

The Happy Assembly Class

Computer Engineering 012c
University of California, Santa Cruz

Alexandra Carey <fire@cse.ucsc.edu>
Cliffon McIntire <mcintire@acm.org>
Joel Ferguson <fjf@cse.ucsc.edu>
Richard Hughey <rph@cse.ucsc.edu>

Table of Contents

Table of Contents	ii	
Table of Figures	v	
Table of Tables	vi	
0	The Happy Assembly Class Lab Manual	1
0.1	Abstract	1
0.1.1	Course Philosophy	1
0.2	Revision History	2
0.3	Notations Used In This Manual	2
0.4	List of Acronyms	3
0.5	Overview of Academic Honesty	4
1	For the Lab	5
1.1	Lab Preparation	5
1.2	Programming Style	5
1.2.1	Comments in code	5
1.3	Your README	6
1.4	Submission Instructions	6
1.5	Grading	7
1.5.1	Getting back grades	7
1.5.2	Getting back grades with checkoffs in lab	7
2	MIPS and MAL	11
2.1	MIPS	11
2.2	Wonderful World of MAL	11
2.2.1	Registers and instructions	11
2.2.2	MAL directives	12
2.2.3	More information	12
2.3	Development Software	14
2.3.1	SSH, Exceed, and xspimsal	14
2.3.2	Xspimsal control buttons, displays, and messages	17
2.3.3	Alternatives to xspimsal	19
2.4	MALgorithms	20
2.4.1	Input and print a number, character by character	20
2.4.2	Extract and print 8 bits	22
2.4.3	What is RPN?	24
2.4.4	Tips on procedure calls	26
2.4.5	MAL to TAL conversion	27
2.4.6	Help on MIPS machine code	28
2.5	Code Examples and Explanations	30
2.5.1	MAL character I/O	30
2.5.2	Procedure calls	32

3	Motorola HC11	35
3.1	Introduction to Microcontrollers & HC11	35
3.2	Beautiful Bounties of HC11	35
3.2.1	HC11 hardware	35
3.2.2	And language	36
3.2.3	Variables in HC11	39
3.2.4	More information	39
3.3	Development Software	40
3.3.1	Building and running HC11 executables	40
3.3.2	Alternatives to TeraTerm	43
3.4	HC11 Algorithms	43
3.4.1	Rot-13	43
3.4.2	What is recursion?	44
3.4.3	Recursion: Fibonacci in HC11	47
3.4.4	HC11 procedure calls	48
3.4.5	Notes on interrupts	48
3.5	Code Examples and Explanations	49
3.5.1	HC11 I/O	49
3.5.2	Procedure calls	50
3.5.3	Light patterns	53
3.5.4	Interrupts	54
3.5.5	Pulse accumulator	54
3.5.6	LCDINT	57
4	Example Labs	59
4.1	Are Only Examples	59
4.2	MAL	59
4.2.1	Efficiency	59
4.2.2	Displaying the Register (MAL I/O)	62
4.2.3	Self-modifying code	63
4.2.4	Stack	64
4.2.5	Procedure calls	66
4.2.6	Negabinary arithmetic	71
4.2.7	Jump table calculator	72
4.2.8	Variable-sized stack operations	75
4.3	HC11	77
4.3.1	I/O: ROT-13	77
4.3.2	Procedure calls	79
4.3.3	Interrupts	83
4.3.4	Hi-Low Game	84
5	Appendix: Summaries and Classwork Help	86
5.1	Logical Operations	86
5.2	Binary Number Conversion Summary	87

5.2.1	Binary	87
5.2.2	Example converting decimal to radix point	88
5.2.3	Octal	89
5.2.4	Hexadecimal	89
5.2.5	Examples	89
5.3	Floating Point Summary	90
5.3.1	Floats are not reals!	90
5.3.2	Normalized numbers	90
5.3.3	SPFP	90
5.3.4	The conversion process	90
5.3.5	Example: converting from decimal to SPFP	91
6	Appendix: Important Tables	92
6.1	MIPS Table of Registers	92
6.2	MIPS System Calls	93
6.3	ASCII Table	94
6.4	MAL Instruction Set	95
6.5	TAL Instruction Set, Machine Code	97
6.6	HC11 Registers	99
6.7	v2_18 Summary	100
7	References	101
8	Index	102

