Mu0 compiler and simulator

Date: 2014-11-13
Author: Nikolaos Kavvadias <nikos@nkavvadias.com>
Email: info@ajaxcompilers.com
Revision: 0.0.1 (2014-11-13) [copy of the original work by benjya: http://everything2.com/title/MU0]
Web site: http://github.com/nkkav/mu0

Contents

1 What is the MU0?

The MU0 is an abstract computer used for educational purposes at the University of Manchester. I was first exposed to it as part of my under-graduate degree course at UMIST. It is based on the SSEM computer which was one of the first computers every built - at the University (and is considered, along with the Harvard Mark 1 to be the first real computer).

As mentioned, it is used to illustrate basic programming concepts, and encourages thorough design due to the fact it only has 7 actual commands. This WU is based on a project I completed at the end of my first year to create a complete development environment for the "processor".

2 The command set

The processor can directly address 4096 words, each 16 bits long. Each word is capable of storing one fixed length command, which consists of 4 bits of opcode and 12 bits of operand, in all cases except the STOP command which takes no operand.

The only internal register is known as the "accumulator" and this is where all processing must take place. It is 16 bits long, and is where both inputs to calculations and results must be stored.

The following is a list of all the commands supported in the processor.

- Opcode 0: `ACC<= [address]`
  - Load accumulator with contents of address
- Opcode 1: `ACC>= [address]`
  - Store contents of accumulator at address
- Opcode 2: `ACC+ [address]`

  - Add contents of address to contents of accumulator

- Opcode 3: `ACC- [address]`
  - Subtract contents of address from contents of accumulator
- Opcode 4: `PC<= address`
  - Jump to address
- Opcode 5: `IF+VE PC<= address`
  - If contents of accumulator are positive, jump to address
- Opcode 6: `IF!=0 PC<= address`
  - If contents of accumulator are not zero, jump to address
- Opcode 7: `STOP`
  - Stop processor

Three major functions, present in most other processors, are missing from the MU0 instruction set, and these should be noted here.

- There is no immediate addressing - such as "Load accumulator with this value", only "Load accumulator with contents of address".
- There is no form of jumping to a subroutine with a return value pushed onto a stack.
- There is no form of indexed addressing.

Methods to work around all three of these will be illustrated in the program examples later on in this document.

3 The development environment

Although not directly related to the processor itself, the development environment was a core part of this system and it helps illustrate the way the processor works. If you want to try compiling these programs (shown at the end), they were written and compiled on Borland Turbo C 2 (yes, that old!) and also GCC on HP-UX. You're welcome to use them for anything you want.

The system allows complete development of MU0 programs, from compilation (assembly really) to execution and debugging. An editor isn't included. The compiler asks for two filenames - the source MU0 assembly listing, and a destination for the "executable" code. There is no library and therefore no need for a linker stage. All the commands listed above are supported, and support is also provided for putting literals in the code and labels to make jumping around easier.

The second part of the system is the executer (emulator) / debugger. This is emulated as a separate program. It reads in the "executable" code and then executes it in various modes (straight through, trace or single step). The user can check the contents of memory and the accumulator at any time.

The compiler source code is designed from the bottom up (as the MU0 has never really existed), but based on other compilers. Comments can be placed anywhere in

a source program, simply by prefixing the line with a semicolon (`;`). On seeing this, the compiler should ignore the rest of the line. Blank lines can also be left anywhere within the program to increase readibility.

Actual commands are written simply as they are listed above, with the operand either in or out of square brackets `[]` as the command demands. There should be a space between the command and the operand, however, and spaces between each section of the two conditional branch commands. If the first character of the operand is a dollar sign (`$`), then the operand is assumed to be a direct address in hexadecimal. If this isn't present, then the operand is assumed to be a label.

The position of labels within the text is set by placing `.label labelname` within the source code. The next pseudo-op is `.word hexnumber` which places a literal 16 bit hexadecimal number in the code. The final pseudo-op is `.end` which is used on its own to indicate the end of the source code. Anything placed after it is completly ignored.

