The Minimal Instruction Set Computer (MISC)

Key words: general computer systems organization, software programming techniques, general simulation and modeling

Abstract: The Minimal Instruction Set Computer (MISC) is a simulation of a simple, fictitious microprocessor. This simulation is written in C++ and does not represent the features of any particular architecture. It serves as an introduction to general principles of computer organization and machine language programming. The simulation is designed so that the development of the program can be given as a student assignment. Various methods and other pieces of code are grouped together in a logical order for sequential implementation. This provides high level language programming practice as well as an introduction to machine architecture. Both the machine specifications and the assignments that can be based on them are presented in detail.

Introduction

The Minimal Instruction Set Computer (MISC) is a simulation of a simple machine architecture written in the C++ programming language. It is not a tool for experimenting with features of machine design—but rather a tool for becoming familiar with general hardware concepts and machine language programming. It is not intended to be given to students in finished form, but to be written by them in a
series of programming assignments. In this respect it is also a tool for gaining experience in programming beyond the standard sequence of introductory courses.

MISC was developed as part of an upper level course in computer architecture. It assumes that students have already had at least one course in C++ programming. It also assumes that students have been exposed to the basic concepts of processor architecture and some assembly language programming. This exposure may come from a prior course, or it may be part of the introductory material in the course in which MISC is used.

To give a fuller context for MISC, the reference list for this paper consists of textbooks that might be used in conjunction with a course in which MISC is included. Two general books on computer architecture are listed. [7, 8] One of these might serve as the text for the course. Specific background on microprocessor architecture and machine language programming is also available in several books. [1, 3, 5, 6] Selected parts of these might be used as supporting material. Finally, the references also include 2 representative books on C++ programming. [2, 4] These would not be used as texts, but as references in writing the simulation code. Whatever book a student had used in a prior programming course should also serve this purpose.

MISC forms an extremely simplified and incomplete representation of a register-oriented microprocessor architecture. No attempt is made to accurately model any particular architecture, and detail is intentionally avoided where possible. On the other hand, MISC was developed with a loose knowledge of the Intel 8088 in mind. Thus, general background in that architecture may be helpful in understanding some of the design decisions in MISC.
This paper falls into two major parts: 1. Machine Language Specifications. 2. Programming Assignment Specifications. The machine language specifications give complete background on the features of the fictitious machine to be simulated. Although written in a formal style, this paper gives a basic exposition, including some explanations at a level lower than necessary for the academic reader. This is because it is also intended to serve as a reference manual for students who make use of MISC.

The programming assignment specifications explain how the task of writing the simulation can be broken up into a sequence of distinct parts. The completed code for the simulation is provided on the Computer Science Teaching Center (CSTC) Web site, http://www.cstc.org, under the Laboratories heading. In it, the various assignments and their parts are clearly marked. To be used, all of the code for the assignments should be removed and the student should be given the shell of the program as a starting point. This shell would consist of some implemented routines and an indication of methods or other segments of code which must be written for the particular assignment under consideration at a given time.

1 Machine and Machine Language Specifications

1.1 The Basic Architecture

The simulation includes the definitions of hardware features like the registers and memory segments and those functions necessary for the machine to interact with the outside world by means of the simulated operating system. The operating system essentially allows you to load a machine language program file, turn execution over to the machine, and dump the contents of the machine after a run.
The machine under consideration has some similarities with the Intel 8088. It is by no means a copy of that, but some of the ideas used come from the architecture and assembly language of that chip. This simulation is far from a complete implementation of any chip architecture, and many of the design decisions in the simulation are aimed at simplicity of understanding and implementation. To the extent possible, those real-life considerations that lead to a general purpose chip architecture have been intentionally ignored.

The machine has 17 registers and 5 distinct memory segments. The registers are 8 bits and the memory segments are 256 bytes. In the simulation, the contents of a register as well as the contents of a byte in memory are modeled by a character array containing 8 bytes. Each bit is then modeled by the presence of either the character ‘1’ or the character ‘0’ in a particular position in one of these 8 byte arrays.

