Humane Assembly Language Tools (HALT)

User's Guide

Contents

This page intentionally left blank.

Preface

This document utilizes many hyperlinks to other resources. If you are reading a hard copy, you are not making full use of this guide. Please retrieve the latest electronic version in PDF format from the project website, at http://halttool.berlios.de There you can also find Frequently asked questions, screenshots of the latest development version, project news, stable version releases, and more.
Also, you may be interested to visit the project code forge hosted at BerliOS, available at http://developer.berlios.de/projects/h
This is an even more extensive website where you can download any previous release, view screenshots of the development version, join user mailing lists, report and track bugs, browse code from any previous development period, and participate in HALT public forum discussions.

Introduction

Humane Assembly Language Tools (HALT) is a toolkit for user-friendly development and inspection of Motorola 68000 assembly language. The philosophy behind HALT is to make assembly language as accessible as possible to a broad audience of programmers. That philosophy is realized through a simple, colorful human interface, connecting the user to advanced tools such as a powerful lexxer/parser, a bare M68000 machine language interpreter, an M68000 assembler and translator, and various debugging and execution tools.

HALT provides a simplified run-time visualization for the internal working environment of a virtual M68000 machine. The visualization updates the state of the stack as new code is typed into the working project. The simulator is also a visualization environment for program execution, and displays the contents of registers and memory as it occurs once the assembly instructions have been successfully translated and interpreted. All this is done at development time, within one simple and easy-to-use framework, so the developer's time to product is minimized. HALT also functions well in a teaching environment, as it follows the KISS principle: Make everything as simple as possible, but no simpler. The bright and simple display of information make the user interface a fun and powerful way to learn and develop Motorola 68000 assembly language code.

HALT is also a tool for developers. HALT produces only machine instructions which are in a strict subset of the Motorola 68000 machine instruction set. Therefore, any program in HALT should also run on any machine that implements the basic 68000 instruction set.

Obtaining HALT

Release versions of HALT for Windows, OSX, and Linux may be downloaded from the website at: http://halttool.berlios.de The current development version is available from the Subversion repository, which is located at https://svn.berlios.de/svnroot/repos/halttool/trunk

Getting Started

Windows

The releases include the most recent Stable Windows executable. Simply unzip to a folder and run halt.exe. There are several example programs available in the examples subdirectory, which can be loaded by specifying an argument to halt.exe at the command line. You may need to copy glut.dll into your %WINROOT%System directory. If you have a Development version obtained via Subversion, you must

compile the code against the OpenGL and GLUT header files and libraries, which may be found at NeHe.

OS X

Regardless of whether you have the Stable release of HALT or the Development version, untar the package and using the make command. This should produce the binary executable 'halt' in the same directory. There are several example programs available in the examples subdirectory, which can be loaded by specifying an argument to halt at the command line.

Linux

Regardless of whether you have the Stable release of HALT or the Development version, untar with tar -xvzf and type make. This should produce the binary executable 'halt' in the same directory. You can also use make install if you wish to make HALT available to all users, but you must have root privileges. Note that you must have the OpenGL, GLU, and GLUT libraries installed before you can compile HALT. There are several example programs available in the examples subdirectory, which can be loaded by specifying an argument to halt at the command line.

Components of the User Interface

Program Editor

Figure 1: Program Control and Program Editor Panes

The program editor is where you will write and edit all Motorola 68000 instructions. It represented by a square box with an I-beam cursor. As you type your instructions into the Program Editor, you can observe the effect that they have on the system's stack space within the Memory and Register viewer.

Memory and Register Viewer

The stack space is represented by the outer ring of circles in our memory viewer. Commands are represented by Green circles, Data members by Blue circles, and other Op-codes by Orange circles. The pipe | delimits the start and end of the system stack. The system stack space between the last element of the front of the stack, and the first element in the end of the stack is irrelevant to the programmer, so it is not shown. Instead it is represented by an ellipsis (...).
Data Registers are represented similarly, as the inner ring of Blue circles. Starting at the | and traversing the ring counter-clockwise, the registers are arranged in ascending order from d0 to d7.
Both registers and memory locations exhibit "glowing". As the amount of time since their last access increases, they become darker. A quick glance at the memory and register Viewer will indicate which registers and memory locations are currently active.

Figure 2: HALT's Memory and Register Viewer

Memory and Register Inspector

At any point, you can click on a memory cell or a register cell, and a new window, the Memory and Register Inspector, will appear. It displays the contents of the currently selected memory location, as well as its type and how many clock cycles ago it was touched.