The debugger will have its own command line and simple command line processor, supporting the following commands.

load filename Load MU0 executable code from specified filename into the start of its `memory`.

go Execute code from the start of "memory" in "trace" mode - displaying each line on execution but without pausing.

goquiet Execute code from the start of "memory" in "quiet" mode - non stop and with no display.

gostep Execute code from the start of "memory" in "single step" mode - display each command on execution and then pause and wait for the user to select to step to another command, change to "trace" or "quiet" mode, or stop execution.

set address value Set specified address in "memory" to specified value.

view address View the contents of the specified address in "memory".

viewacc View the contents of the accumulator.

setacc value Set the accumulator to the specified value.

dump lowadd hiadd Show the contents of all memory locations between the specified addresses.

help Display a list of the available commands.

exit Quit the debugger.

The design outlines are here, and the complete compiler and debugger source code has been made available.

4 Compiler and Debugger outline design

The compiler is of the two pass variety. This is to avoid a major problem with compilers, that if, during compilation, the compiler encounters a label which is defined lower down in the source code, it won't know where it is. So, the compiler makes a first check

through the source, not generating any code but simply building up a list in memory of all the labels and where they point to. The second pass is then the code generation pass.

For the first pass, only the source file is opened, and the address counter reset. The program is the stepped through command by command, and for each command (including the `.word` pseudo-op) the address counter is incremented. Therefore, whenever the compiler encounters a `.label` pseudo-op, it can add the label together with its address within the code to its internal symbol table for future reference. Assuming no invalid commands are found during this pass (including the need for a `.end` at the end), compilation then moves into the second pass.

The second pass is the code generation pass. This time, we open both the source and destination files, and again read in the source file line by line. We also actually attempt to assemble each command in the source code, and process the operand, be it an immediate number or a label. If a label is referenced which isn't already in the symbol table then an error has to be flagged. Again, the total number of errors is displayed at the end of compilation.

Assuming both stages complete successfully, an output file will remain which can then be loaded into the debugger and executer.

The debugger must start by defining an "accumulator" and "memory" within variables to work on. The "accumulator" is 16 bits, and the "memory" must be large enough to accomodate 4096 words, each 16 bits long.

It is based around a command line, obeying the list of commands given above. All addresses and values for the display commands, and during stepped or traced execution, are to be displayed in hexadecimal as this is the standard notation in computing.

On each step of execution if there is any form of display, the current address, current executing instruction, and current contents of the accumulator, should be displayed. If we are in single step mode, this should then be followed by a prompt to halt execution, change to trace mode, change to quite execution or continue in single step mode.

It is assumed that the code which is loaded into the debugger will contain only correct instructions. However, this may not be the case as someone may enter the wrong filename and load a file which is not MU0 exutable code. Therefore, if during execution an invalid command is encountered (opcode is greater than seven) then this should be flagged, but execution continued.

5 Sample Program 1 - Multiplication

The MU0 has no function to multiply two numbers. This program multiplies the contents of A0 by A1 storing the result in A2. It also shows the method for getting round the lack of immediate addressing - a label is created such as ONE and the actual number number is then placed in this location. The program to do this is as follows.

```
; Multiply.MU0
; Test Program
; (C) 1994 Benjy
; Reset A2 (where we store the result)
ACC<= [ZR]
ACC=> [$A2]
; Start the main loop
.label LOOP
```

```
; Load the result into the accumulator
ACC<= [$A2]
; Add the value
ACC+ [$A0]
; Store it out in the result
ACC=> [$A2]
; Load the multiplier
ACC<= [$A1]
; Subtract 1 from it
ACC- [ONE]
; Store it out
ACC=> [$A1]
; If we haven't got to 0, loop around
IF!=0 PC<= LOOP
; Done
STOP
.label ZR
.word 0
.label ONE
.word 1
.end
```

