Documentation for HMMM
(Harvey Mudd Miniature Machine)

Last updated October, 2013

Quick reference: Table of Hmmm Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System instructions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>halt</td>
<td>Stop!</td>
<td></td>
</tr>
<tr>
<td>read rX</td>
<td>Place user input in register rX</td>
<td></td>
</tr>
<tr>
<td>write rX</td>
<td>Print contents of register rX</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>Do nothing</td>
<td></td>
</tr>
<tr>
<td><strong>Setting register data</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>setn rX N</td>
<td>Set register rX equal to the integer N (-128 to +127)</td>
<td></td>
</tr>
<tr>
<td>addn rX N</td>
<td>Add integer N (-128 to 127) to register rX</td>
<td></td>
</tr>
<tr>
<td>copy rX rY</td>
<td>Set rX = rY</td>
<td>mov</td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add rX rY rZ</td>
<td>Set rX = rY + rZ</td>
<td></td>
</tr>
<tr>
<td>sub rX rY rZ</td>
<td>Set rX = rY - rZ</td>
<td></td>
</tr>
<tr>
<td>neg rX rY</td>
<td>Set rX = -rY</td>
<td></td>
</tr>
<tr>
<td>mul rX rY rZ</td>
<td>Set rX = rY * rZ</td>
<td></td>
</tr>
<tr>
<td>div rX rY rZ</td>
<td>Set rX = rY / rZ (integer division; no remainder)</td>
<td></td>
</tr>
<tr>
<td>mod rX rY rZ</td>
<td>Set rX = rY % rZ (returns the remainder of integer division)</td>
<td></td>
</tr>
<tr>
<td><strong>Jumps!</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jumpn N</td>
<td>Set program counter to address N</td>
<td></td>
</tr>
<tr>
<td>jump rX</td>
<td>Set program counter to address in rX</td>
<td>jump</td>
</tr>
<tr>
<td>jegzn rX N</td>
<td>If rX == 0, then jump to line N</td>
<td>jegz</td>
</tr>
<tr>
<td>jnezn rX N</td>
<td>If rX != 0, then jump to line N</td>
<td>jnez</td>
</tr>
<tr>
<td>jgtzn rX N</td>
<td>If rX &gt; 0, then jump to line N</td>
<td>jgtz</td>
</tr>
<tr>
<td>jltzn rX N</td>
<td>If rX &lt; 0, then jump to line N</td>
<td>jltz</td>
</tr>
<tr>
<td>calln rX N</td>
<td>Copy the next address into rX and then jump to mem. addr. N</td>
<td>call</td>
</tr>
<tr>
<td><strong>Interacting with memory (RAM)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loadn rX N</td>
<td>Load register rX with the contents of memory address N</td>
<td></td>
</tr>
<tr>
<td>storen rX N</td>
<td>Store contents of register rX into memory address N</td>
<td></td>
</tr>
<tr>
<td>loadr rX rY</td>
<td>Load register rX with data from the address location held in reg. rY</td>
<td>loadi, load</td>
</tr>
</tbody>
</table>
Table of Contents

- Introduction
- Using Hmmm
- Code Format
- Machine Layout
- The Hmmm Instruction Set
- Input and Output
- Halting the Program
- Diagnostic Features
- Examples
- Contact Information
- Advanced topics

Introduction

Hmmm (Harvey Mudd Miniature Machine) is a 16-bit, 23-instruction simulated assembly language with \(2^8=256\) 16-bit words of memory. Hmmm is written in python, and it is intended as an introduction to assembly coding in general. Programs written in Hmmm consist of numbered lines with one instruction per line, and comments.

The Hmmm implementation consists of three programs: hmmmAssembler.py, hmmmSimulator.py, and ansiHmmmSimulator.py. The assembler is used to assemble code written in Hmmm into binary files, and the simulators read binary files and execute the programs that they contain. The difference between the two simulators is that ansiHmmmSimulator.py offers a more visual interface to the debugger.

Using Hmmm at Harvey Mudd

Both the assembler and simulator for Hmmm are installed on the Computer Science Lab Macs. On any of the machines, follow the directions below to access these programs. They are also available online in the form of source code from the HMC CS5 website at http://www.cs.hmc.edu/twiki/bin/view/CS5/Resources.

