Assembler using MATLAB

VISHNU DIVAKARAN PILLAI

Sheffield Hallam University

MSc Electrical and Electronic Engineering
c1016122@my.shu.ac.uk 31016122

Abstract— In this report, the assembler '.asm' file is converted to a .hack file. After a thorough analysis, the required tools and techniques are selected accordingly. MATLAB is used to convert the .asm file, here. For this conversion, firstly the assembly file is imported to the MATLAB platform, and later parsing of the data is performed to structure the data. Then the mnemonics are created after removing comments, empty lines, and whitespace characters from the code. Then a symbol table is created and the hack assembly file is arranged. Once the hack code conversion starts and the mnemonics are converted to machine codes. The generated hack file is then executed in an assembler to evaluate the conversion by comparing the result of the hack file with that of the '.asm' file.

Keywords— ***MATLAB, Assembler, Hack File.***

Introduction

The assembler is the first and most fundamental module in the software hierarchy. An assembler is a particular kind of computer program that converts a software program that is written in assembly language by humans into a machine language, thus the assembler accepts the basic computer commands and instructions from the assembly code and then translates codes into binary codes which can be executed by a computer. In order for an assembly program to be run on a computer, it must first be interpreted into the binary machine language used by the computer. The translating work is performed by a program called the assembler. It accepts a series of assembly commands as input and creates as output binary instructions that are equal to the assembly commands. The generated code may be loaded into the computer's memory, and hardware can execute it. An assembler is indeed known as an assembly language compiler. It moreover offers a translation service. Any programming language may be used to write the assembler.

Need of assembler

A computer program's most straightforward definition is a sequence of numbers that encode the basic CPU operations. This is known as machine code. Machine code applies to specific CPU manufacturers and, in many cases, a certain model type. Because machine code is very complex for people to understand or write, the lowest level of programming carried out by humans is in assembly language, a language in which each fundamental action is assigned a mnemonic code. Assembly language is a kind of programming language that humans can read and write, and it can be transformed into machine code through the use of an assembler. Assemblers are used to connect symbolic, coded instructions written in assembly language to the computer processor, memory, and other computational components. The primary function is to convert symbols into binary code. The symbol table allows the program to maintain user-defined symbols and recover them to physical memory addresses. Hack Assembler is a simplistic and straightforward illustration of the core software engineering ideas that go into creating an assembler. Assemblers are needed because it increases the efficiency of the programmer, if we don't use the assembler, the programmer will have to write the instruction in machine-level language, which is only zeros and ones, and this takes a lot of time and is very complex to proceed.

Selection of the language to write Assembler code

MATLAB is an acronym for matrix laboratory. MATLAB is a high-performance programming language for scientific and engineering applications. The process of changing the assembly file to a hack file will be carried out using the MATLAB coding language.

The program may be used as a scratchpad to test expressions written into the command line, or it can be used to run massive predefined programs in a single operation. Applications may be produced and updated using the integrated development environment (IDE), and they can be debugged using the MATLAB debugger (which is included). It is ideal for the rapid prototyping of new apps due to the fact that it is easy to learn and utilize. A large number of software development tools are provided, making the application simple to use. In addition, they provide an integrated editor and debugger, online documentation and guides, a workspace browser, and a large number of demonstrations. Its command-line interface and file-oriented structure make it simple to use. The MATLAB Software platform, which enables you to interact with your data in real-time, assists you in keeping track of files and variables and simplifies typical programming and debugging activities.
The MATLAB software includes a vast library of ready-to-use routines for a wide variety of applications, allowing the user to solve technical computer issues considerably more quickly than they could with conventional programming languages such as C++ and FORTRAN [1].

Background

Assembler and compiler both create executable code. The sole difference between assemblers and compilers is that assemblers only translate low-level code to machine code. Assembly languages are low-level programming languages that are grouped along with other programming languages. When we get to this level of abstraction, we can write our code using an easier-to-read set of instructions that are represented by mnemonics rather than huge numbers. An Assembler is used to translate assembly code into machine code. The set of instructions used by a given processor family is unique. For your system's CPU family, the assembler you choose must be able to produce instruction codes [2]. The assembler directives are the most significant distinction between assemblers. Although opcode mnemonics are similar to processor instruction codes, assembler directives are specific to each assembler. Despite the fact that current operating systems are largely written in high-level languages, certain programs can only be written in assembly. Writing device drivers, saving the state of a running program so that another application may utilize the CPU, recovering the stored state of a running program so that it can continue execution, and controlling memory and memory protection devices are all common uses of assembly language. Assembly language is now largely used to manipulate hardware directly, to get access to specialized processor instructions, and to resolve key performance concerns. Device drivers, low-level embedded systems, and real-time systems are all examples of typical applications. Assembly is the one language that can interact directly with a machine or computer. It is the language that a particular CPU understands, and various CPUs understand different forms of it. Assembly language is plain and clear as compared to high-level languages, which are largely in the form of abstract data structures. Assembly language is the closest you can go to a machine's CPU as a programmer. You may use this to write code that reads registers and even directly works with memory addresses to get data and pointers.