On compilation, this gives the following.

```
COMPILE_MU0 - companion program to EXECUTE_MU0
(C) 1994 Benjy

Please enter source filename > multiply.mu0
Please enter destination filename > multiply.mu0.exe
Commencing compilation...

Pass 1
Opening source file multiply.mu0
Clearing label index
Label 1 at address [002]: LOOP
Label 2 at address [00A]: ZR
Label 3 at address [00B]: ONE

End of file marker, pass 1 complete

Pass 2
Opening source file multiply.mu0
Opening destination file multiply.mu0.exe

[000] ; Multiply.MU0
[000] ; Test Program
[000] ; (C) 1994 Benjy
[000]
[000] ACC<= [ZR]                    000A
[001] ACC=> [$A2]                   10A2
[002] .label LOOP
```

```
[002] ACC<= [$A2]                    00A2
[003] ACC+ [$A0]                     20A0
[004] ACC=> [$A2]                    10A2
[005] ACC<= [$A1]                    00A1
[006] ACC- [ONE]                     300B
[007] ACC=> [$A1]                    10A1
[008] IF!=0 PC<= LOOP                6002
[009] STOP                           7000
[00A]
[00A] .label ZR
[00A] .word 0                        0000
[00B]
[00B] .label ONE
[00B] .word 1                        0001

[00C]
[00C] .end
End of file marker, pass 2 complete
```

The first part of this listing shows the first pass being processed, and the compiler is only going through the code to identify labels and build up a list in memory. Pass 2 is the main compilation phase. Each line shows the current address, a line of source code, and the 16 bit hexadecimal code at the end of the line. This can be checked through to be correct.

The testing of this program was performed using the debugging program, and simply consisted of loading the code into "memory", placing various values in A0 and A1, executing it and checking the value in A2. A sample pass of executing this program is shown here, multiplying 1A (26) by 06 (6) and giving the correct answer of 9C (156).

```
EXECUTE_MU0 - companion program to COMPILE_MU0
(C) 1994 Benjy

Enter "help" for command list

> load multiply.mu0.exe
[000] : 000A
[001] : 10A2
[002] : 00A2
[003] : 20A0
[004] : 10A2
[005] : 00A1
[006] : 300B
[007] : 10A1
[008] : 6002
[009] : 7000
[00A] : 0000
[00B] : 0001
> set a0 1a
[0A0] : 001A
> set a1 06
[0A1] : 0006
```

```
> go
PC : 000  PI : 000A  ACC : 0000
PC : 001  PI : 10A2  ACC : 0000
PC : 002  PI : 00A2  ACC : 0000
PC : 003  PI : 20A0  ACC : 0000
PC : 004  PI : 10A2  ACC : 001A
PC : 005  PI : 00A1  ACC : 001A
PC : 006  PI : 300B  ACC : 0006
PC : 007  PI : 10A1  ACC : 0005
PC : 008  PI : 6002  ACC : 0005
PC : 002  PI : 00A2  ACC : 0005
PC : 003  PI : 20A0  ACC : 001A
PC : 004  PI : 10A2  ACC : 0034
PC : 005  PI : 00A1  ACC : 0034
PC : 006  PI : 300B  ACC : 0005
PC : 007  PI : 10A1  ACC : 0004
PC : 008  PI : 6002  ACC : 0004
PC : 002  PI : 00A2  ACC : 0004
PC : 003  PI : 20A0  ACC : 0034
PC : 004  PI : 10A2  ACC : 004E
PC : 005  PI : 00A1  ACC : 004E
PC : 006  PI : 300B  ACC : 0004
PC : 007  PI : 10A1  ACC : 0003
PC : 008  PI : 6002  ACC : 0003
PC : 002  PI : 00A2  ACC : 0003
PC : 003  PI : 20A0  ACC : 004E
PC : 004  PI : 10A2  ACC : 0068
PC : 005  PI : 00A1  ACC : 0068
PC : 006  PI : 300B  ACC : 0003
PC : 007  PI : 10A1  ACC : 0002
PC : 008  PI : 6002  ACC : 0002
PC : 002  PI : 00A2  ACC : 0002
PC : 003  PI : 20A0  ACC : 0068
PC : 004  PI : 10A2  ACC : 0082
PC : 005  PI : 00A1  ACC : 0082
PC : 006  PI : 300B  ACC : 0002
PC : 007  PI : 10A1  ACC : 0001
PC : 008  PI : 6002  ACC : 0001
PC : 002  PI : 00A2  ACC : 0001
PC : 003  PI : 20A0  ACC : 0082
PC : 004  PI : 10A2  ACC : 009C
PC : 005  PI : 00A1  ACC : 009C
PC : 006  PI : 300B  ACC : 0001
PC : 007  PI : 10A1  ACC : 0000
PC : 008  PI : 6002  ACC : 0000
PC : 009  PI : 7000  ACC : 0000
> view a2
[0A2] : 009C
> exit
Done
```