Table of Figures

Figure 1	README template.	8
Figure 2	Example README.	9
Figure 3	Example checkoff sheet.	10
Figure 4	Opening SSH. Use whichever server CATS prefers today.	15
Figure 5	Using Exceed to log in. Note the diagram shows the incorrect host!	15
Figure 6	Xspimsal screenshot.	18
Figure 7	Algorithm: Integer input, character by character.	20
Figure 8	Algorithm: Printing a number, character by character. Note that pseudocode to find Divisor is not provided.	21
Figure 9	Example: Printing the ASCII character 'a', bit by bit.	22
Figure 10	Algorithm: Printing the last 8 bits of a character.	23
Figure 11	Code fragment showing the use of jal and jr.	26
Figure 12	Nested jal wiping out \$ra.	26
Figure 13	Program: An example of MAL character I/O.	31
Figure 14	Program: MAL procedure calls with various methods of parameter passing. 34	
Figure 15	Diagram of HC11 internal registers [7].	37
Figure 16	Connecting a microkit to the computer via TeraTerm.	41
Figure 17	HC11 command menu.	41
Figure 18	Algorithm: Rot-13 in pseudo-C.	44
Figure 19	Nested instances in a series of recursive calls.	45
Figure 20	Recursive nested instances of a call to factorial.	46
Figure 21	Algorithm: Fibonacci defined recursively in pseudo-C.	47
Figure 22	Algorithm: Fibonacci coded in C.	47
Figure 23	Program: Simple HC11 I/O.	50
Figure 24	Program: Non-recursive factorial in HC11.	52
Figure 25	Program: Code fragment of a binary-counter light pattern.	53
Figure 26	Program: Example interrupt service routine (ISR).	54
Figure 27	Program: Using the pulse accumulator to generate a periodic time tick and pseudorandom numbers	56
Figure 28	Program: Displaying a 16-bit positive integer to the LCD.	58
Figure 29	The sign, exponent, and mantissa of a SPFP number.	91
Figure 30	HC11 Registers.	99

Table of Tables

Table 1	Some acronyms used in this manual.	3
Table 2	MIPS table of registers.	13
Table 3	MAL directives.	13
Table 4	Useful page numbers from the textbook.	14
Table 5	Xspimsal control buttons.	17
Table 6	Xspimsal display areas.	17
Table 7	MAL to TAL conversion summary.	28
Table 8	HC11 valid prefixes.	36
Table 9	Some HC11 instructions and their MAL "equivalents."	38
Table 10	Declaring HC11 variables. Note all variables must be initialized.	39
Table 11	Summary of logical operations. A and B are input values; the last five columns show outputs of logical operations.	86
Table 12	Number conversions of a few numbers	89
Table 13	MIPS table of registers.	92
Table 14	System calls and the associated value in \$v0 (\$2).	93
Table 15	ASCII table of elements.	94
Table 16	MAL instruction set. Modified from [1].	96
Table 17	TAL and machine code. Modified from [1].	98
Table 18	Summary of procedures in v2_18.	100

0 The Happy Assembly Class Lab Manual

*How sweet are your numbers converted to i!
And all of those zeroes and ones going by...
Ext2.fs, assembly, and RISC,
A whole Linux kernel loaded in your RAM disk!*

*How sweet are your makefiles, gcc -lm
Your kindness must from your 0x300 stem
For in 10-base-T and in ternary too,
I can't figure out what's as perfect as you!*

0.1 Abstract¹

Welcome to Computer Engineering 12c, Introduction to Computer Organization. In this class, you will use assembly language as a tool to learn how a computer stores, organizes, and accesses data.

This manual contains requirements, hints, and algorithms you may find useful throughout the quarter. In the class and lab you will learn computer architecture through assembly language programming.

A computer executes series of ones and zeros, called *bits*. To humans, these bits are difficult to understand, so we use assembly language (ASM). ASM is the closest usable language to machine language and matches closely what the machine actually does. By programming in assembly you will see and be able to control everything that is happening internally (inside the computer) and do some things that high level languages such as C++ and Java don't allow you to do.

You will learn assembly language using two models: the MIPS machine, which is a load-store architecture, and the accumulator-based HC11.

In this section, you will find revision history, notations, and acronyms used in this manual, as well as an important note on academic honesty.

0.1.1 Course Philosophy

This course will introduce you to computer engineering by allowing you to manipulate the machine on the lowest level possible. Assembly language is the exact median between hardware and software. You will program software routines that interact directly with the hardware!

0.2 *Revision History*

The lab manual was originally created for Summer Session 2000. It was compiled from Web pages, newsgroup posts, and class notes. Cliff McIntire, Joel Ferguson, and Richard Hughey are all previous (and great!) instructors of Computer Engineering 12c, and Alexandra Carey (fire@cats.ucsc.edu) is an ambitious course assistant. Most of the information in this lab manual is a collaborative effort of the main authors, arranged by fire.

Date	Function	Author
12/02	Updated from old version (last manual file corrupt). Re- and auto-numbering. Logic diagrams by awhill.	fire
7/00	Created	fire
9/00	Slight modification	fire
12/00	Changed formatting of TOC, added more labs, added coding examples	fire
2/00	Proofreading, more examples for Technical Writing final project Added appendices, screenshots, tables, and figures.	fire

0.3 *Notations Used In This Manual*

Normal text and explanations are denoted in normal font.