Installing Hmmm

If you plan on installing them on your own machine, you will need a working python version of python on your machine, but nothing more. Download the hmmmAssembler.py, hmmmSimulator.py, and binary.py files, and the hw6pr1.py file, at least for starters. Put these in a single directory where you will also save your Hmmm programs. Open hw6pr1.py in IDLE - that is place to get started with Hmmm.

Using Hmmm

Please refer to your lecture notes and the current Hmmm assignment for an introduction to using Hmmm.


Code Format

Each line of Hmmm code consists of a line number, an instruction, and one or more arguments. The line numbers must start at 0 and be consecutive integers, and they must be placed at the beginning of the line with no preceding characters. After the line number, use at least one character of whitespace to separate the instruction. The line number corresponds directly to the memory address of the line, with free space beginning after the last line of the program.

The instruction consists of a single word; all of the instructions are composed of lowercase alphabetic characters. After the instruction, separated by any combination of whitespace and ',' characters, come 0 or more arguments, depending on the instruction.

Each argument must be either a register or a number. Registers are denoted by 'r' followed by the number of the register, as in 'r3'. Numbers must fit in 8 bits. (Their decimal value must either be in the range -128 to 127 inclusive, or 0 to 255 inclusive, depending on the particular instruction. This is because numbers are represented using a method called "twos complement".) Each argument must be separated from any preceding arguments by any combination of whitespace and ',' characters, and the line may optionally be ended with some combination of whitespace and comments.

See the examples section for examples of proper and improper syntax.

Comments in Hmmm are identical to comments in python: a '#' character begins a comment that continues to the end of the line. Comments are allowed both on empty lines and on instruction lines after the code. Comments on otherwise empty lines should not have line numbers, and will not be counted towards the line number of any following lines. Completely blank lines are also permitted.

Machine Organization

Hmmm simulates a computer that has 16 16-bit registers and 256 16-bit words of memory. When the simulator is started, it loads a program into memory starting at location 0, so a program 12 lines long has 244 free words of memory starting at location 12 (locations 0 through 11 are occupied by the program's instructions). The program counter starts at location 0 and is incremented by 1 each cycle. It can also be redirected using the various jump commands. Each cycle, the simulator reads the instruction at the memory location pointed to by the program counter and executes it. This continues until it executes a halt instruction.

15 of the 16 registers are interchangeable from a hardware standpoint (although refer to the conventions below). The only special register is r0. When used as a source operand, r0 always provides a zero; when used as a destination it discards the result.

The Hmmm Instruction Set

There are 23 different instructions in Hmmm, each of which accepts between 0 and 3 arguments. Two of the instructions, loadn and addn, accept a signed numerical argument between -128 and 127. The load, store, call, and jump instructions accept an unsigned numerical argument between 0 and 255. All other instruction arguments are registers. In the code below, register arguments will be represented by 'rX', 'rY', and 'rZ', while numerical arguments will be represented by '#'. In real code, any of the 16 registers could take the place of 'rX' 'rY' or 'rZ'. The available instructions are:

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Binary</th>
<th>Description</th>
</tr>
</thead>
</table>