Methodology

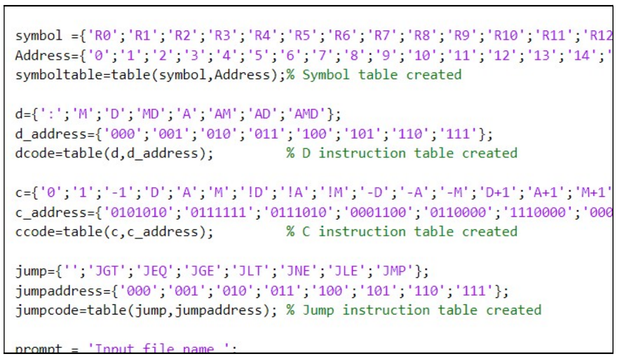
To achieve the desired result, the .asm file is converted to the .hack file in the following steps— first, parsing of the data is performed on the .asm program. The parsing technique consists of analyzing the stored symbols in a string. This process converts the unstructured data into a structured format like CSV or JSON. This technique of parsing the data verifies the rule of formal grammar. The data parsing technique is essential to detect the symbols and the words within the .asm file. All the empty spaces, comments are removed to be easily detected by the MATLAB code for converting the .asm file to. hack format. Otherwise, errors would occur in the binary values of the .hack file based on the white spaces and comments.
Next, it is necessary to check for any command remaining in the .asm file. And if there is any command detected then that command type is found and the code is being executed again by the data parsing technique to preprocess the data. In the next step, the code is written to convert the commands to their equivalent binary values.
Then to differentiate the variables and the reserved words, the mnemonics are created. Now the hack converter code is written in a way that the conversion of mnemonics to the machine codes takes place and then for the mnemonics the separate tables are created. Thus, all the mnemonics codes are converted to equivalent binary values.
Accordingly, a hack symbol table is created. Subsequently, the arrangement of the hack assembly table and parsing is performed. Then with the help of an input file parser is created. Now, the input file is scanned for labels and the file parser is begun again. Thereafter the binary code is commenced resulting in the conversion of the .asm file to a .hack file.

.

Discussion

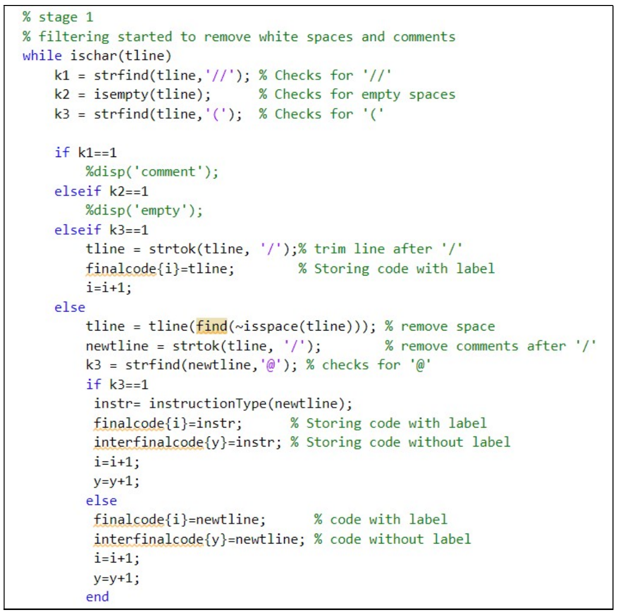
Any high-performing language can be used to design a HACK assembler, but in this report, the Hack assembler is designed using MATLAB. And the same is explained further. The HACK assembler is generated in two stages: with and without symbols. This program I have separated

into 6 stages. In the first stage creation of the symbol table is then remove any white spaces and comments are removed from the test code, making the process easier for the design to be developed later 'A - instructions' and the 'C - instructions' are separated first. Then, any 'A' instructions are checked with decimal digits and are directly converted to equivalent 16-bit binary. Further, each 'C' instruction is checked, and each field is assigned with a digit, and then that digit is converted into a 16-bit binary code. Subsequently, an output file is generated with 16-bit instructions. Secondly, the assembler design is developed with symbols. For the process to execute correctly, a library is created consisting of predefined tables
pre-defined C instruction Table,
pre-defined valued for D instruction Table,
pre-defined value for Jump instruction Table,
and symbol Table for all others.



```
symbol ={'R0';'R1';'R2';'R3';'R4';'R5';'R6';'R7';'R8';'R9';'R10';'R11';'R12
Address={'0';'1';'2';'3';'4';'5';'6';'7';'8';'9';'10';'11';'12';'13';'14';'
symboltable=table(symbol,Address);% Symbol table created

d={':';'M';'D';'MD';'A';'AM';'AD';'AMD'};
d_address={'000';'001';'010';'011';'100';'101';'110';'111'};
dcode=table(d,d_address);         % D instruction table created

c={'0';'1';'-1';'D';'A';'M';'!D';'!A';'!M';'-D';'-A';'-M';'D+1';'A+1';'M+1'
c_address={'0101010';'0111111';'0111010';'0001100';'0110000';'1110000';'000
ccode=table(c,c_address);         % C instruction table created

jump={'';'JGT';'JEQ';'JGE';'JLT';'JNE';'JLE';'JMP'};
jumpaddress={'000';'001';'010';'011';'100';'101';'110';'111'};
jumpcode=table(jump,jumpaddress); % Jump instruction table created

prompt = 'Input file name ';
```

