Contents

List of Figures

List of Tables

Project Overview

This document is the defining source of information for the development of an assembler language translator for the C6461 Computer. It contains a description of the architecture of the C6461 Simulator to provide the student with sufficient information to develop the assembler, including a representation of the internal structure and the binary formats for the instructions that the C6461 must execute.

C6461 Instruction Set Architecture

Instruction Set Architecture Overview

The instruction set architecture (ISA) consists of the structures and functions visible to the assembly language level programmer or to the compiler writer. It acts as the interface between a computer's hardware and software, defining the set of instructions that a processor can execute, including operations such as arithmetic, data movement, and control flow. The ISA determines how software communicates with the hardware, specifying how instructions are formatted, how data is accessed and manipulated, and how the processor responds to various commands. The design of an ISA affects the efficiency, performance, and power consumption of a computer system, as well as its compatibility with software.

The development of the C6461 begins with the development of a C6461 Assembler. During this project you will develop a detailed understanding of the machine as described in this document. In the “which came first” question regarding a particular machine’s hardware design or ISA design, you will see that the answer is not always clear.

Think of the machine as an object in an Object Oriented (OO) programming language. Assembly code, often called Machine Code, generated by a compiler or the programmer in the form of binary numbers, is directly executable on the machine. In our Von Neumann architecture, the C6461 executes assembly code stored in memory by fetching an instruction, determining the type and function of the instruction, and then executing that instruction. It then proceeds to the next instruction.

Internal Structures

The internal structures consist of register hardware, memory, program counter (location of next instruction to execute). As with a software defined object, there are internal structures that are not seen by the programmer and are not necessarily fixed. This allows the development of and improvements in the internal structure of the hardware. As a common example, the Intel 64 Instruction Set Architecture has several implementations, evolving from the original Pentium architecture and branching out into other internal designs by, for example, AMD. Like the OO Design of Software, the external interface remains the same, with the ISA being the focal point of reuse.

C6461 Internal Structure

Figure 1 C6461 Computer Organization shows the internal components of the C6461 Computer. The components of C6461 are as follows:

Buses

- The Main Bus (vertical blue arrow) is a set of 16 data wires (our machine is a 16 bit machine) along with control information for accessing the bus.
- Memory Bus from **MBR to memory**, is a 16 bit bus (16 data wires)
- Memory Address Bus from **MAR to Memory** is a 12 bit bus carrying an address of the memory location to be accessed (read or write)
- Other lines are directly connected.