The registers are packaged together in an array named “reg”. The index of the array identifies the particular register. Here are the common, descriptive names of the registers and their designation as elements of the register array. This is followed by the corresponding binary value of the index, shown as the contents of an 8 byte array, emphasizing the representation used in the machine simulation.

<table>
<thead>
<tr>
<th>register name</th>
<th>decimal index</th>
<th>binary code</th>
</tr>
</thead>
<tbody>
<tr>
<td>identification in reg array</td>
<td>of index</td>
<td></td>
</tr>
</tbody>
</table>

| unused | reg[0] | "00000000" |

general
<table>
<thead>
<tr>
<th>Purpose</th>
<th>Register</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>reg[1]</td>
<td>&quot;00000001&quot;</td>
</tr>
<tr>
<td>B</td>
<td>reg[2]</td>
<td>&quot;00000010&quot;</td>
</tr>
<tr>
<td>C</td>
<td>reg[3]</td>
<td>&quot;00000011&quot;</td>
</tr>
<tr>
<td>D</td>
<td>reg[4]</td>
<td>&quot;00000100&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Offsets</th>
<th>Register</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Offset</td>
<td>reg[5]</td>
<td>&quot;00000101&quot;</td>
</tr>
<tr>
<td>Data Offset</td>
<td>reg[6]</td>
<td>&quot;00000110&quot;</td>
</tr>
<tr>
<td>Stack Offset</td>
<td>reg[7]</td>
<td>&quot;00000111&quot;</td>
</tr>
<tr>
<td>Gen Purp Offset</td>
<td>reg[8]</td>
<td>&quot;00001000&quot;</td>
</tr>
<tr>
<td>Output Offset</td>
<td>reg[9]</td>
<td>&quot;00001001&quot;</td>
</tr>
<tr>
<td>Flag</td>
<td>reg[10]</td>
<td>&quot;00001010&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control Unit Registers</th>
<th>Register</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand 1</td>
<td>reg[12]</td>
<td>&quot;00001100&quot;</td>
</tr>
<tr>
<td>Operand 2</td>
<td>reg[13]</td>
<td>&quot;00001101&quot;</td>
</tr>
</tbody>
</table>
extra reg[14] "00001110"

ALU registers

aluinreg1 reg[15] "00001111"
aluinreg2 reg[16] "00010000"
aluoutreg reg[17] "00010001"

These are the five memory segments, each of which is itself an array of 256 such 8 byte arrays.
codesegment
datasegment
stacksegment
genpurpsegment
outputsegment

This simulation does not implement a full set of machine instructions, only a subset. Instructions for moving data, doing simple arithmetic on data, and jumping from one instruction to another during program execution are enough to be able to do simple loops that count, multiply, etc. Provision is made in the machine specifications to allow the use of memory variables as well as registers. The implementation of the machine instructions has to include the use of constants, registers, and memory locations as possible operands.
This simulation uses the convention that in a line of code, the instruction comes first, followed by the
destination operand, followed by the source operand, followed by an extra space which does not yet
have a designated use. If the line of code contains a data declaration rather than an instruction, only two
bytes are needed in order to contain this information: a memory offset and an initial value.

The general rules for both move and arithmetic instructions are these: 1. A register or a memory
variable can be a destination. 2. A constant, a register, or a memory variable can be a source. 3.
Memory to memory operations are not allowed. These simple rules have a great effect on the nature of
the machine language. Because everything in machine language is represented by simple binary codes,
there is no way to distinguish between a constant, a register, or a memory offset. The result is that there
have to be separate machine instructions for every possible legal combination of operands. In other
words, the machine language programmer distinguishes the operands by means of the choice of
instructions. In contrast, in assembly language, the operands would be distinguished by naming
conventions, and it is possible to have a single mnemonic for MOVE, ADD, SUB, etc.