6 Sample Program 2 - Sorting out Odd and Even numbers

The program shown previously is very simple, and for a more extensive test, another program was written. Its function is to work its way through a list of numbers starting at memory location 200 going on until 240, and place all even numbers into a list starting at address 241.

This, on most processors, would make use of "indexed" addressing, whereby the contents of an index register are added onto the specified address to generate the true address, making accessing a list very easy. This mode does not exist on the MU0, so "self modifying" code has to be used instead. This is where the program actually updates itself during its execution to allow incrementing through the list. The program also has to check if a number is even or odd, what would normally be done by shifting the Least Significant Bit into the carry and branching according to the carry. Again this isn't possible, so instead we handle the problem by successively subtracting 2 from the number, and seeing if we get to zero or -1 first, zero indicating an even number.

The code is as follows.

```
; odd_even.mu0
; Finds even numbers in a list
;
; THIS PROGRAM IS SELF-MODIFYING. However, it will ensure
; that it resets itself before each run so it doesn't need
; reloading.
;
; List is from $200 to $240
; Even numbers copied to list starting at $241
;
; Version 1, 13 October 1994
;
; (C) Benjy (Soft Eng A2)

; Start by resetting "indirect" addressing bits
ACC<= [DEF_GET_SOURCE]
ACC=> [GET_SOURCE]
ACC=> [GET_SOURCE_1]
ACC<= [DEF_PUT_DEST]
ACC=> [PUT_DEST]

; Main program loop
.label MAIN_LOOP

; Here is where we get the source number - self modifying
.label GET_SOURCE
ACC<= [$200]

; Now check to see if even by subtracting two many times
.label SUBTRACT_LOOP
; We must check if done first to ensure that 0 works
IF+VE PC<= SUBTRACT_NOT_DONE
```

```
PC<= SUBTRACT_DONE
.label SUBTRACT_NOT_DONE
ACC- [TWO]
PC<= SUBTRACT_LOOP

; We now have either -1 (odd) or 0 (even) in ACC.
.label SUBTRACT_DONE
IF!=0 PC<= ODD

; The number is even so copy it to the second list
; This (both commands) are now self modifying
.label GET_SOURCE_1
ACC<= [$200]
.label PUT_DEST
ACC=> [$241]

; Increment the destination index
ACC<= [PUT_DEST]
ACC+ [ONE]
ACC=> [PUT_DEST]

.label ODD
; Now (always) increment the source index
ACC<= [GET_SOURCE]
ACC+ [ONE]
ACC=> [GET_SOURCE]
ACC=> [GET_SOURCE_1]

; And check to see if we've finished
ACC- [END_GET_SOURCE]
IF!=0 PC<= MAIN_LOOP

; All Done
STOP

; Default command to get from source list
.label DEF_GET_SOURCE
ACC<= [$200]

; What the GET_SOURCE will be when we finish
.label END_GET_SOURCE
ACC<= [$241]

; Default command to write to destination list
.label DEF_PUT_DEST
ACC=> [$241]

; Constants
.label ONE
.word 1
```

```
.label TWO
.word 2

; End
.end
```