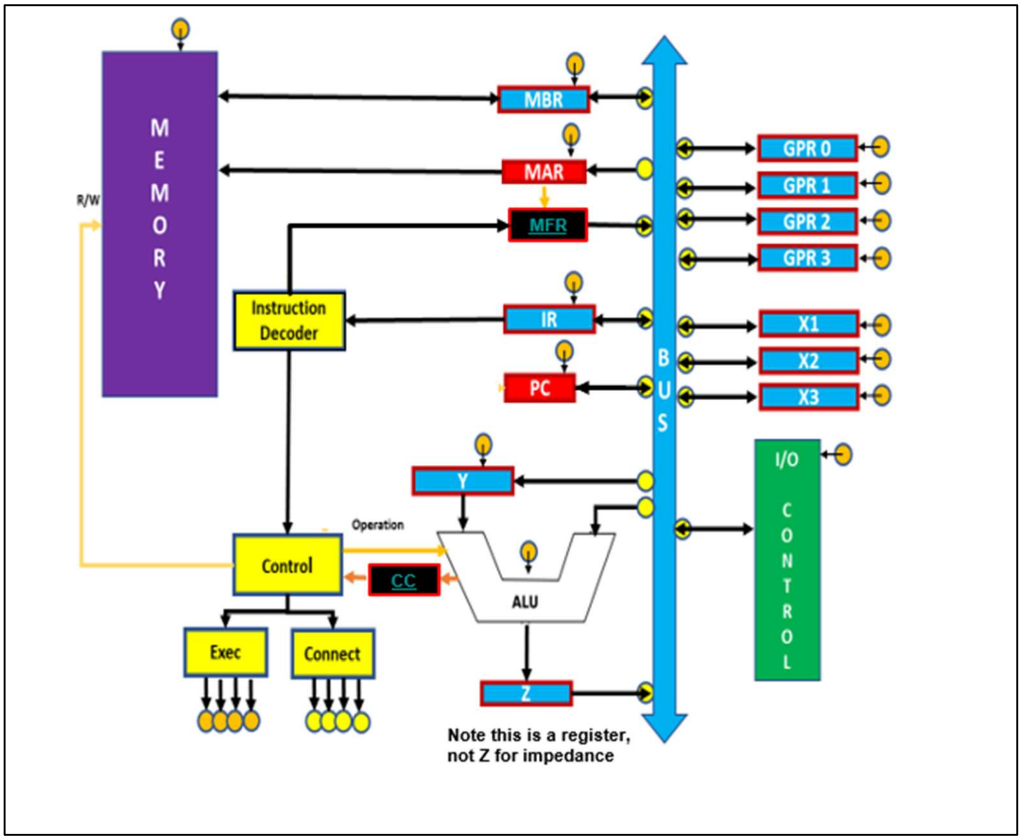


Figure 1 C6461 Computer Organization

Programmer Accessible Registers

- Four 16-bit general purpose registers GPR0-GPR3, used for storing fetched or computed numbers
- Three 16-bit index registers X1-X3 that can hold data but are primarily used to hold addresses or parts of addresses and be used by instructions to compute addresses (note that these will be referred to IX1-IX3 in text below. There is no X0. Use of 0 in an index register field will indicate no indexing.

- In a C6461 instruction, a number can be added to a value from an index register to compute an **Effective Address** (EA)

Additional Registers

Mnemonic	Size	Name
PC	12 bits	Program Counter: address of next instruction to be executed. Note that 2^{12} = 4096.
CC	4 bits	Condition Code: set when arithmetic/logical operations are executed; it has four 1-bit elements: overflow, underflow, division by zero, equal-or-not. They may be referenced as cc(0), cc(1), cc(2), cc(3). Or by the names OVERFLOW, UNDERFLOW, DIVZERO, EQUALORNOT
IR	16 bits	Instruction Register: holds the instruction to be executed
MAR	12 bits	Memory Address Register: holds the address of the word to be fetched from memory
MBR	16 bits	Memory Buffer Register: holds the word just fetched from or the word to be /last stored into memory
MFR	4 bits	Machine Fault Register: contains the ID code if a machine fault after it occurs
GPRs 0-3	16 bits	4 general purposes registers for computational use.
IXRs 1-3	16 bits	3 index registers for addressing
Other registers as needed	Defined by student	Internal registers for your simulator. Note in the diagram the Y and Z registers are used for input and output of computations respectively

Table 1: Non Addressable Registers

Memory

- Word addressable (Word size for C6461 is 16 bits
- Memory of 2048 16-bit words, expandable to 4096, we will use 2048
- Read only Memory used for system load or other functions. We will simulate the read only memory with system memory load and system memory clear functions

Reserved Memory Locations

The C6461 computer, like all computers, has a set of reserved memory locations for use by operating system functions. Reserved memory locations, in earlier computers, could usually be accessed by a programmer through the front panel. The programmer at the front panel could enter instructions directly into any place in memory or activate a load program which could read a program into any place in memory. Once loaded and in execution, reserved memory locations could not be accessed.

Memory Address	Usage
0	Used for trap instruction, which when executed, looks in location 0 for the address of a list of address to up to 16 routines to execute, with the trap code being 0-15
1	Used for machine faults (errors) and contains the address of code to handle machine faults. Implemented in part III. See the machine fault table below.
2	Location for storing the current PC when a trap occurs. This allows a trap to save the PC and then use the address at location 0 and the trap code to execute trap procedures, after which it can return to the program causing the trap.
3	Not used
4	Store PC for machine fault. Similar to location 2
5	Not used

Table 2: Reserved Memory Addresses

C6461 Operation

Interrupts

The C6461 does not implement interrupts or complex I/O.

Machine Faults

Implemented in Part III of the project): ***(How to set the Machine Fault Register-MFR)***

An erroneous condition in the machine will cause a machine fault. The machine traps to memory address 1, which contains the address of a routine to handle machine faults. Your simulator must check for faults.

The possible machine faults that are predefined are:

ID	Fault
0	Illegal Memory Address to Reserved Locations ***MFR set to binary 0001***
1	Illegal TRAP code ***MFR set to binary 0010***
2	Illegal Operation Code ***MFR set to 0100***
3	Illegal Memory Address beyond 2048 (memory installed) ***MFR set to binary 1000***

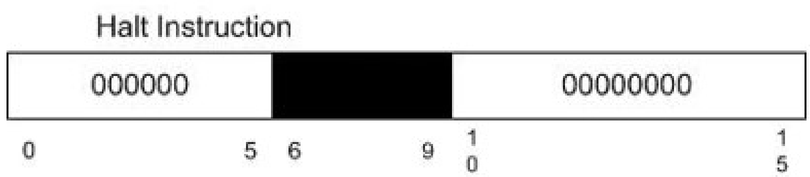
When a Trap instruction or a machine fault occurs, the processor saves the current PC and MFR saves (stored with MFR register) contents to the locations specified in Table 2: Reserved Memory Addresses above, then fetches the address from Location 0 (Trap) or 1 (Machine Fault) into the PC which becomes the next instruction to be executed. This address will be the first instruction of a routine which handles the trap or machine fault.