One other important aspect of the simulation has to be taken into account. The machine should have the
capability of using both positive and negative numbers. 2’s complement representation of integers is
preferable for handling positive and negative values in a simple machine. In effect, one bit out of 8 is
used for the sign, leaving only 7 for the value itself. This leads to the distinction between signed and
unsigned integers and arithmetic. Observe that offset values are always positive and the machine
supports 256 offsets into each memory segment. This means that all 8 bits have to be used to contain an
offset, and when arithmetic is done on offset values, it is purely binary arithmetic on positive numbers.
The results have to be interpreted accordingly, especially when considering the meaning of overflows,
when the sum of two numbers has more bits than can be stored in a register.
The use of 2’s complement arithmetic also has a serious effect on anyone trying to program in machine language. A programmer will have to convert all numeric values to 2’s complement notation by hand when writing machine language code, and remember to treat offsets as unsigned numbers. One of the features of 2’s complement notation has already been mentioned: it’s a scheme whereby a 1 in the high order position of the representation signals a negative value. The three big advantages of 2’s complement notation are: 1) There is only one representation of the number zero. 2) Subtraction does not have to be implemented as a separate operation; it can be implemented as adding the negative of the value to be subtracted. 3) Also, if a positive value is incremented one unit beyond the largest representable positive value, the result is the negative value of greatest magnitude that can be represented in the machine; and when the value –1 is incremented, when the carry bit is ignored, the remaining 8 bits simply contain the value 0. In other words, incrementation and decrementation simply cause rollover when the extreme values in the representation are reached.

In 2’s complement notation positive integer values are represented as straight binary numbers—with the provision that the high order bit for positive numbers is always 0. This is the equivalent of a plus sign. The representation of negative values is formed in this way: Take the full 8 bit representation of the corresponding positive value. Form the 1’s complement of this. That means flip all of the 1 bits to 0 and all of the 0 bits to 1. Now add 1 using regular binary addition. This procedure always results in a 1 in the high order bit. Observe that if you try to convert the value 0, 00000000, to a negative representation, when you flip all of the bits you get 11111111, and adding 1 gives 100000000. This however, is a 9 bit number. The 1 is a “carry” that is automatically lost from the remaining 8 bits of the representation. In other words, although the representation for the value zero has an initial 0 bit, there is no negative representation of it. The 8 bit 2’s complement number 10000000 represents the decimal
value –128. Overflows and whether or not the results of an operation were positive or negative are recorded in various bits of the flag register. These outcomes are used when doing comparisons for JUMP instructions.

1.2 The Flag Register and Arithmetic Operations

In order to do comparisons and jumps it is necessary to store information about the outcome of the most recent arithmetic operation in the flag register. This information may also be helpful for showing out-of-range conditions when debugging a program. Although there is a conceptual distinction between signed and unsigned integers, in the simulation at the adder level there is no distinction. In other words, whichever kind of number is put in, the adder will simply do addition. The machine doesn’t “know” whether rollover from positive to negative or negative to positive on signed numbers signals an out-of-range condition. Only the programmer knows whether values were supposed to be signed or unsigned, and thus what rollover might signal, and whether the results from the adder are arithmetically correct. The programmer can determine what happened because the outcome of an arithmetic operation, positive, negative, or zero, is stored in the flag register. The fact that there was a carry when the high order bits of the adder registers were added is also recorded. This signals out-of-range conditions for unsigned integers.

The 3 possible outcomes of an arithmetic operation are 0, positive, or negative. A single bit position can only record 2 possible states, so 2 bits out of the flag register will have to be reserved for the outcome. Let those 2 bits be positions 0 and 1 in the register. Each arithmetic operation has to set those 2 bits. Let the following codes be used:
flag[0]  flag[1]

0  0  The outcome of the operation was 0
1  0  The outcome of the operation was > 0
0  1  The outcome of the operation was < 0
1  1  This combination is unused

Let a separate bit position in the flag register be used for signalling overflow. If flag[2] contains a 0, that indicates there was no overflow. If it contains a 1, that signals an overflow. For the time being, the other 5 bits of the flag register are unused.