On compiling, the output is as follows.

```
COMPILE_MU0 - companion program to EXECUTE_MU0
(C) 1994 Benjy

Please enter source filename > odd_even.mu0
Please enter destination filename > odd_even.mu0.exe
Commencing compilation...

Pass 1
Opening source file odd_even.mu0
Clearing label index
Label 1 at address [005]: MAIN_LOOP
Label 2 at address [005]: GET_SOURCE
Label 3 at address [006]: SUBTRACT_LOOP
Label 4 at address [008]: SUBTRACT_NOT_DONE
Label 5 at address [00A]: SUBTRACT_DONE
Label 6 at address [00B]: GET_SOURCE_1
Label 7 at address [00C]: PUT_DEST
Label 8 at address [010]: ODD
Label 9 at address [017]: DEF_GET_SOURCE
Label 10 at address [018]: END_GET_SOURCE
Label 11 at address [019]: DEF_PUT_DEST
Label 12 at address [01A]: ONE
Label 13 at address [01B]: TWO

End of file marker, pass 1 complete

Pass 2
Opening source file odd_even.mu0
Opening destination file odd_even.mu0.exe

[000] ; odd_even.mu0
[000] ; Finds even numbers in a list
[000] ;
[000] ; THIS PROGRAM IS SELF-MODIFYING. However, it will ensure
[000] ; that it resets itself before each run so it doesn't need
[000] ; reloading.
[000] ;
[000] ; List is from $200 to $240
[000] ; Even numbers copied to list starting at $241
[000] ;
[000] ; Version 1, 13 October 1994
[000] ;
[000] ; (C) Benjy (Soft Eng A2)
```

```
[000]
[000] ; Start by resetting "indirect" addressing bits
[000] ACC<= [DEF_GET_SOURCE]        0017
[001] ACC=> [GET_SOURCE]            1005
[002] ACC=> [GET_SOURCE_1]          100B
[003] ACC<= [DEF_PUT_DEST]          0019
[004] ACC=> [PUT_DEST]              100C
[005]
[005] ; Main program loop
[005] .label MAIN_LOOP
[005]
[005] ; Here is where we get the source number - self modifying
[005] .label GET_SOURCE
[005] ACC<= [$200]                  0200
[006]
[006] ; Now check to see if even by subtracting two many times
[006] .label SUBTRACT_LOOP
[006] ; We must check if done first to ensure that 0 works
[006] IF+VE PC<= SUBTRACT_NOT_DONE  5008
[007] PC<= SUBTRACT_DONE            400A
[008] .label SUBTRACT_NOT_DONE
[008] ACC- [TWO]                    301B
[009] PC<= SUBTRACT_LOOP            4006
[00A]
[00A] ; We now have either -1 (odd) or 0 (even) in ACC.
[00A] .label SUBTRACT_DONE
[00A] IF!=0 PC<= ODD                6010
[00B]
[00B] ; The number is even so copy it to the second list
[00B] ; This (both commands) are now self modifying
[00B] .label GET_SOURCE_1
[00B] ACC<= [$200]                  0200
[00C] .label PUT_DEST
[00C] ACC=> [$241]                  1241
[00D]
[00D] ; Increment the destination index
[00D] ACC<= [PUT_DEST]              000C
[00E] ACC+ [ONE]                    201A
[00F] ACC=> [PUT_DEST]              100C
[010]
[010] .label ODD
[010] ; Now (always) increment the source index
[010] ACC<= [GET_SOURCE]            0005
[011] ACC+ [ONE]                    201A
[012] ACC=> [GET_SOURCE]            1005
[013] ACC=> [GET_SOURCE_1]          100B
[014]
[014] ; And check to see if we've finished
[014] ACC- [END_GET_SOURCE]         3018
[015] IF!=0 PC<= MAIN_LOOP          6005
```

```
[016]
[016] ; All Done
[016] STOP                          7000
[017]
[017] ; Default command to get from source list
[017] .label DEF_GET_SOURCE
[017] ACC<= [$200]                  0200
[018]
[018] ; What the GET_SOURCE will be when we finish
[018] .label END_GET_SOURCE
[018] ACC<= [$241]                  0241
[019]
[019] ; Default command to write to destination list
[019] .label DEF_PUT_DEST
[019] ACC=> [$241]                  1241
[01A]
[01A] ; Constants
[01A] .label ONE
[01A] .word 1                       0001
[01B] .label TWO
[01B] .word 2                       0002
[01C]
[01C] ; End
[01C] .end
End of file marker, pass 2 complete
```