Traps are not implemented until phase III. It is recommend that location 1 contain the number 6 (first available non-protected location) and that location 6 contain a halt to be able to view the fault.

C6461 Instructions

Miscellaneous Instructions:

Miscellaneous instructions do not fit into another category (given the size of the machine). The formats are:



Note that areas that are blocked out in the above formats are not used in the decoding or interpretation of the instruction . The numbers below the figures illustrate bit positions.

OpCode$_8$	Instruction	Description
00	HLT	Stops the machine.
30	TRAP code	Traps to memory address 0, which contains the address of a table in memory. Stores the PC+1 in memory location 2. The table can have a maximum of 16 entries representing 16 routines for user-specified instructions stored elsewhere in memory. Trap code contains an index into the table, e.g., it takes values 0 – 15. When a TRAP instruction is executed, it goes to the routine whose address is in memory location 0, executes those instructions, and returns to the instruction stored in memory location 2. The PC+1 of the TRAP instruction is stored in memory location 2.

Table 3: Miscellaneous Instructions

Do not implement the TRAP instruction until Part III.

Load/Store Instructions

Load/Store instructions only move contents between memory and registers. Memory addresses are computed and denoted as Effective Addresses (EA). The basic instruction format is shown below:

Load/Store Instruction

Opcode	R	IX	I	Address
0 5	6 7	8 9	10	11 15

Field	#Bits	Description
Opcode	6	Specifies one of 64 possible instructions. Not all may be defined in this project
IX	2	Specifies one of three index registers; may be referred to by X1 – X3. O value indicates no indexing.
R	2	Specifies one of four general purpose registers; may be referred to by R0 – R3
I	1	If I =1, specifies indirect addressing; otherwise, no indirect addressing.
Address	5	Specifies one of 32 locations - Unsigned

Table 4: Field Definitions for Load/Store Instructions

To address all of memory, indexing will be required. We will use a base address indexing scheme. The value of IX is used to select an index register and to specify indirect addressing:

00	No Indexing
01	Index Register 1
10	Index Register 2
11	Index Register 3

Computing the Effective Address:

```
Effective Address =                 //First add Address Field and Index Register
    If IX field =00 Then
        EA = c(Address)             // contents of the Address Field
    Else IF c(IX) = 1..3 Then       // IX field has a valid index register number
        EA = c(IX) + c(Address)
    EndIf

    If I field = 1 Then             //indirection – the EA computed above is the
        EA = c(EA)                  //address in Memory of the EA, so fetch it.
    EndIf
```

The effective address is the location of the operand in memory. The operand could be a source or a destination.

Load/Store instructions move data from/to memory and a register. The access to memory may be indirect (by setting the I bit).

Notation:
c(EA) means "fetch the contents of the memory location specified by EA," where EA = 0 … maximum memory size, or c(IX) means "get the contents of the field IX in the instruction".

[,I] in an instruction indicates that the indirect bit in assembly code is optional. So, the instruction might not have the ,I at the end.

OpCode$_8$	**Instruction**	**Description**
01	LDR r, x, address[,I]	Load Register From Memory, r = 0..3 r <– c(EA) note that EA is computed as given above
02	STR r, x, address[,I]	Store Register To Memory, r = 0..3 Memory(EA) <– c(r)
03	LDA r, x, address[,I]	Load Register with Address, r = 0..3 r <– EA
41	LDX x, address[,I]	Load Index Register from Memory, x = 1..3 Xx <- c(EA)
42	STX x, address[,I]	Store Index Register to Memory. X = 1..3 Memory(EA) <- c(Xx)

Table 5: Load/Store Instructions

As an example, consider the instruction: LDR 3,0,31 (Symbolic Form)

This would be read as: Load register 3 with the contents of the memory location 31. Since IX = 00, there is no indexing, so 31 is the EA.

This instruction would be encoded as:

Opcode	R	IX	I	Address
000001	11	00	0	11111

Note that in this representation, the contents of the A field are always considered positive.

Transfer Instructions

The Transfer instructions change control of program execution. Conditional transfer instructions evaluate the value of a register. Note R = 0…3. They have the same format as the Load/Store instructions.