Virtual Machine

The Virtual Machine is controlled directly through the Control pane. The Reset button sets the program counter to the first line of the program. The Step button steps through the program a single line at a time, for debugging purposes. Finally, the Green button runs the entire program without interruption until either a stop instruction or a runtime error is reached. The control panel pane is shown here in the same figure as the Program Editor pane.
As for the specifications of the virtual machine, the stack space consists of an arbitrarily large amount of memory. Like the Motorola 68000 architecture, eight data registers (d0, d1, ... d7) of length 16-bits, and eight address registers (a0, a1, ... a7) of length 32-bits are provided. Additionally, several status bit registers are provided. They are "Z"ero, "C"arry, o"V"erflow, e"X"tend, and "N"egative.

Addressing Modes

Data Register Direct

`dn`

Data register dn is manipulated directly. This is the most common addressing mode for most tasks.

Address Register Direct

`an`

Address register an is manipulated directly. The address stored by an can be changed, much like manipulating a pointer.

Address Register Indirect

`(an)`

The memory pointed to by address register an is manipulated directly. This is similar to de-referencing a pointer.

Address Register Indirect with pre-decrement

`-(an)`

The memory pointed to by address register an is decremented to the previous memory address. The contents of that memory address are then manipulated directly. This is similar to decrementing and then de-referencing a pointer.

Address Register Indirect with post-increment

`(an)+`

The contents of the memory pointed to by address register are manipulated directly. The memory address of the register is then incremented to the next memory address. This is similar to de-referencing and then incrementing a pointer.

Symbolic Address Indirect

`foo`

The contents of the memory pointed allocated after the label foo are manipulated directly. This is especially helpful in manipulating arrays or other arbitrary data structures available on the stack. Example:

```
fibonacci word[10] = 1,1,2,3,5,8,13,21,34,55
lea fibonacci,a1
add (a1)+, d1 ;add the first  fibonacci number to d1
add (a1)+, d1 ;add the second fibonacci number to d1
```

Literal Mode

```
3
```

The literal specified represents itself. The following code will place a 3 in register d0.

```
move 3,d0
```

Supported Instructions

ADD

Arithmetic ADD takes 2 arguments. It arithmetically adds the contents of the first argument with the contents of the second argument, and stores the result in the second argument.

AND

Bitwise AND takes 2 arguments. It performs a bitwise AND on the contents of the first argument and the contents of the second argument, and stores the result in the second argument.

BGE

Branch Greater-Than-Or-Equal BGE takes 1 argument. If the previous CMP instruction resulted in a "greater-than-or-equal" state being placed on the status registers, then the program counter will move to the label taken as argument. Otherwise, the instruction will perform a NOP.

BGT

Branch Greater-Than BGT takes 1 argument. If the previous CMP instruction resulted in a "greater-than" state being placed on the status registers, then the program counter will move to the label taken as argument. Otherwise, the instruction will perform a NOP.

BLE

Branch Less-Than-Or-Equal BLE takes 1 argument. If the previous CMP instruction resulted in a "less-than-or-equal" state being placed on the status registers, then the program counter will move to the label taken as argument. Otherwise, the instruction will perform a NOP.

BLT

Branch Less-Than BLT takes 1 argument. If the previous CMP instruction resulted in a "less-than" state being placed on the status registers, then the program counter will move to the label taken as argument. Otherwise, the instruction will perform a NOP. It is not a bacon-lettuce-and-tomato sandwhich.

BNE

Branch Not-Equal BNE takes 1 argument. If the previous CMP instruction resulted in a "not-equal" state being placed on the status registers, then the program counter will move to the label taken as argument. Otherwise, the instruction will perform a NOP.

BRA

Branch Unconditional BRA takes 1 argument. BRA moves the program counter to the label taken as argument.

CLR

Clear CLR takes 1 argument. It sets the contents of the argument to 0.

CMP

Compare CMP takes 2 arguments. It subtracts the first argument from the second argument, and sets the status registers depending on the result. N Set if the result is negative, cleared otherwise. Z Set if the result is zero, cleared otherwise. CMP is usually followed by a conditional branch.

DIV

Divide DIV takes 2 arguments. It arithmetically divides the contents of the first argument by contents of the second argument, and stores the quotient in the second argument.

EOR

Exclusive-OR EOR takes 2 arguments. It performs a bitwise exclusive OR on the contents of the first argument and the contents of the second argument, and stores the result in the second argument.

LEA

LEA takes 2 arguments. It loads the address of data or instructions associated with a label (first argument) into the second argument.

MOVE

MOVE takes 2 arguments. It moves the contents of the first argument into the location of the second argument, overwriting the contents of the second argument.

MUL

Multiply MUL takes 2 arguments. It arithmetically multiplies the contents of the first argument with the contents of the second argument, and stores the result in the second argument.

NEG

Negative NEG takes 1 argument. It produces the arithmetic negative of the argument, and stores it in the same place.

NOP

No-op NOP takes 0 arguments. It is the Null operator, or no-op. This instruction performs no operation for one clock cycle. It is used for synchronization, pipelining, and preventing instruction overlap.