https://www.cs.hmc.edu/~cs5grad/cs5/hmmm/documentation/documentation.html#Machine
<table>
<thead>
<tr>
<th>Command</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0000 0000</td>
<td>Halt program</td>
</tr>
<tr>
<td>nop</td>
<td>0110 0000</td>
<td>Do nothing</td>
</tr>
<tr>
<td>read rX</td>
<td>0000 0000</td>
<td>Stop for user input, which will then be stored in register rX (input is an integer from -32768 to +32767). Prints &quot;Enter number: &quot; to prompt user for input</td>
</tr>
<tr>
<td>write rX</td>
<td>0000 0000</td>
<td>Print the contents of register rX on standard output</td>
</tr>
<tr>
<td>setn rX,#</td>
<td>0001 0000</td>
<td>Load an 8-bit integer # (-128 to +127) into register rX</td>
</tr>
<tr>
<td>load rX,rY</td>
<td>0100 0000</td>
<td>Load register rX from memory word addressed by rY: rX = memory[rY]</td>
</tr>
<tr>
<td>store rX,rY</td>
<td>0100 0000</td>
<td>Store contents of register rX into memory word addressed by rY: memory[rY] = rX</td>
</tr>
<tr>
<td>loadn rX,#</td>
<td>0010 0000</td>
<td>Load register rX with memory word at address #</td>
</tr>
<tr>
<td>storen rX,#</td>
<td>0011 0000</td>
<td>Store contents of register rX into memory word at address #</td>
</tr>
<tr>
<td>addn rX,#</td>
<td>0101 0000</td>
<td>Add the 8-bit integer # (-128 to 127) to register rX</td>
</tr>
<tr>
<td>copy rX,rY</td>
<td>0110 0000</td>
<td>Set rX = rY</td>
</tr>
<tr>
<td>neg rX,rY</td>
<td>0111 0000</td>
<td>Set rX = -rY</td>
</tr>
<tr>
<td>add rX,rY,rZ</td>
<td>0110 0000</td>
<td>Set rX = rY + rZ</td>
</tr>
<tr>
<td>sub rX,rY,rZ</td>
<td>0111 0000</td>
<td>Set rX = rY - rZ</td>
</tr>
<tr>
<td>mul rX,rY,rZ</td>
<td>1000 0000</td>
<td>Set rX = rY * rZ</td>
</tr>
<tr>
<td>div rX,rY,rZ</td>
<td>1001 0000</td>
<td>Set rX = rY / rZ</td>
</tr>
<tr>
<td>mod rX,rY,rZ</td>
<td>1010 0000</td>
<td>Set rX = rY % rZ</td>
</tr>
<tr>
<td>jump rX</td>
<td>0000 0000</td>
<td>Set program counter to address in rX</td>
</tr>
<tr>
<td>jumpn n</td>
<td>1011 0000</td>
<td>Set program counter to address #</td>
</tr>
<tr>
<td>jeqz rX,#</td>
<td>1100 0000</td>
<td>If rX = 0 then set program counter to address #</td>
</tr>
<tr>
<td>jnez rX,#</td>
<td>1101 0000</td>
<td>If rX ≠ 0 then set program counter to address #</td>
</tr>
<tr>
<td>jgtz rX,#</td>
<td>1110 0000</td>
<td>If rX &gt; 0 then set program counter to address #</td>
</tr>
<tr>
<td>jltz rX,#</td>
<td>1111 0000</td>
<td>If rX &lt; 0 then set program counter to address #</td>
</tr>
</tbody>
</table>

https://www.cs.hmc.edu/~cs5grad/cs5fhmmn/documentation/documentation.html#Machine
Input and Output

Hmmm supports incredibly simplified I/O in the form of the **read** and **write** instructions:

The *read* instruction prompts the user to enter a number and puts the number into the given register. It checks overflow boundaries and re-prompts the user on bad input. It can also be used as an impromptu break point because if bad input is entered, further input of 'q' can be used to halt program execution. The *write* instruction simply writes the given register to the user's console.

Halting The Program

The *halt* instruction ends the program, and is equivalent to a *jump* to line -1. At any point during execution, the user can type ctrl-C to stop the program, and entering 'q' at either the debug mode prompt or the bad input prompt will stop the program. Pressing ctrl-D at any prompt will also stop the program.

Diagnostic Features

The simulator program features a debug mode that is useful for diagnosing errors in the program. It is invoked using either the -d or --debug command-line options, or through the prompt when the simulator starts. To start the simulator without prompting for debug mode, use the -n or --no-debug options.

The debug mode prints information after executing each instruction, showing what instruction was just executed and where that instruction was found in memory (What the program counter was). It also prints the debug prompt. Any unrecognized input at the debug prompt causes the simulator to step one command forward. Recognized commands include 'c' or 'continue', 'd' or 'dump', 'h' or 'help', 'p' or 'print', 'q' or 'quit', and 'r' or 'run'.