The simulation is provided with aluinreg1, aluinreg2, and aluoutreg. These are reg[15], reg[16], and reg[17]. These are the arithmetic/logic unit registers. When arithmetic is to be performed, the operands should be copied into these “in” registers and performed there. The result, which is placed in the “out” register, should then be copied to the destination register. Arithmetic operations in the simulation have been implemented as binary algorithms. However, it should be noted that at various stages internal to the simulation it is necessary to work with the contents of registers and there is no need to do all of this using binary arithmetic. In order to make things like incrementation of the register containing the offset into the code segment relatively easy and transparent, the incrementarray() method is used. It, in turn, uses the inttoarray() and arraytoint() methods to convert an 8 bit binary array representation to a decimal, add a 1, and then convert the result back into an array representation. Doing this kind of operation on register contents is a convenience in writing the simulation, but the actual machine arithmetic operations are not implemented in this way.
1.3 Machine Instruction Execution

When the machine takes control of execution (by means of the takecontrol() method called from the Osystem), it steps through the contents of the code segment until it encounters an empty (“00000000”) instruction byte. It starts with the value 0 in the code offset register and takes the contents of 4 contiguous bytes of code memory and puts them into the instruction, operand1, operand2, and extra registers, reg[11], reg[12], reg[13], and reg[14], respectively. After the retrieval of each instruction and before its execution, the code offset is incremented by 4 for the next retrieval. It should be emphasized that it is a design decision to make incrementation of the code offset a part of takecontrol(). Stepping through program code is in effect an “automatic” feature of the system. It would be possible to implement differently by putting suitable offset incrementation into the implementation of each individual executable instruction. Because incrementation is done in takecontrol(), it is critical that it be done before the execution of the instruction retrieved. If the instruction were a jump, for example, it would place a new value in the code offset register, the address of the instruction to jump to. If automatic incrementation were done after the execution of the instruction, it would increment the address to be jumped to, which would be incorrect.

Not only does the simulation automatically step through instructions in the code segment when a program is being run; it is also built on the assumption that access to any memory segment during a program run is sequential. Whatever offset is currently in the offset register for the segment of interest is taken as the desired offset. When the method accessing memory is executed, the offset register for that segment is incremented. This simplifies loading data and programs for example. It also has important implications for machine language programs, which do not necessarily access memory in
sequential order, but access memory variables at specific memory offsets. There is no such thing as a machine language command that makes use of a memory variable and also takes in the offset of that variable as a parameter. The commands are generic—the variable they affect is the one that currently has its offset in the offset register. An instruction affecting some specific offset in a memory segment has to be preceded by a move putting the desired offset into the offset register for that segment. Although not necessarily easy, this requirement is straightforward in machine language. It should also be noted that this design pretty much rules out self-modifying code. There is only one code offset register. At run time it is incremented automatically or by jump statements to hold the offset of the next instruction to execute. Instructions which set the offset register in order to move new instructions into the code segment would cause out of order execution.

1.4 The MOVE Instruction

Moves with constants as destinations are not legal and moves with two memory variables as operands are not legal. Each legal combination of parameters gets its own machine instruction. The possible legal combinations, the prototypes of the methods as they are implemented, and their binary instruction codes are given below. A move instruction does not affect the flag register. It is not necessary to use the ALU registers when moving. A move instruction involving a memory variable should be preceded by a move instruction that places into the offset register the variable’s offset in the data segment. Here are the commands given in pseudo-assembler, the corresponding method names in the MISC simulation, and the binary code representing them in machine language.