A UNIX command, example code, or filename is displayed in fixed-width font.

A variable or definition is displayed in *italics*.

0.4 List of Acronyms

The following table shows some acronyms used in this manual.

<i>ASCII</i>	American standard code for information interchange
<i>CATS</i>	Computer and technology services
<i>CISC</i>	Complex instruction set computer
<i>CCR</i>	Condition codes register
<i>CR</i>	Carriage return
<i>FTP</i>	File transfer protocol
<i>I/O</i>	Input/output
<i>IRQ</i>	Interrupt request
<i>JAL</i>	Jump and link (MIPS instruction)
<i>JR</i>	Jump register (MIPS instruction)
<i>LCD</i>	Liquid crystal display
<i>LED</i>	Light-emitting diode
<i>LF</i>	Line feed
<i>LSB</i>	Least significant bit/Byte
<i>MAL</i>	MIPS assembly language
<i>MIPS</i>	Microprocessor without interlocked pipeline stages
<i>MSB</i>	Most significant bit/Byte
<i>OS</i>	Operating system
<i>PC</i>	Program counter
<i>PWD</i>	Present working directory (UNIX)
<i>RA</i>	Return address
<i>RISC</i>	Reduced instruction set computer
<i>RPN</i>	Reversed polish notation
<i>SCI</i>	Serial communications interface
<i>SAL</i>	Simple abstract language
<i>SP</i>	Stack pointer
<i>TAL</i>	True assembly language

Table 1 Some acronyms used in this manual.

0.5 Overview of Academic Honesty

Academic honesty means you may not try to pass off somebody else's work as your own.

No collaboration is allowed on assignments unless explicitly permitted in the assignment specifications. When permitted, collaboration must be explicitly acknowledged, and must be with those currently taking the class. An electronic copy is never ever ever allowed—be sure to rewrite code you discussed "in your own words." Failure to give credit when collaboration is allowed is a form of academic dishonesty and can be grounds for failure of the course and dismissal from your major.

Please see the current syllabus for more information.

The UNIX command `webster plagiarize` gives this definition.

pl.a.gia.rize \pla-j*-ri-z also -je-*-\ vt : to steal and pass off as one's own (the ideas or words of another) to present as one's own an idea or product derived from an existing source—*pl.a.gia.riz.er* n

1 For the Lab

1.1 Lab Preparation

You will be attending biweekly lab sessions. In the lab, you will be working out programs and projects, either by yourself or in small groups, under the supervision of your lab tutor(s) and TA(s). Don't worry—it's fun!

Before coming to lab, be sure you have (or are obtaining) all of the following.

- The course textbook, *A Programmer's View of Computer Architecture* [1]
- A CATS account—for more information and instructions, see <http://www2.ucsc.edu/cats/cgi-bin/register.pl> [2]
- This manual [3]
- A spot in one of the labs. Enroll early! If you cannot enroll in your preferred lab, e-mail the lab tutor(s) in charge of the lab. Check the course Web page, <http://www.cse.ucsc.edu/classes/cmpe012c/> [4].

This section covers programming style, your README, submitting your files, and how you will be graded.

1.2 Programming Style

*Place blocks of .data before blocks of .text
Call functions by names that will tell you what's next.
Use comments in code not to clutter but aid
your peers to debug and your graders to grade!*

Let's make this easy for everybody! If you follow the style specifications, the tutors and graders will have an easier time reading your program.

- In MIPS: Place blocks of `.data` before `.text`.
- In HC11: Place variable declarations before code.
- Use self-explanatory function names.
- Follow comment specifications.
- Name your README and files as requested.
- Document your sources.

1.2.1 Comments in code

A *block comment* is a big comment before a block of code that tells you what the code will do. Place a block comment at the beginning of every major procedure or code block. You will forget which registers contain what values! In MIPS, you have 32 registers to monitor. The block comment should contain the general functionality of the procedure, as well as register listings: What are you storing in which variable or register?

The top of each program should have a block comment like this,

```
# filename
# Your Name, e-mail address
# Lab number, Date
#
# Brief synopsis of what the program does
```

and each procedure also should have a block comment, such as

```
procedure_name:
# What the procedure does
# Preconditions, postconditions, side effects
# Any registers used
```

In-line comments, or comments beside a line of code, should be used in moderation. They should be there to clarify, not clutter. Avoid discussing the program's algorithm in the comments. For examples of comments in the code, see the program examples (with some comments) in this lab manual.

Remember that we want to minimize the in-line comments. Most of the time, in-line comments are cluttering!