Command Effect

<table>
<thead>
<tr>
<th>Command</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>continue</em></td>
<td>Causes the debugger to run through the rest of the program without prompting for debugging commands, but continuing to print debugging information.</td>
</tr>
<tr>
<td><em>dump</em></td>
<td>Immediately prints the contents of memory, printing the code lines first one-per-line followed by the initially empty portions of memory in 6 columns, and then asks for another debugging command.</td>
</tr>
<tr>
<td><em>help</em></td>
<td>Prints a short summary of the debug commands and returns to the prompt.</td>
</tr>
<tr>
<td><em>print</em></td>
<td>Prints the contents of the registers in a single column and returns to the prompt.</td>
</tr>
<tr>
<td><em>quit</em></td>
<td>Causes the program to exit immediately.</td>
</tr>
<tr>
<td><em>run</em></td>
<td>Causes the program to continue running as if it had been invoked with debug mode off: no debugging information is printed and no further prompts are given.</td>
</tr>
</tbody>
</table>

The ANSI simulator works a bit differently. In debug mode, the register and memory contents are displayed and updated in real time. The debug prompt is displayed after every instruction is executed. Additional displays include the last and next instructions (the last instruction was just executed), and a list of all inputs and outputs. The ANSI debugger does not support the 'continue', 'print' or 'dump' commands (except in no-memory mode, where it accepts 'dump'). It supports three additional commands, however: the 'n' or 'next', 'b' or 'break', and 'c' or 'clear' commands.
Command Effect

next #  Executes the next # instructions without displaying the debug prompt, after which the prompt is displayed again.

break #  Inserts a breakpoint at location # in memory. If the simulator would not otherwise display the debugging prompt before executing the instruction at that location, it does so, reentering normal debugging mode. A breakpoint cancels the next and run commands when reached. The breakpoint is indicated by an arrow on the memory display unless no-memory mode is on.

clear #  Removes a breakpoint at location # in memory. If there is no breakpoint at that location, the command is not valid and is interpreted as a single step command.

If the ANSI debug mode is to display the entire contents of memory and the registers, a nonstandard terminal size is required. To ensure proper functionality, a terminal width of 200 characters and height of 52 characters should be used, although the minimum specifications depend on the length of the binary file being executed. Because such a large terminal size is usually impossible on standard-resolution (1024x768) monitors without a very small font size, there is a --no-memory option (or simply -m) that disables memory display. This allows the debugging mode to be used on smaller terminals. To view the contents of memory in no-memory mode, enter 'd' or 'dump' at the debugging prompt.

Examples

- Good Code
- Bad Code
- Improper Syntax

Good Code:

This is a well-written and -commented program that takes two numbers, echoes the first, and returns the first divided by the second, or 0 if the second is 0.

```
# program title
# author and date
# descriptive comment
0    read r1        # read dividend from the user
1    write r1       # echo the input
2    read r2        # read divisor from the user
3    jeqz r2, 7    # jump to 7 if trying to divide by 0
4    div r3, r1, r2 # divide user's parameters
5    write r3       # print the result
6    halt
7    setn r3, 0    # 0 is the return for division by 0
8    write r3       # print the result
9    halt
```

This is the output from the assembler when the program is assembled (note that the assembler truncates comments to make the lines fit neatly on your screen):

```
--------------
| ASSEMBLY SUCCESSFUL |
--------------
0 : 0000 0001 0000 0001 0    read r1        # read dividend from
1 : 0000 0001 0000 0010 1    write r1       # echo the input
2 : 0000 0010 0000 0001 2    read r2        # read divisor from
```
This is an example of running the simulator on the binary:

(1)% hmmSimulator
Enter input file name: good.b
Enter debugging mode? no
Enter number: 16
16
Enter number: 4
4
Execution Terminated Successfully
(2)%

Bad Code:

This is the same program, with the same functionality. However, this is very low-quality code.