Figure 1 Predefined instruction Table

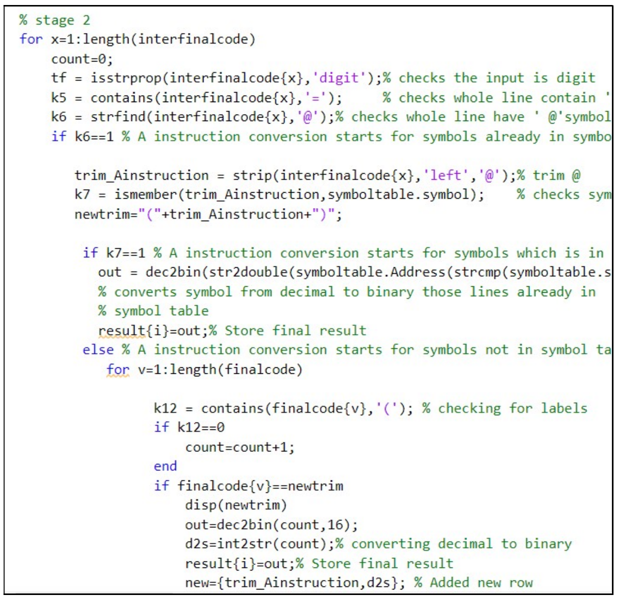


```
 % stage 1
 % filtering started to remove white spaces and comments
 while ischar(tline)
     k1 = strfind(tline,'//'); % Checks for '//'
     k2 = isempty(tline);      % Checks for empty spaces
     k3 = strfind(tline,'(');  % Checks for '('

     if k1==1
         %disp('comment');
     elseif k2==1
         %disp('empty');
     elseif k3==1
         tline = strtok(tline, '/');% trim line after '/'
         finalcode{i}=tline;        % Storing code with label
         i=i+1;
     else
         tline = tline(find(~isspace(tline))); % remove space
         newtline = strtok(tline, '/');        % remove comments after '/'
         k3 = strfind(newtline,'@'); % checks for '@'
         if k3==1
          instr= instructionType(newtline);
          finalcode{i}=instr;      % Storing code with label
          interfinalcode{y}=instr; % Storing code without label
          i=i+1;
          y=y+1;
         else
          finalcode{i}=newtline;      % code with label
          interfinalcode{y}=newtline; % code without label
          i=i+1;
          y=y+1;
         end
```

Figure 2 Sortation (Stage 1)

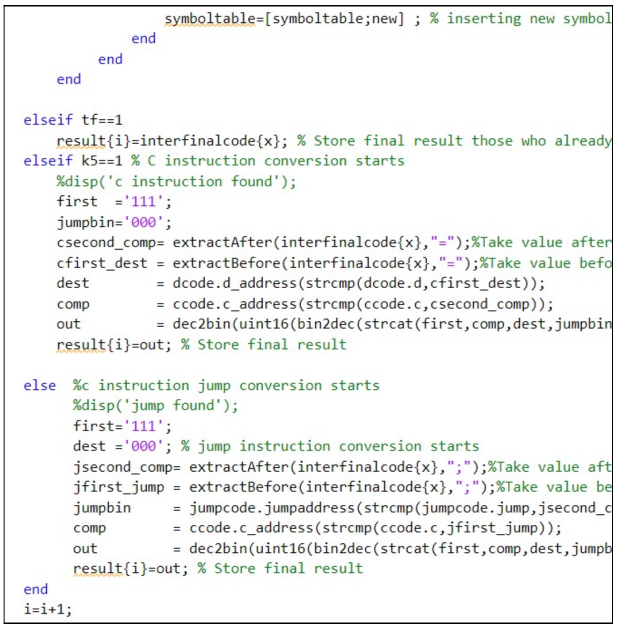
While designing assembly code with symbols, it is necessary to add pre-defined symbols, label symbols, and variable symbols. The pre-defined symbols represent unique memory locations, and the Hack language specification lists 23 of them. Now primarily, it is required to check—if it is an A instruction without any label; if yes, then the decimal value is given to it starting from 16. As shown from the table up to the value of 15, the Registers are already assigned. One example of A instructions is @0, which is directly converted to its equivalent binary value. The second one is @R0 to R15, which is also set with decimal numbers and converted to binary. The next one is @SCREEN or @KBD, which are assigned from 15 onward values.
Another one is checked for the C instruction. The C instruction instructs the compiler to achieve the program's goal. The function of the C instructions is to command the compiler to perform a specific action. For example, the C compiler does not know how to add two numbers unless written in a code or command using the arithmetic operator '+' and numbers. Now when the C instruction is checked here in this design, for example (=), it is assigned with a digit and then converted to the binary value. The C-instruction function takes the instruction as an input and generates a binary code as output concatenated with a comp value, destination value, and jump value. This whole process comes under parsing.
Another part of the design of the assembler is the pre-defined symbol table. Whenever such a situation arises, it checks— the symbols and corresponding values, the symbol names, and a variable that indicates the value of symbols in the array. In this way, Cell Array is created for both with and without symbols.
So, in stage 1 after the creation of Tables, the comments and empty spaces are separated from the given code. And the process for this involves—firstly, ‘prompt' is written in the code which is used to intimate the user to type the input file name. Then this user file name is stored in the 'str'. Next, 'fopen' is operated, which opens the file stored in 'str'. The 'fgetl' read line from the file in the following line, removing newline characters. The 'finalcode' and 'interfinalcode' are used to create two Cell Arrays for copying the command with and without labels.
Furthermore, the 'i' and 'y' loop variables are written in the code for the iteration purpose to store code. The code is first read by the variable 'tline' and checked to see if it contains a comment or not. This is achieved by the command— 'strfind' and 'isempty'. Using strfind command finds the comments in the tline Further, the 'isemplty' command finds the empty spaces in the tline. In the next line, 'strfind' command checks for the label “(“, to find the label symbols in the given code. If we find the label, we will start to store that into cell array finalcode. Then finally it goes to check for else part, where it will determine it’s a C instruction or A instruction. If it’s a C instruction then it will store directly into a cell array. If it’s an instruction it will go to function instructiontype. So, there it will check whether the A instruction is an