MOVE register, register
void movedestregsrcreg(); "10000001"

MOVE memory, register

void movetomemfromreg(); "10000010"

MOVE register, memory

void movetoregfrommem(); "10000011"

MOVE memory, constant

void movetomemfromconst(); "10000100"

MOVE register, constant

void movetoregfromconst(); "10000101"

1.5 The ADD Instruction

The presentation of the material below follows the pattern for move instructions. The contents of the source operand are not changed by the operation. The contents of the destination operand are replaced with the results. Each add instruction sets the first 3 bits of the flag register depending on the outcome of the instruction. Values are placed in the ALU registers and all arithmetic algorithms are implemented using values stored in them. The results are then copied from aluoutreg to the correct destination. Addition of signed and unsigned integers can lead to rollover and overflow, respectively, but it’s up to the programmer to watch for these outcomes. An add instruction involving a memory variable should
be preceded by a move instruction that places into the offset register the variable’s offset in the data segment. Here are the commands given in pseudo-assembler, the corresponding method names in the MISC simulation, and the binary code representing them in machine language.

ADD register, register
void adddestregsrcreg();  “10000110”

ADD memory, register
void addtomemfromreg();  “10000111”

ADD register, memory
void addtoregfrommem();  “10001000”

ADD memory, constant
void addtomemfromconst(); “10001001”

ADD register, constant
void addtoregfromconst(); “10001010”

1.6 The SUB Instruction

The presentation of the material below follows the pattern for add instructions. The contents of the source operand are subtracted from the destination operand. This is accomplished by converting the
second operand to its negative representation in 2’s complement, and adding it to the first operand. The
source operand itself is unchanged because the 2’s complement conversion only appears in the aluinreg1
register. The contents of the destination operand are replaced with the results. Each sub instruction sets
the first 3 bits of the flag register depending on the outcome of the instruction. Values are placed in the
ALU registers and all arithmetic algorithms are implemented using values stored in them. The results
are then copied from aluoutreg to the correct destination. Subtraction of signed integers can lead to
overflow. Subtraction of unsigned integers is problematic. If the result is a negative number, strictly
speaking, what is represented in the machine is neither the result of overflow or rollover—it is simply
out of type. The programmer can’t rely on automatic type conversion between unsigned and signed
integers, and should take care to avoid doing the wrong kind of arithmetic on the wrong types. A sub
instruction involving a memory variable should be preceded by a move instruction that places into the
offset register the variable’s offset in the data segment. Here are the commands given in pseudo-
assembler, the corresponding method names in the MISC simulation, and the binary code representing
them in machine language.

SUB register, register
void subdestregsrcreg();  “10001011”

SUB memory, register
void subfrommemsrcreg();  “10001100”

SUB register, memory
void subfromregsrmem();  “10001101”
SUB memory, constant

void subfrommemsrcconst(); “10001110”

SUB register, constant

void subfromregsrcconst(); “10001111”

1.7 The JUMP Instruction

Strictly speaking, the jump is not an arithmetic instruction. It is a control instruction. In its simplest form, the jump puts the offset of a program instruction into the code offset register. It is important for the machine language programmer to make sure to count by 4’s when figuring out the jump offset. There is an unconditional jump which simply changes the program execution path. In addition, there are conditional jumps which depend on the current contents of the flag register. These instructionns have a single parameter or operand, which is an unsigned integer value placed in the operand1 register. This is interpreted as an offset into code memory. It has to be treated as unsigned, and in order to work correctly it has to fall on a 4 byte instruction boundary. In code memory every line consists of an instruction, two operands, plus the extra slot, even if one or more of the operands is empty. The offsets of the machine instructions will always have offsets which are multiples of 4. Here are the commands given in pseudo-assembler, the corresponding method names in the MISC simulation, and the binary code representing them in machine language.

JMP unsigned integer

void jumpunconditional(); “10010000”
JPOS unsigned integer
void jumponpositive();  “10010001”

JNEG unsigned integer
void jumponnegative();  “10010010”

JZERO unsigned integer
void jumponzero();  “10010011”

JOVER unsigned integer
void jumponoverflow();  “10010100”

1.8 Handling the Data Segment in Code

In machine language programs, data will come in pairs of bytes, the first being an offset into data memory and the second being a value. The runprogfile() routine will interpret the data portion of a program in this way and load only the values at the given offsets in data memory. What this means is that the operating system will loop twice, filling arrays byteholder1[] and byteholder2[]. The inputinstruction() method is then used, first to load the data segment offset, and then to load the value into that data segment offset. The data portion of the program file should come first and then the program code itself. This will make it easier to write and understand machine code because the offsets identifying memory variables will be declared before the appearance of any code referring to them.
1.9 Running the Simulation and Using the Operating System Commands