0    read r1
1    write r1
2    read r2  # read r2
3    jeqz r2, 7
4    div r3 r1, r2  # divide user's
5    write r3  # print the result
6    halt
7    setn r3, 0  # 0 is the return
8    write r3  # print the result
9    halt

This is the assembler output for the above code:

-------------------
| ASSEMBLY SUCCESSFUL |
-------------------

0 : 0000 0001 0000 0001 0 : read r1
1 : 0000 0001 0000 0010 1 : write r1
2 : 0000 0010 0000 0001 2 : read r2  # read r2
3 : 1100 0010 0000 0111 3 : jeqz r2, 7
4 : 1001 0011 0001 0010 4 : div r3 r1, r2
5 : 0000 0011 0000 0010 5 : write r3
6 : 0000 0000 0000 0000 6 : halt  # end
7 : 0001 0011 0000 0000 7 : setn r3, 0
8 : 0000 0011 0000 0010 8 : write r3
9 : 0000 0000 0000 0000 9 : halt

This is the result of running the simulator on the assembled version of the program:

(1)% hmmSimulator -f bad.b -n
Enter number: 32
32
Enter number: 4
8
Improper Syntax:

This is the same program a third time, this time modified to demonstrate improper syntax. This program will not assemble.

```
# program title
# program title
# author and date
# descriptive comment
0    read r1,     # trailing characters are not allowed
1    write[r1]   # no grouping symbols allowed
2    read r2, r3  # too many arguments here
3    jeqz r2, r7  # second argument must be a number
4    div r3, rlr2 # arguments must be separated
5    write 3      # write argument must be a register
6    halt         # line number is incorrect
7    setn r3, 128 # maximum number for setn and addn commands is 127
                  # (min is -128)
9    writer3      # instruction must be separated from argument
10   halt         # instruction must be separated from line number
```

Here is the assembly output for this program, demonstrating the error messages for the errors listed above:

ARGUMENT ERROR:
WRONG NUMBER OF ARGUMENTS.
DETECTED 2 ARGUMENTS, EXPECTED 1 ARGUMENTS.
read r1,

SYNTAX ERROR ON LINE 1:
1    write[r1]   # no grouping symbols allowed

ARGUMENT ERROR:
WRONG NUMBER OF ARGUMENTS.
DETECTED 2 ARGUMENTS, EXPECTED 1 ARGUMENTS.
read r2, r3

ARGUMENT ERROR:
'r7' IS NOT A VALID NUMBER.

ARGUMENT ERROR:
WRONG NUMBER OF ARGUMENTS.
DETECTED 2 ARGUMENTS, EXPECTED 3 ARGUMENTS.
div r3, rlr2

REGISTER ERROR:
'3' IS NOT A VALID REGISTER.

BAD LINE NUMBER AT LINE 6:
LINE NUMBER: 5 EXPECTED: 6

ARGUMENT ERROR:
'128' IS OUT OF RANGE FOR THE ARGUMENT.

OPERATION ERROR:
'writer' IS NOT A VALID OPERATION.
SYNTAX ERROR ON LINE 9:
10halt # instruction must be separated from line number

***** ASSEMBLY TERMINATED UNSUCCESSFULLY *****
ASSEMBLY RESULTS:

0 : ***ARGUMENT ERROR HERE*** 0 read r1, # trailing
1 : ***SYNTAX ERROR HERE*** 1 write[r1] # no grouping
2 : ***ARGUMENT ERROR HERE*** 2 read r2, r3 # too many arguments
3 : ***ARGUMENT ERROR HERE*** 3 jeqz r2, r7 # second argument
4 : ***ARGUMENT ERROR HERE*** 4 div r3, r1r2 # arguments must be
5 : ***REGISTER ERROR HERE*** 5 write 3 # write argument
6 : ***BAD LINE NUMBER HERE*** 6 halt # line number is
7 : ***ARGUMENT ERROR HERE*** 7 setn r3, 128 # maximum number for
8 : ***OPERATION ERROR HERE*** 9 writer3 # instruction must
9 : ***SYNTAX ERROR HERE*** 10halt #

Although it may not be obvious from the example above, the assembler will stop trying to assemble each line of code as soon as it finds an error. This means that if a line of code has multiple errors in it, only one error will be reported initially. Once that error is fixed, the next error will be reported if it is still there.

Contact Information

If this document is inaccurate or you believe that you have found a bug in either the assembler or the simulator, or even if you think that something should be added to any of these, please contact the CS5 staff at the usual cs5help address.