are several different forms of operands that can be used in the operand fields of records; Registers, Constants, 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

#### Constants

#### Immediates (imm5)

#### Addresses (addr)

#### Indexes (index6)

#### Symbols

#### Literals

### Pseudo Operations

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

#### .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 will refer 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 x1 and xFFFF. 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 will not contain any data after created. A label is optional; if used, it will refer 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

### 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 will be 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 will ignore 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

# Compiling a Program

## Commands

## Output

# Debugging Error Messages