The C++ program file for the simulation is miscsimulation.cpp. The bulk of the code is in the header file miscsimulation.h. The program file can be compiled and run. When it is running, it presents a simple operating system prompt. The operating system has only 3 commands:

**rpf** = run program file. Upon entering this command the user is prompted for the name of the program (machine language) file to run. Machine language files have to be simple text files. When prompting for the file name the O/S expects a name with the .txt extension. It will seemingly “accept” files without the extension, but it will not work correctly.

**dmc** = dump memory contents. Upon entering this command the user is prompted for the name of the output file to create. The system will put the contents of the machine into this file after a program run. The operating system in effect has no I/O capabilities. Program results can only be determined by looking at the hardware contents afterwards. The output file specified should also be a text file with a .txt extension.

**exit** = quit the simulation. A file called “showfile” will show up in the directory where the simulation is run. This is simply a debugging tool. It shows the contents of the machine after a run. In fact, it should contain exactly what is in the file produced by the dmc command. In case of multiple runs it probably makes more sense to use dmc and save output in files with distinct names rather than relying on showfile.
1.10 An Example Machine Language Program with Assembly Language Guide

Machine language code is an unbroken string of binary digits and symbols. Shown below for each line of source are the following on separate lines:

1. A possible assembly language version, including directives. This is given in capitals with slashes as punctuation.
2. A verbal rendition of the assembly language version, effectively commenting the assembly language mnemonics.
3. The name of the method in the MISC implementation that handles this instruction, and the parameter values it would receive.
4. Finally, the MISC machine code given in binary digits.

The assembly language is given in order to serve as a guide to the machine language statements, which would otherwise verge on the incomprehensible. The .DATA section of the machine language code ends with a ‘#’ sign. The machine language program overall ends with a ‘*’. This program implements a loop and finds the sum of the first 10 positive integers.

/ DATA //

/ LOOPLIM/ X0B //

loop limit offset 0, value 11
accumulator offset 1, value 0

(move reg D, const 1
movetoregfromconst 4, 1
10000101000001000000000100000000

(Label) move data offset, const 1
movetoregfromconst 6, 1
100001010000011000000000100000000

add data segment, reg D
adddatumfromreg 2, 4
1000011100000011000000001000000000
/ADD/D/X01/
add reg D, 1
addtoregfromconst 4, 1
100010100000010000000000100000000

/MOVE/C/LOOPLIM/
move data offset, const 0
movetoregfromconst 6, 0
10000101000001100000000000000000

move reg C, data segment
movetoregfrommem 3, 2
10000011000000110000001000000000

/SUB/C/D/
sub reg C, reg D
subtractdestregsrsreg 3, 4
10001011000000110000010000000000

/JPOS/LOOPTOP/
jump on positive to “LABEL”
jumponpositive 4
10010001000001000000000000000000
The machine language alone (with artificial line breaks) is as follows:

0000000000001011
0000000100000000#
10000101000001000000000100000000
10000101000001100000000100000000
1000011100000100000001000000000000
10000101000001000000000100000000
10000101000001000000000000000000
10000101000001000000010000000000
1000101100000100000001000000000000
10010001000001000000000000000000*

2 Programming Assignment Specifications

There are 7 assignments in all, each divided into about half a dozen different parts. In most cases, an individual part is a single method in the simulation that needs to be implemented. Assignments 1 through 5 have to do with completing the basic simulation. Assignment 6 involves writing a machine language program for use on the simulation. Assignment 7 involves including a multiplication instruction that is implemented by means of a machine language program loaded into the general
purpose segment and modifications in the machine simulation needed to make use of that code when a multiplication instruction is encountered.