integer or not using the function `'isstrprop'.` If it's an integer then we will convert it into binary value and the function will return binary value and it will store it into both cell arrays for final execution. If A-instruction is not in number format the 'instructiontype' function will directly send back without any change.



```
% stage 2
for x=1:length(interfinalcode)
    count=0;
    tf = isstrprop(interfinalcode{x},'digit');% checks the input is digit
    k5 = contains(interfinalcode{x},'=');     % checks whole line contain '
    k6 = strfind(interfinalcode{x},'@');% checks whole line have ' @'symbol
    if k6==1 % A instruction conversion starts for symbols already in symbo

       trim_Ainstruction = strip(interfinalcode{x},'left','@');% trim @
       k7 = ismember(trim_Ainstruction,symboltable.symbol);    % checks sym
       newtrim="("+trim_Ainstruction+")";

        if k7==1 % A instruction conversion starts for symbols which is in
          out = dec2bin(str2double(symboltable.Address(strcmp(symboltable.s
          % converts symbol from decimal to binary those lines already in
          % symbol table
          result{i}=out;% Store final result
        else % A instruction conversion starts for symbols not in symbol ta
           for v=1:length(finalcode)

                 k12 = contains(finalcode{v},'('); % checking for labels
                 if k12==0
                     count=count+1;
                 end
                 if finalcode{v}==newtrim
                     disp(newtrim)
                     out=dec2bin(count,16);
                     d2s=int2str(count);% converting decimal to binary
                     result{i}=out;% Store final result
                     new={trim_Ainstruction,d2s}; % Added new row
```

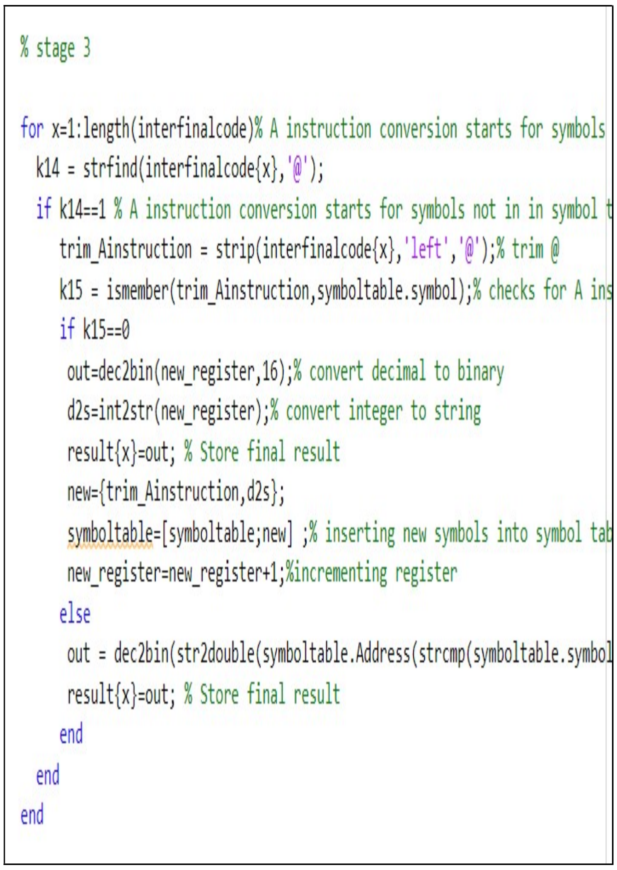
Figure 3 Conversion (Stage 2)



```
                    symboltable=[symboltable;new] ; % inserting new symbol
                end
            end
        end

    elseif tf==1
        result{i}=interfinalcode{x}; % Store final result those who already
    elseif k5==1 % C instruction conversion starts
        %disp('c instruction found');
        first  ='111';
        jumpbin='000';
        csecond_comp= extractAfter(interfinalcode{x},"=");%Take value after
        cfirst_dest = extractBefore(interfinalcode{x},"=");%Take value befo
        dest        = dcode.d_address(strcmp(dcode.d,cfirst_dest));
        comp        = ccode.c_address(strcmp(ccode.c,csecond_comp));
        out         = dec2bin(uint16(bin2dec(strcat(first,comp,dest,jumpbin
        result{i}=out; % Store final result

    else  %c instruction jump conversion starts
          %disp('jump found');
          first='111';
          dest ='000'; % jump instruction conversion starts
          jsecond_comp= extractAfter(interfinalcode{x},";");%Take value aft
          jfirst_jump = extractBefore(interfinalcode{x},";");%Take value be
          jumpbin     = jumpcode.jumpaddress(strcmp(jumpcode.jump,jsecond_c
          comp        = ccode.c_address(strcmp(ccode.c,jfirst_jump));
          out         = dec2bin(uint16(bin2dec(strcat(first,comp,dest,jumpb
          result{i}=out; % Store final result
    end
    i=i+1;
```