Notation: c(r) means "the contents of register r," r = 0..3

OpCode$_8$	Instruction	Description
10	JZ r, x, address[,I]	Jump If Zero: If c(r) = 0, then PC <– EA Else PC <- PC+1
11	JNE r, x, address[,I]	Jump If Not Equal: If c(r) != 0, then PC <–- EA Else PC <- PC + 1
12	JCC cc, x, address[,I]	Jump If Condition Code cc replaces r for this instruction cc takes values 0, 1, 2, 3 as above and specifies the bit in the Condition Code Register to check; If cc bit = 1, PC <– EA Else PC <- PC + 1
13	JMA x, address[,I]	Unconditional Jump To Address PC <- EA, Note: r is ignored in this instruction
14	JSR x, address[,I]	Jump and Save Return Address: R3 <– PC+1; PC <– EA R0 should contain pointer to arguments Argument list should end with –1 (all 1s) value
15	RFS Immed	Return From Subroutine w/ return code as Immed portion (optional) stored in the instruction's address field. R0 <– Immed; PC <– c(R3) IX, I fields are ignored.
16	SOB r, x, address[,I]	Subtract One and Branch. R = 0..3 r <– c(r) – 1 If c(r) > 0, PC <- EA; Else PC <- PC + 1
17	JGE r,x, address[,I]	Jump Greater Than or Equal To: If c(r) >= 0, then PC <- EA Else PC <- PC + 1

Table 6: Transfer Instructions

OpCode 016 allows you to support simple loops. I like this instruction. It was included on the Data General Eclipse S/200 and many other computers of the minicomputer era.

Arithmetic and Logical Instructions

Arithmetical and Logical instructions perform most of the computational work in the machine.
For <u>immediate</u> instructions, the Address portion is the Immediate value.
The condition codes are set for the arithmetic operations. The maximum absolute value of the Immediate value is **31.** (5 bits without sign).

OpCode$_8$	**Instruction**	**Description**
04	AMR r, x, address[,I]	Add Memory To Register, r = 0..3 r<– c(r) + c(EA)
05	SMR r, x, address[,I]	Subtract Memory From Register, r = 0..3 r<– c(r) – c(EA)
06	AIR r, immed	Add Immediate to Register, r = 0..3 r <– c(r) + Immed Note: 1. if Immed = 0, does nothing 2. if c(r) = 0, loads r with Immed IX and I are ignored in this instruction
07	SIR r, immed	Subtract Immediate from Register, r = 0..3 r <– c(r) - Immed Note: 1. if Immed = 0, does nothing 2. if c(r) = 0, loads r1 with –(Immed) IX and I are ignored in this instruction

Table 7: Add/Subtract Immediate and to Memory Operations

As an example, add to r2 the contents of memory location 523.
ADD 2,1,23 where c(X1) = 500

Transfer the immediate value 10 to register 3 so that the value of register 3 is 10.
But register 3 may already have something in it!

```
STR 3,0,20        ; store register 3 contents in location 20
SMR 3,0,20        ; clear register 3!
AIR 3,10          ; load register 3 with 10
```

How do we test for overflow? Underflow? How do you know this occurs?
Hennessey and Patterson discuss this.

Register to Register Operations

Certain arithmetic and logical instructions are register to register operations. The format of these instructions is:

Opcode	Rx	Ry	
0 – 5	6 7	8 9	10 – 15

The blacked-out portion means that portion of the instruction is ignored. Rx and Ry refer to one of R0-R3.

Multiply/Divide and Logical Operations

$OpCode_8$	Instruction	Description
70	MLT rx,ry	Multiply Register by Register rx, rx+1 <- c(rx) * c(ry) rx must be 0 or 2 ry must be 0 or 2 rx contains the high order bits, rx+1 contains the low order bits of the result Set OVERFLOW flag, if overflow
71	DVD rx,ry	Divide Register by Register rx, rx+1 <- c(rx)/ c(ry) rx must be 0 or 2 rx contains the quotient; rx+1 contains the remainder ry must be 0 or 2 If c(ry) = 0, set cc(3) to 1 (set DIVZERO flag)
72	TRR rx, ry	Test the Equality of Register and Register If c(rx) = c(ry), set cc(4) <– 1; else, cc(4) <– 0
73	AND rx, ry	Logical And of Register and Register c(rx) <– c(rx) AND c(ry)
74	ORR rx, ry	Logical Or of Register and Register c(rx) <– c(rx) OR c(ry)
75	NOT rx	Logical Not of Register To Register C(rx) <– NOT c(rx)

Table 8: Multiply/Divide and Logical Operations

The logical instructions perform bitwise operations.

TRR 0,2 where r0 = 0 000 000 000 000 001 and r2 = 0 000 000 000 000 001.
Then the condition code register cc(4) gets 1, indicating equality
NOT 3 where r3 = 1 000 000 000 110 110
Then r3 = 0 111 111 111 001 001

Shift/Rotate Operations

Shift and Rotate instructions manipulate a datum in a register.