A Sample execution is shown now. We load the code, then set some values from address 200 onwards to various numbers. The code is executed, and the dump shows the even numbers appear starting at address 241.

```
EXECUTE_MU0 - companion program to COMPILE_MU0
(C) 1994 Benjy

Enter "help" for command list

> load odd_even.mu0.exe
[000] : 0017
[001] : 1005
[002] : 100B
[003] : 0019
[004] : 100C
[005] : 0200
[006] : 5008
[007] : 400A
[008] : 301B
[009] : 4006
[00A] : 6010
[00B] : 0200
[00C] : 1241
[00D] : 000C
[00E] : 201A
```

```
[00F] : 100C
[010] : 0005
[011] : 201A
[012] : 1005
[013] : 100B
[014] : 3018
[015] : 6005
[016] : 7000
[017] : 0200
[018] : 0241
[019] : 1241
[01A] : 0001
[01B] : 0002
> set 200 1
[200] : 0001
> set 201 2
[201] : 0002
> set 202 3
[202] : 0003
> set 203 4
[203] : 0004
> set 204 89
[204] : 0089
> set 205 a2
[205] : 00A2
> set 206 b3
[206] : 00B3
> set 207 f1
[207] : 00F1
> set 208 e0
[208] : 00E0
> set 209 12
[209] : 0012
> set 20a 0
[20A] : 0000
> set 20b 15
[20B] : 0015
> set 20c 93
[20C] : 0093
> set 20d d4
[20D] : 00D4
> set 20e f4
[20E] : 00F4
> dump 200 240

[200]  0001 0002 0003 0004  0089 00A2 00B3 00F1
[208]  00E0 0012 0000 0015  0093 00D4 00F4 0000
[210]  0000 0000 0000 0000  0000 0000 0000 0000
[218]  0000 0000 0000 0000  0000 0000 0000 0000
[220]  0000 0000 0000 0000  0000 0000 0000 0000
```

```
[228]  0000 0000 0000 0000  0000 0000 0000 0000
[230]  0000 0000 0000 0000  0000 0000 0000 0000
[238]  0000 0000 0000 0000  0000 0000 0000 0000
[240]  0000
> goquiet

> dump 200 240

[200]  0001 0002 0003 0004  0089 00A2 00B3 00F1
[208]  00E0 0012 0000 0015  0093 00D4 00F4 0000
[210]  0000 0000 0000 0000  0000 0000 0000 0000
[218]  0000 0000 0000 0000  0000 0000 0000 0000
[220]  0000 0000 0000 0000  0000 0000 0000 0000
[228]  0000 0000 0000 0000  0000 0000 0000 0000
[230]  0000 0000 0000 0000  0000 0000 0000 0000
[238]  0000 0000 0000 0000  0000 0000 0000 0000
[240]  0000
> dump 241 260

[241]  0002 0004 00A2 00E0  0012 0000 00D4 00F4
[249]  0000 0000 0000 0000  0000 0000 0000 0000
[251]  0000 0000 0000 0000  0000 0000 0000 0000
[259]  0000 0000 0000 0000  0000 0000 0000 0000
> exit
Done
```

7 Compiler Design

7.1 Top Level Design

```
1  Request Source and Destination Filenames
2  Compile Pass 1
3  If Pass 1 was successful
4    Compile Pass 2
5  End If
```