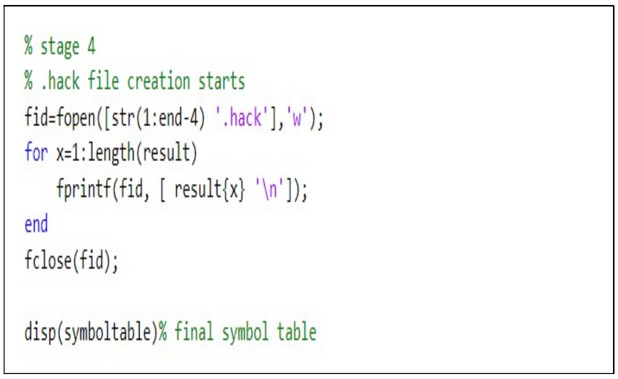
Figure 4 Conversion (Stage 2)

Now the program moves to stage 2, where the code is checked with a for loop for separation and conversion purposes. Here the for-loop iteration length is code length which has in 'intefinalcode'. Initially, we will look for integers, commands which have equal symbols ('=') and have '@' symbols in the code. If the code identifies as integer, then nothing, we need to do we will directly copy that into our result array. Next, if the for loop identifies the '@' symbol the line identifies as A-instruction then we will enter into the if loop for binary conversion. In this MATLAB assembler code, A-instruction conversion is in four-stage and the first stage will be direct conversion second is A-instruction the value will have already been stored in the symbol table the third stage is A-instruction which will have a label in the code and the final stage is A-instruction which doesn't have a label or predefined label in the symbol table. The final stage is achieved in stage 3. So, in stage 2 we will check for C instruction and Jump instruction. This is achieved by checking a line that has '=' or ';' symbols. If a code line has a '= 'symbol the assembler identified as C instruction and will process accordingly. If a line has ';' symbol the assembler identified as jump instruction then the assembler will follow further instructions



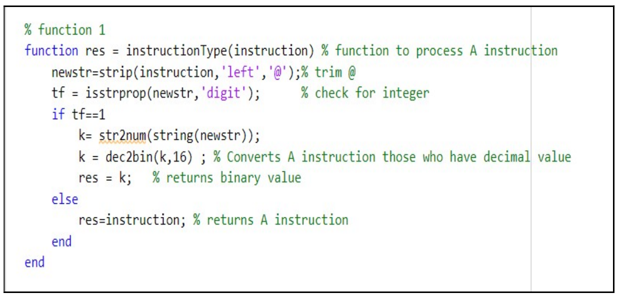
```
% stage 3

for x=1:length(interfinalcode)% A instruction conversion starts for symbols
  k14 = strfind(interfinalcode{x},'@');
  if k14==1 % A instruction conversion starts for symbols not in in symbol t
     trim_Ainstruction = strip(interfinalcode{x},'left','@');% trim @
     k15 = ismember(trim_Ainstruction,symboltable.symbol);% checks for A ins
     if k15==0
      out=dec2bin(new_register,16);% convert decimal to binary
      d2s=int2str(new_register);% convert integer to string
      result{x}=out; % Store final result
      new={trim_Ainstruction,d2s};
      symboltable=[symboltable;new] ;% inserting new symbols into symbol tab
      new_register=new_register+1;%incrementing register
     else
      out = dec2bin(str2double(symboltable.Address(strcmp(symboltable.symbol
      result{x}=out; % Store final result
     end
  end
end
```

Figure 5 Conversion (Stage 3)



```
% stage 4
% .hack file creation starts
fid=fopen([str(1:end-4) '.hack'],'w');
for x=1:length(result)
    fprintf(fid, [ result{x} '\n']);
end
fclose(fid);

disp(symboltable)% final symbol table
```

Figure 6 Output file creation (Stage 4)



```
% function 1
function res = instructionType(instruction) % function to process A instruction
    newstr=strip(instruction,'left','@');% trim @
    tf = isstrprop(newstr,'digit');      % check for integer
    if tf==1
        k= str2num(string(newstr));
        k = dec2bin(k,16) ; % Converts A instruction those who have decimal value
        res = k;   % returns binary value
    else
        res=instruction; % returns A instruction
    end
end
```

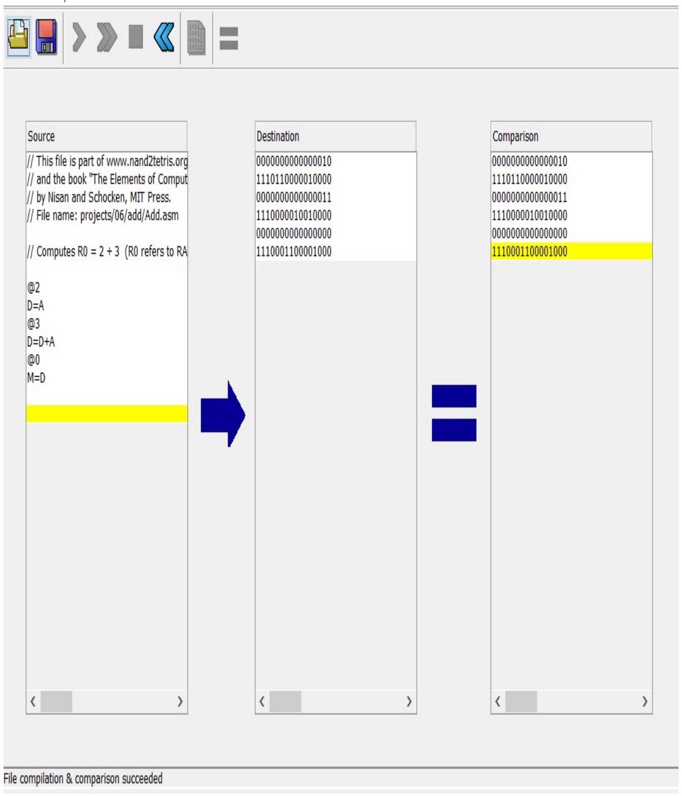
Figure 7 Function 1

Now the assembler will move to stage 3, the final sortation of A-instruction. Here we will convert A-instruction which doesn't have a label or doesn't have in the predefined symbols in the symbol table. The new symbol will be added to the symbol table and assigned a decimal value from 16(because 0 to 15 there are already reserved registers stored in the symbol table). If the same symbols are repeated in the code, it will retrieve data from the symbol table else new symbol came then it will be added to the symbol table and give decimal value 17 and so on. Once we assigned a decimal value to a symbol, we will convert that value into a binary value and store that into a result array.