Arithmetic Shift (A/L = 0) instructions move a bit string to the right or left, with excess bits discarded (although one or more bits might be preserved in flags). The sign bit is not shifted in this instruction.

Logical Shift (A/L = 1) instructions move a bit string left or right, with excess bits discarded and zero(es) inserted at the opposite end.

Logical Rotate (A/L = 1) instructions are like shift instructions, except that rotate instructions are circular, with the bits shifted out one end returning on the other end. Rotates can be to the left or right.

The format for shift and rotate instructions is:

OpCode	R	A/L	L/R		Count
0 … 5	6 7	8	9	10 11	12 … 15

Note: We have 16-bit words, but the maximum value for Count can be 16. So, what happens when the Count is specified to be 15?

OpCode	Instruction	Description
31	SRC r, count, L/R, A/L	Shift Register by Count c(r) is shifted left (L/R =1) or right (L/R = 0) either logically (A/L = 1) or arithmetically (A/L = 0) XX, XXX are ignored Count = 0…15 If Count = 0, no shift occurs
32	RRC r, count, L/R, A/L	Rotate Register by Count c(r) is rotated left (L/R = 1) or right (L/R =0) either logically (A/L =1) XX, XXX is ignored Count = 0…15 If Count = 0, no rotate occurs

Table 9: Shift and Rotate Operations

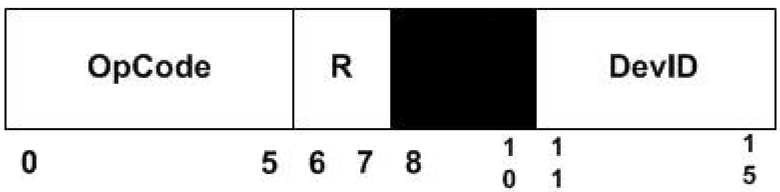
On arithmetic shifts to the right, the sign bit is replicated in the position 1 for each shift.
There is a lot going on here with these instructions. These are examples of some early machines which packed a lot of functionality into a few instructions.

So, suppose r3 = 0 000 000 000 000 110
Then, SRC 3,3,1,1 would yield r0 = 0 000 000 000 110 000
e.g., shift left bits 13…15

So, suppose r1 = 1 000 000 000 000 110
Then, SRC 1,2,0,0 would yield r1 = 1 110 000 000 000 001
e.g., shift right 2 bits
And underflow would be set. Why?

I/O Operations

I/O operations communicate with the peripherals attached to the computer system. This is a simple model of I/O meant to give you a flavor of how I/O works. For character I/O, the instruction format is:



OpCode	Instruction	Description
61	IN r, devid	Input Character To Register from Device, r = 0..3
62	OUT r, devid	Output Character to Device from Register, r = 0..3
63	CHK r, devid	Check Device Status to Register, r = 0..3 c(r) <- device status

We will assume the devices whose DEVIDs are:

DEVID	Device
0	Console Keyboard
1	Console Printer
2	Card Reader
3-31	Console Registers, switches, etc

Notes:
(1) You may only use the IN and CHK instructions with the console keyboard and the card reader.
(2) You may only use the OUT and CHK instruction with the console printer.
(3) Devices 3 – 31 are affected only by the IN and OUT opcodes. Some of these devices may be affected by only one of these opcodes. *Can you think of an example now?*

Floating Point Instructions/Vector Operations

Do not implement floating point numbers until Part IV

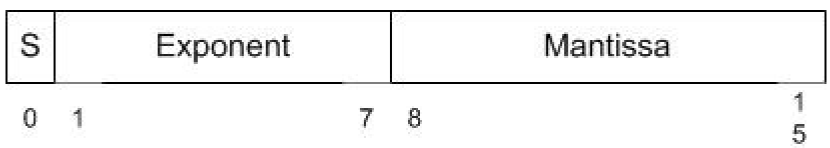
We have limited space in our instruction set, with only six bits for opcodes. So, we must limit our floating point and vector operations. This will give you a chance to think about how to write a software routine to do multiplication and division for both floating point numbers.

There are two floating point registers: FR0 and FR1. Each is 16 bits in length.

The format of a floating point number is the same as that for a load/store instruction, except that the r field takes only 2 values: 0 or 1 to specify the two floating point registers.

Vector operations are performed from memory to memory. This was used on several models of vector processors as opposed to using lots of expensive registers to hold vectors (unless you were Seymour Cray).

Floating Point numbers are 16 bits in length. So, a floating point number has the representation:



There are 7 bits for the exponent and 8 bits for the mantissa. The first bit of the exponent is its sign bit. The S bit (bit 0) is the sign of the entire floating point number. The exponent ranges from –63 to 64, e.g., -2^6-1 to 2^6.