The assignments are spelled out below using the following general format:

1. A general description of the assignment

2. A list of the parts of the assignment by number and the name of the method or the segment of code involved.

2.1 Assignment 1

Write the basic array manipulation methods that will be used in the simulation to work with the contents of registers, etc., which are represented as arrays.

Assignment 1, part 1

int Machine::<
arraycompare(char * array1, char * array2)

Assignment 1, part 2

void Machine::<
arraycopy(char * destarray, char * sourcearray)

Assignment 1, part 3
int Machine::
arraytoint(char * array)

Assignment 1, part 4
void Machine::
inttoarray(int anint, char * bytearray)

Assignment 1, part 5
void Machine::
incrementarray(char * array)

2.2 Assignment 2

Write the basic methods which are used to do machine level I/O and to clear the contents of the machine before or after certain operations.

Assignment 2, part 1
void Machine::
getfromrega(char * bytearray)

Assignment 2, part 2
void Machine::
moveatoop2()
Assignment 2, part 3

void Machine::<br>
resetoffsets()<br>

Assignment 2, part 4

void Machine::<br>
movetoregfrommem()<br>

Assignment 2, part 5

void Osystem::<br>
outputinstruction(char * instructioncode,<br>char * operand1code, char * operand2code,<br>char * bytetoget)<br>

2.3 Assignment 3

Write the methods which implement the MOVE instructions not included in assignment 2, some helper methods needed for implementing arithmetic instructions, and the first 2 ADD instructions.

Assignment 3, part 1

void Machine::<br>
movetomemfromconst()
Assignment 3, part 2

void Machine::
movetoregfromconst()

Assignment 3, part 3

void Machine::
dotheadd()

Assignment 3, part 4

void Machine::
setflags(char overflow)

Assignment 3, part 5

void Machine::
twoscomplement(char * array)

Assignment 3, part 6

void Machine::
adddestregsrcsrcreg()

Assignment 3, part 7

void Machine::
addtomemfromreg()
2.4 Assignment 4

Write the code for the remaining 3 ADD instructions and for all 5 of the SUB instructions.

Assignment 4, part 1

void Machine::

addtoregfrommem()

Assignment 4, part 2

void Machine::

addtomemfromconst()

Assignment 4, part 3

void Machine::

addtoregfromconst()

Assignment 4, part 4

void Machine::

subdestregsrcreg()

Assignment 4, part 5

void Machine::
2.5 Assignment 5

Write the methods for the 5 JUMP instructions. Also write the method takecontrol() in the Machine class and the method runprogfile() in the Osystem class, which allow a machine language program to be taken in and take control of the machine simulation—in other words to be run on the simulation.
Assignment 5, part 2
void Machine::
jumponpositive()

Assignment 5, part 3
void Machine::
jumponnegative()

Assignment 5, part 4
void Machine::
jumponzero()

Assignment 5, part 5
void Machine::
jumponoverflow()

Assignment 5, part 6
void Machine::
takecontrol()

Assignment 5, part 7
void Osystem::
runprogfile()
2.6 Assignment 6

Write and test a machine language program to do multiplication on the simulation. Here are the basic specifications:

1. Let the numbers to be multiplied together be placed in general purpose registers A and B.
2. Write the code so that it will handle all combinations of operands:
   
   (+)(+), (+)(-), (-)(+), and (-)(-).
3. Initialize a memory variable to 0 and store the result there.

The sample test program that accumulated the sum of the first 10 integers is a guide to writing this program. The major difference between that program and this one is that this one will have to include jumps that implement the if's needed to handle the different combinations of + and – in the input.

2.7 Assignment 7

The previous assignment involved writing machine language code that implemented multiplication of two integers placed in general purpose registers A and B. The purpose of this assignment is to implement integer multiplication as a special purpose machine instruction that is executed in software rather than hardware. Nothing in particular about this simulation or the whole set of assignments is representative of a real machine or architecture, and this assignment is no different. The specifications
are designed to be special purpose—simply to make it possible to implement multiplication in software with a minimum amount of trouble. The general purpose segment will be used to hold the code for the routine. The stack segment will be used to save information from the calling program.