The final or fourth stage of the assembler code is generating a hack file in the text format, for this purpose we will retrieve the data that have been stored in the 'result' array. In this code, we use one function only.

Result Evaluation

The result evaluation was done with the help of Java Assembler which has already been given. The MATLAB assembler will auto-generate a hack file. We need to upload one asm file on one side and need to upload MATLAB generated hack file on another side of the Assembler. If the generated result is correct the assembler will show compilation and comparison success. In this way, we tested 4 ASM files which have given in Lab 6 and compared them successfully.



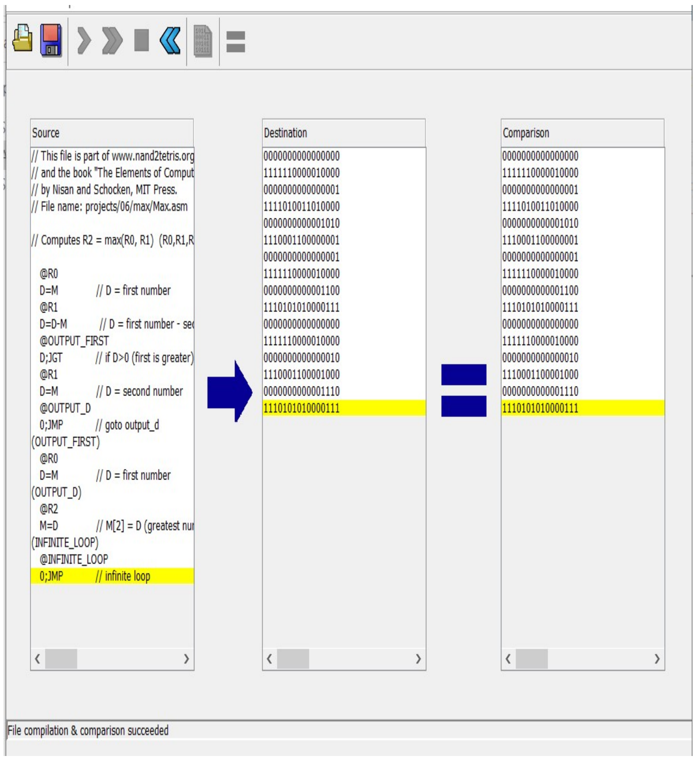
```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Comput
// by Nisan and Schocken, MIT Press.
// File name: projects/06/add/Add.asm

// Computes R0 = 2 + 3  (R0 refers to RA

@2
D=A
@3
D=D+A
@0
M=D
```

```
0000000000000010
1110110000010000
0000000000000011
1110000010010000
0000000000000000
1110001100001000
```

```
0000000000000010
1110110000010000
0000000000000011
1110000010010000
0000000000000000
1110001100001000
```

File compilation & comparison succeeded

Figure 8 Add.asm



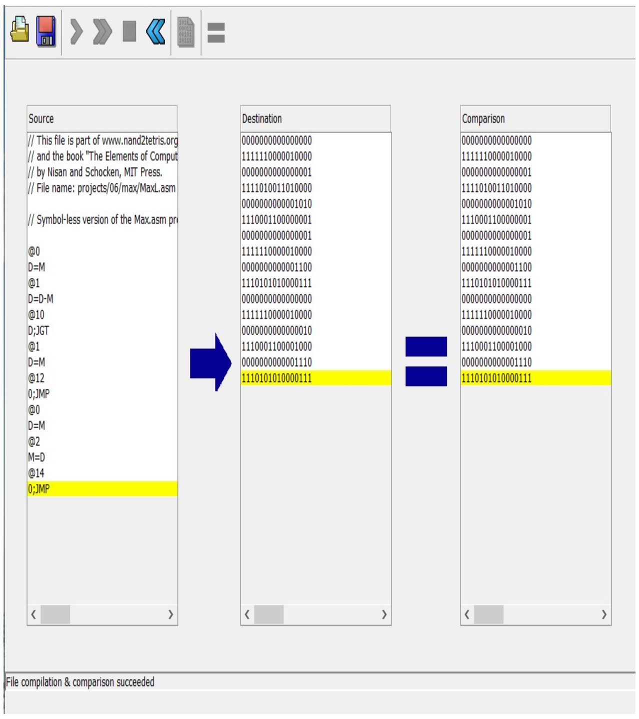
```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Comput
// by Nisan and Schocken, MIT Press.
// File name: projects/06/max/Max.asm

// Computes R2 = max(R0, R1)  (R0,R1,R

  @R0
  D=M              // D = first number
  @R1
  D=D-M            // D = first number - sec
  @OUTPUT_FIRST
  D;JGT            // if D>0 (first is greater)
  @R1
  D=M              // D = second number
  @OUTPUT_D
  0;JMP            // goto output_d
(OUTPUT_FIRST)
  @R0
  D=M              // D = first number
(OUTPUT_D)
  @R2
  M=D              // M[2] = D (greatest nu
(INFINITE_LOOP)
  @INFINITE_LOOP
  0;JMP            // infinite loop
```

```
0000000000000000
1111110000010000
0000000000000001
1111010011010000
0000000000001010
1110001100000001
0000000000000001
1111110000010000
0000000000001100
1110101010000111
0000000000000000
1111110000010000
0000000000000010
1110001100001000
0000000000001110
1110101010000111
```

```
0000000000000000
1111110000010000
0000000000000001
1111010011010000
0000000000001010
1110001100000001
0000000000000001
1111110000010000
0000000000001100
1110101010000111
0000000000000000
1111110000010000
0000000000000010
1110001100001000
0000000000001110
1110101010000111
```