OpCode	Instruction	Description
33	FADD fr, x, address[,I]	Floating Add Memory To Register c(fr) <– c(fr) + c(EA) c(fr) <– c(fr) + c(c(EA)) if I bit set fr must be 0 or 1. OVERFLOW may be set
34	FSUB fr, x, address[,I]	Floating Subtract Memory From Register c(fr) <– c(fr) - c(EA) c(fr) <– c(fr) - c(c(EA)) if I bit set fr must be 0 or 1 UNDERFLOW may be set
35	VADD fr, x, address[,I]	Vector Add fr contains the length of the vectors c(EA) or c(c(EA)), if I bit set, is address of first vector c(EA+1) or c(c(EA+1)), if I bit set, is address of the second vector Let V_1 be vector at address; Let V_2 be vector at address+1 Then, $V_1[i] = V_1[i]+ V_2[i]$, i = 1, c(fr).
36	VSUB fr, x, address[,I]	Vector Subtract fr contains the length of the vectors c(EA) or c(c(EA)) if I bit set is address of first vector c(EA+1) or c(c(EA+1)) if I bit set is address of the second vector Let V_1 be vector at address; Let V_2 be vector at address+1 Then, $V_1[i] = V_1[i] - V_2[i]$, i = 1, c(fr).
37	CNVRT r, x, address[,I]	Convert to Fixed/FloatingPoint: If F = 0, convert c(EA) to a fixed point number and store in r. If F = 1, convert c(EA) to a floating point number and store in FR0. **The r register contains the value of F before the instruction is executed.**
50	LDFR fr, x, address [,i]	Load Floating Register From Memory, fr = 0..1 fr <– c(EA), c(EA+1) fr <- c(c(EA), c(EA)+1), if I bit set
51	STFR fr, x, address [,i]	Store Floating Register To Memory, fr = 0..1 EA, EA+1 <– c(fr) c(EA), c(EA)+1 <- c(fr), if I-bit set

Table 10: Floating Point and Vector Operations

Note: Opcode 037 is a strange beast! It latches the result to FR0 when converting from integer to floating point – no other choices allowed!

So, the vector add instruction might be encoded as:

VADD 0, 1, 31 w/ I = 0

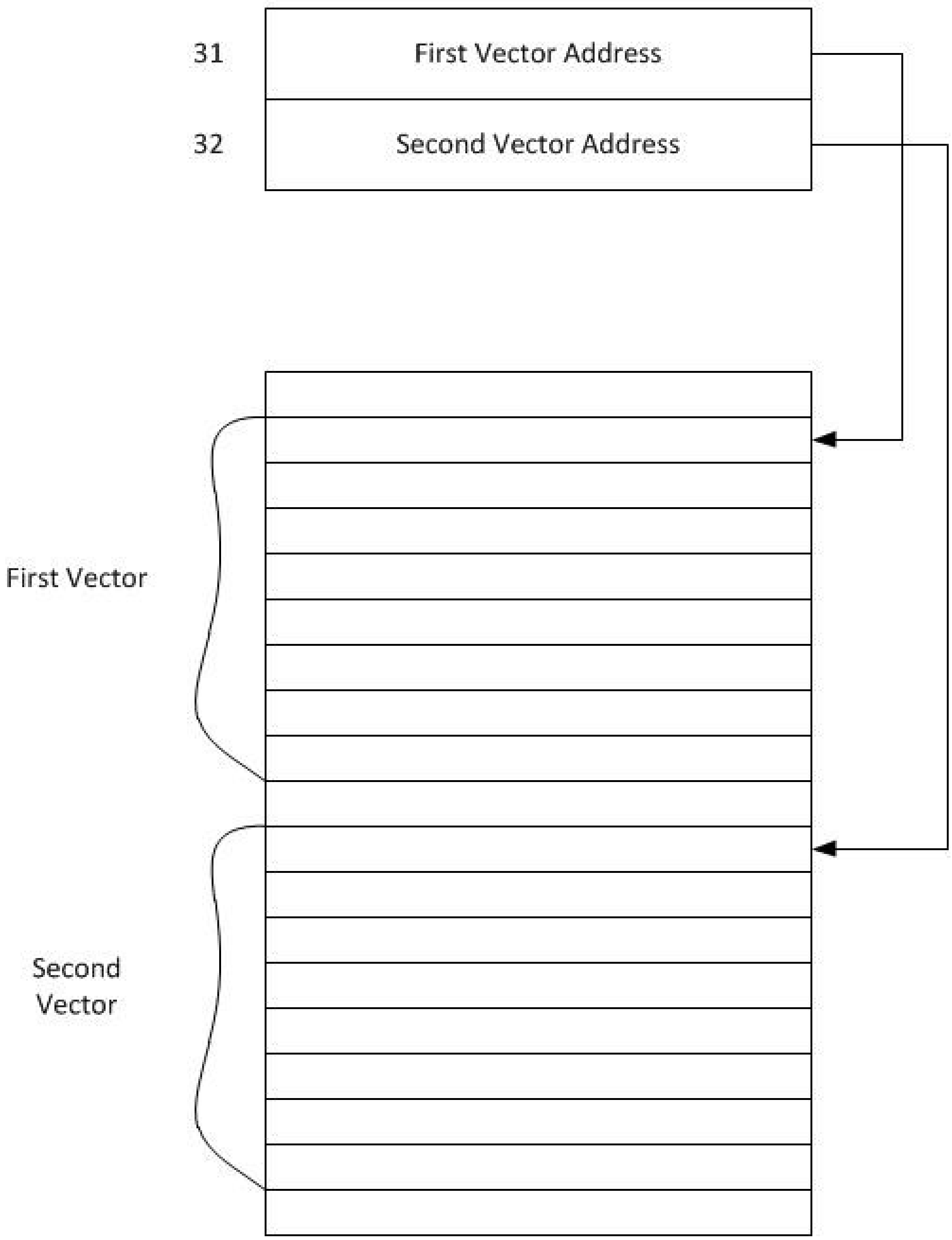
In memory this would look like:

Opcode	fr	I	IX	Address
011110	00	0	01	11111

R field designates either FR0 or FR1.

At memory location c(X0) + 31: address of first vector
At memory location c(X0) + 32: address of second vector
Each of these vectors would be c(fr) words long

How Vectors Are Stored in Memory



There is a lot for you to think about here!

Building the Assembler

Assembler High Level Description

An assembler translates "assembly language" instructions for a particular instruction set architecture into a numeric representation that can be loaded into the computer and executed. As seen in Figure 2 Overall Flow of C6461 Program Development below, a Source Program (text file) is developed by the programmer for input into the assembler. The assembler reads the source program and generates two types of files. One file is a Listing File that shows the results of processing the code by the assembler, and could include errors for the programmer to correct. A Load File is also generated. This is a numeric file containing, in the case of C6461 computer, two numbers per line. The first number is an octal address, and the second number is the octal contents of the number at that address. This file is sometimes called an object file. It can be loaded into the machine using an initiation sequence (Init) button.

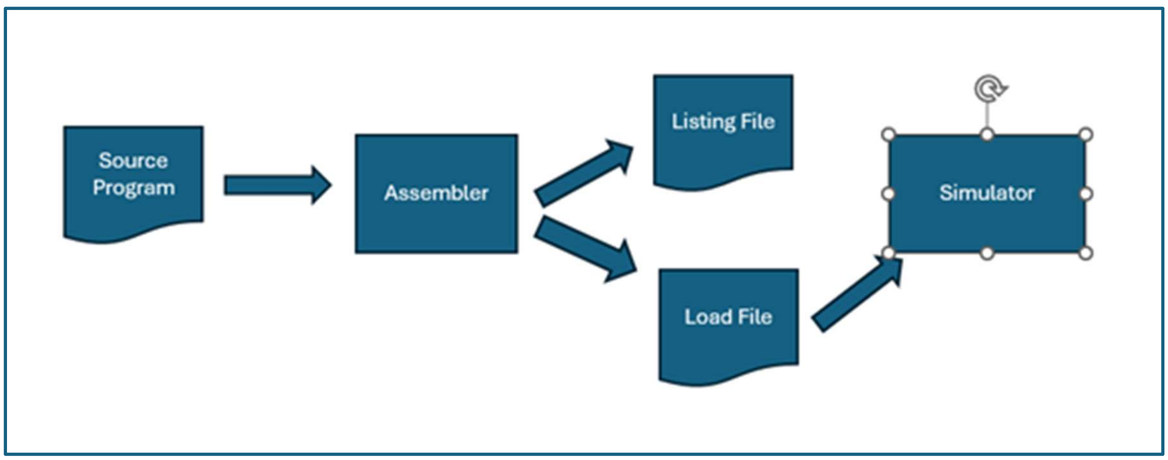


Figure 2 Overall Flow of C6461 Program Development

Note that in our simulator, we will be simulating a load when the Init button is pressed. This is artificial as a real machine will execute internal read only memory that looks for specific locations from which to load the file. The file being loaded is a binary file, in binary notation (as opposed to text).