NOT

Bitwise NOT takes 1 arguments. It performs a bitwise inversion on all the bits in the argument, storing the result. This is also sometimes called the 1s-complement operation.

OR

Bitwise OR takes 2 arguments. It performs a bitwise OR on all the bits in the first argument, storing the result in the second.

STOP

STOP takes 0 arguments. STOP signifies the end of a program.

SUB

Arithmetic Subtraction SUB takes 2 arguments. It arithmetically subtracts the contents of the first argument with the contents of the second argument, and stores the result in the second argument.

Additional Language Features

Arrays

A labeled array can be produced on the stack using the syntax

```
label datatype[NUMBER OF ELEMENTS]
```

An anonymous array is possible, but there will be no way to access the data, wasting stack space. Additionally, the array elements may be initialized with a list of values, using this syntax. Note that if a full initialization list is not provided, the remaining elements are initialized to 0.

```
word[3] = 1,2,3
```

Comments

Anything found after a semicolon is ignored, until the next line. This is similar to C++ style comments.

Labels

Any identifier in the first column which is not a reserved word is considered a label. Labels should be unique, and may be used in logical branch instructions (BRA, BLE, BLT, BGE, BGT, BNE). Labels are used to define variables or to provide control points for loops.

HALT Developer's Notes

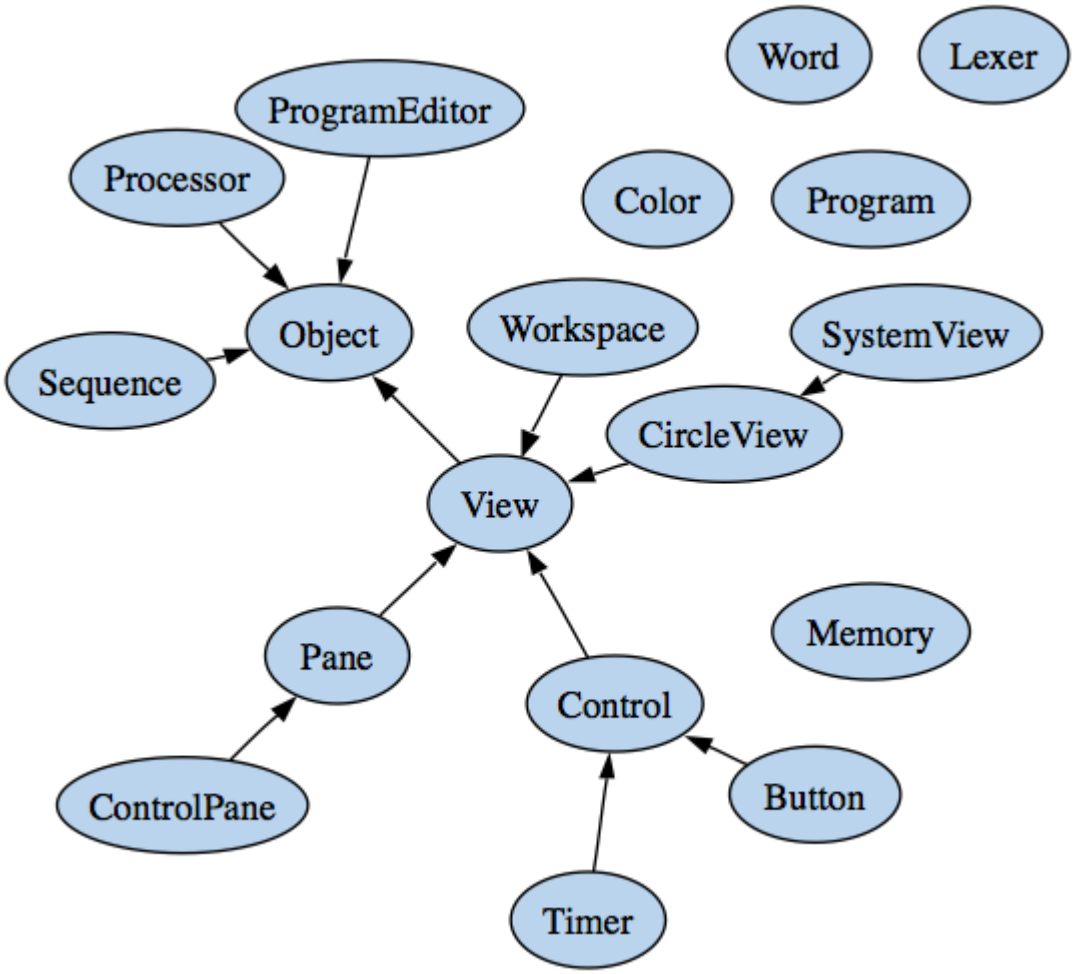

Figure 3: Class Dependencies Diagram