File compilation & comparison succeeded

Figure 9 max.asm



Source

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Comput
// by Nisan and Schocken, MIT Press.
// File name: projects/06/max/MaxL.asm

// Symbol-less version of the Max.asm pr

@0
D=M
@1
D=D-M
@10
D;JGT
@1
D=M
@12
0;JMP
@0
D=M
@2
M=D
@14
0;JMP
```

Destination

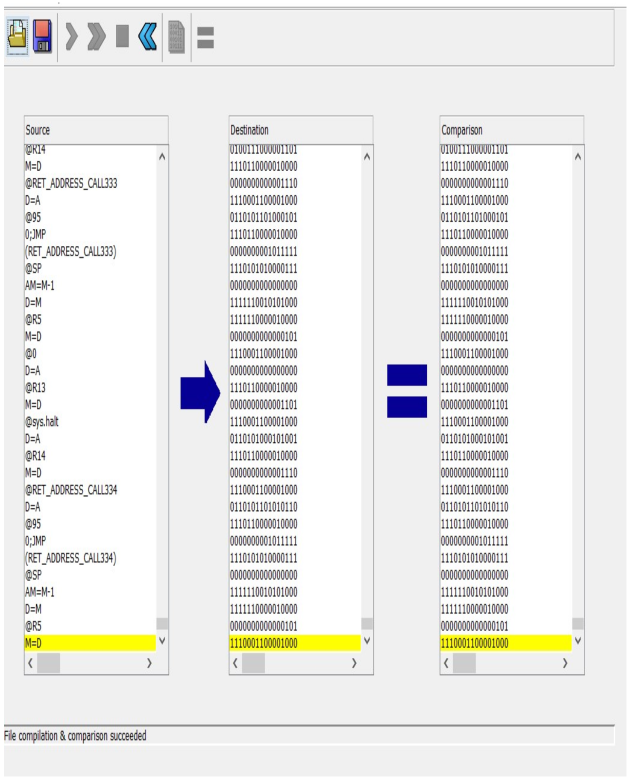
```
0000000000000000
1111110000010000
0000000000000001
1111010011010000
0000000000001010
1110001100000001
0000000000000001
1111110000010000
0000000000001100
1110101010000111
0000000000000000
1111110000010000
0000000000000010
1110001100001000
0000000000001110
1110101010000111
```

Comparison

```
0000000000000000
1111110000010000
0000000000000001
1111010011010000
0000000000001010
1110001100000001
0000000000000001
1111110000010000
0000000000001100
1110101010000111
0000000000000000
1111110000010000
0000000000000010
1110001100001000
0000000000001110
1110101010000111
```

File compilation & comparison succeeded

Figure 10 maxl.asm



Source

```
@R14
M=D
@RET_ADDRESS_CALL333
D=A
@95
0;JMP
(RET_ADDRESS_CALL333)
@SP
AM=M-1
D=M
@R5
M=D
@0
D=A
@R13
M=D
@sys.halt
D=A
@R14
M=D
@RET_ADDRESS_CALL334
D=A
@95
0;JMP
(RET_ADDRESS_CALL334)
@SP
AM=M-1
D=M
@R5
M=D
```

Destination

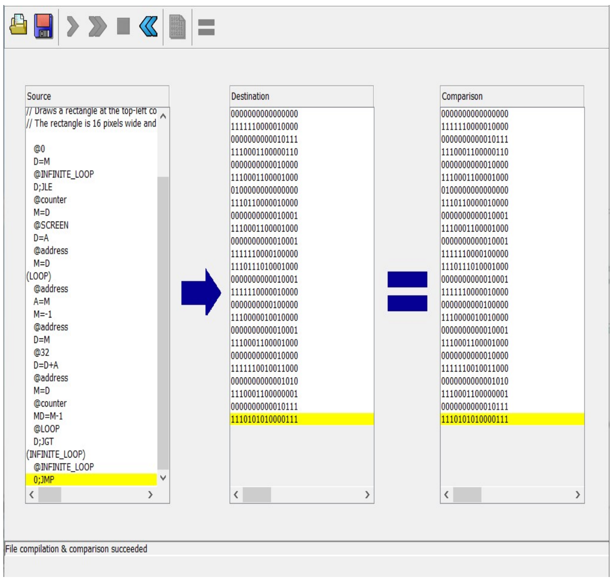
```
0100111000001101
1110110000010000
0000000000001110
1110001100001000
0110101101000101
1110110000010000
0000000001011111
1110101010000111
0000000000000000
1111110010101000
1111110000010000
0000000000000101
1110001100001000
0000000000000000
1110110000010000
0000000000001101
1110001100001000
0110101000101001
1110110000010000
0000000000001110
1110001100001000
0110101101010110
1110110000010000
0000000001011111
1110101010000111
0000000000000000
1111110010101000
1111110000010000
0000000000000101
1110001100001000
```

Comparison