7.2 Compile Pass 1

```
2.1   Attempt to open Source File
2.2   If Open was not successful
2.3     Display message and return with error
2.4   End If
2.5   Clear Label Symbol Table and Address Counter
2.6   Loop Forever
2.7   Read in Line from File
2.8   If End Of File
2.9     Close File and Return Unexpected EOF
2.10    End If
2.11    If valid command or ".word" operation
```

```
2.12      Increment Address Counter
2.13    End If
2.14    If ".label" operation
2.15      Add label into table
2.16    End If
2.17    If ".end"
2.18      Close File, Return successful
2.19    End If
2.20  End Loop
```

7.3 Compile Pass 2

```
4.1   Attempt to open Source File (must be successful)
4.2   Attempt to open Destination File
4.3   If Open was not successful
4.4     Display message and return with error
4.5   End If
4.6   Reset Address Counter
4.7   Loop Forever
4.8     Read in Line from File
4.9     If End Of File
4.10      Close both Files and Return Unexpected EOF
4.11    End If
4.12    If ".word" operation
4.13      Increment Address Counter and output operand
4.14    End If
4.15    If ".label" operation
4.16      Ignore it this time round
4.17    End If
4.18    If ".end" operation
4.19      Close both Files and Return successful
4.20    End If
4.21    If Command (all other cases)
4.22      If Command is valid
4.23        Output command with operand (except for STOP)
4.24      Else
4.25        Output blank entry
4.26      End If
4.27      Increment Address Counter
4.28    End If
4.29  End Loop
```

8 Executer / Debugger design

8.1 Top Level Design

```
1  Initialise and Clear 4096 x 16 bit words of "memory" and
```

```
-  16 bit accumulator
2  Loop Forever
3   Prompt for Command
4   Case of Command
5     HELP       Show help information
6     EXIT       Exit program
7     VIEW       View one address
8     SET        Set address to value
9     VIEWACC    View accumulator
10    SETACC     Set accumulator to value
11    GO         Execute code in trace mode
12    GOSTEP     Execute code in step mode
13    GOQUIET    Execute code in quiet mode
14    DUMP       Dump chunk of memory to screen
15    LOAD       Load code into memory
16    OTHERWISE  Invalid command
17    End Case
18  End Loop
```

Most of these commands are quite simple and will not be expanded further at this stage, as they simply involve taking one or two further parameters from the command string, and either displaying an entry or setting an entry to something else.

8.2 Execute Commands

All three execute commands (`GO`, `GOSTEP` and `GOQUIET`) will be dealt with together, as they will eventually be handled by the same function, which will have a parameter passed to indicate which mode it's running in. The numbering will be `11.?` to correspond to the `GO` command in the top level design.

```
11.1  Set Current Address to 0
11.2  Loop Forever
11.3    Fetch contents of memory pointed to by Current Address
11.4    If in Trace or Step mode, display Current Address,
--.-    Command and Accumulator
11.5    Case of Command (4 Most Significant Bits)
11.6      0   Load operand into accumulator
11.7      1   Store accumulator to operand
11.8      2   Add operand to accumulator
11.9      3   Subtract operand from accumulator
11.10     4   Set Current Address to operand
11.11     5   If accumulator is positive, set Current Address
--.--         to operand
11.12     6   If accumulator is not zero, set Current Address
--.--         to operand
11.13     7   Stop program and return to command mode
11.14     OTHERWISE Display error but continue execution (bad command)
11.15   End Case
11.16   Increment Current Address
11.17   If in Step mode
```

```
11.18     Prompt and get key for Stop, Change to Trace, Change to
--.--     Quiet or Continue
11.19     Case of key
11.20       Stop  Return to command mode
11.21       Trace Change to Trace mode
11.22       Quiet Change to Quiet mode
11.23       Continue  Continue executing in step mode
11.24     End Case
11.25   End If
11.26 End Loop
```

8.3 Load Code into Memory

```
15.1  Get Filename from Command Line
15.2  Attempt to Open File
15.3  If Open was not successful
15.4    Return Error to user
15.5  End If
15.6  While Not End Of File
15.7    Get two bytes from file
15.8    Place into "memory"
15.9  End While
15.10 Close File
```