Assembler Source File

Figure 3 Sample Assembler Source File below is a sample assembler language source file for our simulator.

```
            LOC     6            ;BEGIN AT LOCATION 6
            Data    10           ;PUT 10 AT LOCATION 6
            Data    3            ;PUT 3 AT LOCATION 7
            Data    End          ;PUT 1024 AT LOCATION 8
            Data    0
            Data    12
            Data    9
            Data    18
            Data    12
            LDX     2,7          ;X2 GETS 3
            LDR     3,0,10       ;R3 GETS 12
            LDR     2,2,10       ;R2 GETS 12
            LDR     1,2,10,1     ;R1 GETS 18
            LDA     0,0,0        ;R0 GETS 0 to set CONDITION CODE
            LDX     1,8          ;X1 GETS 1024
            JZ      0,1,0        ;JUMP TO End IF R0 = 0
            LOC     1024
End:        HLT                         ;STOP
```

Figure 3 Sample Assembler Source File

Source File Structure/Content

A source file for our assembler contains lines of code broken into 4 possible fields. These fields are:

1. Label – text followed by a colon ":" . This field is used to define locations. Note that the value of the label is its location as we will see below
2. Operation Code (Op Code) or Assembler Directive. Op codes are provided in the above sections on the ISA. Assembler Directives are as follows:
    - LOC n - This tells the assembler to set the load location to the number specified by n which provides a means for the programmer to specify where the instructions are placed in memory. **Note that n is provided by the programmer in DECIMAL.**
    - Data – This tells the assembler to allocate 1 word (16 bits) of memory at the location (kept by the assembler) and place at that location the value specified by a **DECIMAL** number n or the location of a label. Note that in the fourth line we see "Data End" which instructs the assembler to allocate a word and put the location of the label "End" into that location. You may choose to allow text in parentheses, but this is not required. If you provide this, only two characters can be specified per line.

3. Instruction code operands. These vary by instruction as provided in the ISA description section.
4. Comments – a semicolon followed by text describing the program operation.

Note in the example source file that fields may be optional, depending upon the use of the instruction.

Listing Output File

Figure 4 Assembler Listing Output is a listing output file for Figure 3 Sample Assembler Source File above. Note that the assembler has added two columns, both in OCTAL, (000012 is the 16 bit octal representation for decimal 10).

```
                        LOC    6            ;BEGIN AT LOCATION 6
000006    000012        Data   10           ;PUT 10 AT LOCATION 6
000007    000003        Data   3            ;PUT 3 AT LOCATION 7
000010    002000        Data   End          ;PUT 1024 AT LOCATION
000011    000000        Data   0
000012    000014        Data   12
000013    000011        Data   9
000014    000022        Data   18
000015    000014        Data   12
000016    102207        LDX    2,7          ; X2 GETS 3
000017    003412        LDR    3,0,10       ;R3 GETS 12
000020    003212        LDR    2,2,10       ;R2 GETS 12
000021    002652        LDR    1,2,10,1     ;R1 GETS 18
000022    006000        LDA    0,0,0        ;R0 GETS 0
000023    102110        LDX    1,8          ;X1 GETS 1024
000024    020100        JZ     0,1,0        ;JUMP TO End if R0 = 0
                        LOC    1024
002000    000000 End:   HLT                 ;STOP
```

Figure 4 Assembler Listing Output

Note the following:

- The LOC directive is a message to the assembler to begin counting the location at the number provided. The number following LOC is in decimal. Note that it will get translated to Octal on the listing. The LOC **does not allocate memory.**

- The location of End (a label) ends up in location 8 (000010). **This means that the assembler must determine label addresses.**
- **Note that no code is generation for the LOC or commented locations.** If you put 0's in these spaces the machine would halt.

Load File

A load file for the code is shown in Figure 5 Load File. Note that it does not have blank spaces.

```
000006    000012
000007    000003
000010    002000
000011    000000
000012    000014
000013    000011
000014    000022
000015    000014
000016    102207
000017    003412
000020    003212
000021    002652
000022    006000
000023    102110
000024    020100
002000    000000
```

Figure 5 Load File

In the implementation of the load file, this file will be simulated as a text file rather than a binary file. Only non-blank lines should be loaded.

Building the Assembler

The following hints are provided for building the assembler.

Use two passes

Eary assemblers read the program twice.

Pass 1:

1. Set code location to 0
2. Read a line of the file
3. Use the split command to break the line into its parts
4. Process the line, if it is a label, add the label to a dictionary with the code location. Process the rest of the line (it could be blank, if so no code is generated). Check for errors in the code.
5. If code or data was generated increment the code location and go to step 2 until termination.

Pass 2:

1. Set code location to 0
2. Read a line of the file
3. Use the split command to break the line into it parts
4. Convert the code according to the second field.
5. Add line to listing file and to load file.
6. If code or data generated, increment the code counter, and go to step2 until termination.

A two pass assembler is a simpler form from the standpoint of tracing errors and separating functionality.