```
0100111000001101
1110110000010000
0000000000001110
1110001100001000
0110101101000101
1110110000010000
0000000001011111
1110101010000111
0000000000000000
1111110010101000
1111110000010000
0000000000000101
1110001100001000
0000000000000000
1110110000010000
0000000000001101
1110001100001000
0110101000101001
1110110000010000
0000000000001110
1110001100001000
0110101101010110
1110110000010000
0000000001011111
1110101010000111
0000000000000000
1111110010101000
1111110000010000
0000000000000101
1110001100001000
```

File compilation & comparison succeeded

Figure 11 Pong.asm



Source

```
// Draws a rectangle at the top-left co
// The rectangle is 16 pixels wide and

  @0
  D=M
  @INFINITE_LOOP
  D;JLE
  @counter
  M=D
  @SCREEN
  D=A
  @address
  M=D
(LOOP)
  @address
  A=M
  M=-1
  @address
  D=M
  @32
  D=D+A
  @address
  M=D
  @counter
  MD=M-1
  @LOOP
  D;JGT
(INFINITE_LOOP)
  @INFINITE_LOOP
  0;JMP
```

Destination

```
0000000000000000
1111110000010000
0000000000010111
1110001100000110
0000000000010000
1110001100001000
0100000000000000
1110110000010000
0000000000010001
1110001100001000
0000000000010001
1111110000100000
1110111010001000
0000000000010001
1111110000010000
0000000000100000
1110000010010000
0000000000010001
1110001100001000
0000000000010000
1111110010011000
0000000000001010
1110001100000001
0000000000010111
1110101010000111
```

Comparison

```
0000000000000000
1111110000010000
0000000000010111
1110001100000110
0000000000010000
1110001100001000
0100000000000000
1110110000010000
0000000000010001
1110001100001000
0000000000010001
1111110000100000
1110111010001000
0000000000010001
1111110000010000
0000000000100000
1110000010010000
0000000000010001
1110001100001000
0000000000010000
1111110010011000
0000000000001010
1110001100000001
0000000000010111
1110101010000111
```

File compilation & comparison succeeded

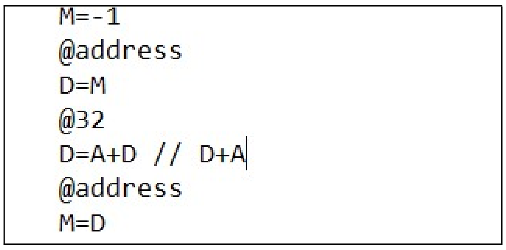
Figure 12 rect.asm

Critical reflection

The designed assembler with the MATLAB code has faced a few issues like— This code can't handle complex symbol notations. Its processing time is higher for lengthy codes as compared to the other assemblers. Moreover, it has limitations in processing complex instruction types, because as complexity increases the symbol table increases. While comparing with the conversion time of pongl.asm file with pong.asm file the MATLAB assembler taking more time for conversion for pong.asm (around **30 seconds**!). Except for pong.asm all other programs are taking only **1 to 2 seconds**. So, the MATLAB compiler takes more time for converting those codes to have more symbols and more number lines. While creating MATLAB assembler code the more challenges I faced was for the sortation logic of A-instruction code which we need to add from register 16 onwards in the symbol table. So, I created a separate for-loop outside the main parsing for-loop. To increase MATLAB iteration speed in multicore computers we can use parallel **for loops (parfor)** to run independent iteration in parallel iteration.

Another important point noted down during execution time is this MATLAB assembler won't work if any C-instruction came in the below-mentioned way. Mathematically **D+A** is equal to **A+D** but our assembler won't understand this. We need to specify this in our Table also. For testing purposes, I added both values to the table. But it will be difficult to add all possibilities to the table. Because the program length will increase and program execution time will also increase. For further investigation purposes, I tried to execute **mult.asm** in **lab**

4 but there are a few lines that can’t able to converted due to the above reasons.



```
M=-1
@address
D=M
@32
D=A+D // D+A
@address
M=D
```

Figure 13 Rect.asm

Conclusion

The prime objective of our project was to create an assembler program using the MATLAB platform that translates programs written in the symbolic hack assembly language into binary code. We consider MATLAB as a platform to create assemblers because when it comes to numeric calculation or analysis, MATLAB gives faster results than the other programming languages. The Hack assembler reads input as a text file named Prog.asm (a code with .asm extension) and develops a text file named Prog.hack (a code with ‘. hack’ extension), that contains a translated Hack machine code. While moving from add.asm to pong.asm its observed that execution time increased. Finally, the code generated from MATLAB assembler is compared with code generated from Java assembler.

References

[1] Trauth M.H. (2007) Introduction to MATLAB. In: MATLAB® Recipes for Earth Sciences. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-72749-1_2

[2] Blum, Richard. Professional Assembly Language, John Wiley & Sons, Incorporated, 2005. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/shu/detail.action?docID=225831

[3] Ock, J., Kim, H., Kim, H.S., Paek, J. and Bahk, S., 2019. Low-power wireless with denseness: The case of an electronic shelf labeling system—Design and experience. *IEEE Access*, *7*, pp.163887-163897.