Here is an outline of the changes that can be made in order to implement multiplication as a software supported instruction on the machine.

1. At the bottom of the constructor for the Machine add code that will load the machine language multiplication routine from an external file into the general purpose segment. The code to accomplish this can be copied with small changes from the code in the runprogfile() method that loads a user’s program into the code segment.

2. The Machine constructor includes code to clean out all of the registers and all of the memory segments. The method totalreset() which is called by runprogfile() also does that. In order to keep from wiping out the code for multiplication that has been loaded into the general purpose segment at construction time, the code which clears the general purpose segment has to be commented out of or removed from totalreset().

3. The machine language program to do multiplication can be modified for system use. Here is a list of possible changes you may need to make.

A. Get rid of the data declaration. Assume that the space in the data segment is available for use, and take care that this assumption is not violated.
B. Get rid of the code that puts operands in the A and B registers. The system multiplication routine will be used under the assumption that the code that calls it has already placed the operands in those registers before making the call.

C. Add code at the beginning that will explicitly zero out the contents of offset 0 in the data segment so that it can be used as a blank variable.

D. By adding and removing code it is likely that the hardcoded values for the LABELs in the JUMP statements are no longer right and will have to be changed.

E. Add code at the end that will put the result into the A register.

4. Assign the multiplication instruction the binary identification code “11000000” and add code to the takecontrol() method to detect this instruction and call the method multdestasrcb() in order to accomplish it.

5. Observe that the code for takecontrol() would be virtually unchanged whether trying to run a user program or trying to run a system program for multiplying. Make minor changes to takecontrol() so it can be called out of the operating system or by the multiplication method. Let it have a single parameter that signals whether it’s supposed to run code from the code segment or the general purpose segment. Rewrite the method so that there are local variables for an offset register and segment. Depending on the value of the incoming parameter, assign to these local variables either the values for the code segment offset and register or the general purpose segment offset and register. Make sure that a suitable parameter is sent in the places where takecontrol() is called and update the prototype.
6. The fact that code can be run either from the code segment or the general purpose segment also leads to changes in the jump instructions. They are hardcoded now to copy an operand into the code offset register. The jump methods will have to be rewritten with a signal parameter like the takecontrol() method. Whenever a jump method is called from takecontrol(), it has to be passed the same signal that takecontrol() received. If the signal indicates that the code is in the code segment, then the jump offset should be moved to the code offset register. If, on the other hand, the signal indicates that the code is in the general purpose segment, then the jump address should be moved to the general purpose offset register.

7. Write the new method multdestasrcb(). The basic things that have to happen in the method are:

   A. Save the contents of every register on the stack. Also store the contents of offset 0 of the data segment on the stack. You could use the stack offset register to keep track of where things are being saved. This would be for the best, but the code would be much simpler if that were register 0. Rather than rewriting the code, it is possible here to make use of the “unused” register 0 for this purpose.

   B. Call takecontrol(), sending as parameters the general purpose offset register and segment.

   C. Restore the former contents of the registers except for the contents of the A register, which should contain the results of the multiplication. It is a convenience here to add a new static method, decrementarray(), in order to be able to conveniently step back through the sequence of offsets where things were saved. Feel free to add this if you want to.

8. The final task is testing all of the changes to the simulation by writing and running a machine language program that makes use of the new multiplication instruction.
Assignment 7, part 1

Prototype the new and modified methods for this assignment.

Assignment 7, part 2

In the Machine constructor, declare the file stuff needed to take in the multiplication code.

Assignment 7, part 3

In the Machine constructor, load multiplication machine code into the general purpose segment.

Assignment 7, part 4

void Machine::

decrementarray(char * array)

Assignment 7, part 5

In totalreset(), comment out the spot where the general purpose segment is wiped clean.

Assignment 7, part 6

void Machine::

multdestasrcb()

Assignment 7, part 7

void Machine::
takecontrol(int signal)

References