1.3 *Your README*

The README, a very important file, contains detailed information about your program (all the stuff you could not put in comments). It does not have to be a novel, but it should contain general and in-depth information about your program, including algorithm explanation and anything that went wrong.

The README should be named just that—README, in all caps. It should be in UNIX format, with no lines wrapping on a standard terminal window (80 columns).

Note: A bad or missing README will drop your score by one full check-grade!

You can use the template in the figure on the following page to generate your own README. There is also a completed example README.

1.4 *Submission Instructions*

You will be submitting weekly lab assignments into the 12c locker using submit on the CATS UNIX system. The submit command should work like this.

```
submit cmpe12c.qqq Lab#.tutor README myfile1.mal
```

Here, "qqq" is the quarter, # is the lab number, *tutor* is the login name (and e-mail address) of your tutor, and all other arguments are the files you are submitting. For example, if I were enrolled in *fire*'s lab in Fall 2003, I would submit lab1.mal and the README as follows.

```
submit cmpe12c.f03 Lab1.fire README lab1.mal
```

Be sure that you are enrolled in the laboratory that you are attending; the lab tutors will have a list of those who are in their lab session.

1.5 Grading

Lab grading will be done on a lab-by-lab basis, with 70% the usual passing mark. There may be opportunities to earn extra credit!

You must pass each lab assignment to pass the lab portion of the class. There will (usually) be a resubmission locker for failed or missed labs.

What you will be graded on will vary each week. Be sure to read carefully the assignment specifications and to include each part of the assignment in your submission.

1.5.1 Getting back grades

After submitting the program file and README, your lab tutor will assign a grade to your work and will e-mail it to you. If you do not receive an e-mail with your grade, make sure to let the tutor know!

1.5.2 Getting back grades with checkoffs in lab

There are a number of ways your tutor can let you know how you are doing in lab. Some tutors choose to give out grades via e-mail. Others have a Web-based system for grade disbursement. The most common way to get your grades back is to receive a checkoff in lab.

Recently we began giving checkoffs in lab. This is an optional way for you to know in advance what your grade will be for a given lab assignment. To get a checkoff, your program must be functional and you must be able to explain any part of the program if asked. *You will still be required to submit the program and README into the lab locker.* You will receive written confirmation of your checkoff which is your tentative grade for that particular lab assignment. Remember that a poor or missing README will result in a lower grade!

The shows an example checkoff sheet. The top half is for the tutor to keep on file, the bottom half is the student confirmation.

```
Your Name
e-mail address at CATS
Lab number, lab tutor

NOTES TO GRADER: Any notes to the grader—is there a special
way you handle I/O? Did you debug this on a different program
than what is being used to grade?

INSTRUCTIONS: Instructions on use of program.

ALGORITHM: How the program works.
  o algorithmically: what you wanted.
  o functionally: what you programmed.

WHAT WENT WRONG:
  If you know where errors are coming from but don't
  know how to fix them, indicate this. BE SPECIFIC. If
  you don't show me that you know what you're doing, I
  can't assume you know what's going on.

Anything else you're supposed to include by the program
description.

SPECIAL THANKS: Make sure it's OK to work together before
you start collaborating! Who helped you? With whom did you
develop your algorithm? Something like "I worked with Foo
Bar on the debugging the bit printing algorithm" is fine,
though please be specific. And give credit. Give credit.

COMMENTS: Any other comments! Comments in the READMEs are
great. What did you think of this lab? Was it a waste of
time, or did you really learn something?
```

Figure 1 README template.

```
Alexandra Carey  
fire@cats.ucsc.edu  
Lab 1, Tutor: Cliff
```

```
NOTE TO GRADER: I debugged this program on spim, a non-  
graphical version of xspimsal. It handles input differently  
(you will have to press enter after every input).
```

```
INSTRUCTIONS: Wait for prompt. Input two numbers separated  
by a comma. Output will be the square of their difference,  
or ERROR if cannot be computed.
```

```
Sample I/O:
```

```
Input two numbers separated by comma!  
>> 12,15  
9  
Input two numbers separated by comma!  
>> 10,100  
ERROR
```

```
ALGORITHM: This program takes two numbers and converts them  
from their ASCII representations to the integers. This is  
code from lab 2. These integers are pushed onto the stack.  
Then, in RPN format, they are pulled from the stack and the  
difference is taken (second minus first). If the difference  
is negative, it is multiplied by -1 to make it positive (so  
the order of entering the numbers doesn't matter). I then do  
a check to see if the result will be larger than can be held  
in a register ( $2^{31} - 1$ ), (the trap handler should be the one  
to catch this, but I have written a procedure that will  
prevent this from happening). If so, an error message is  
displayed and the program loops. If not, I take the square  
of the number by multiplying it by itself, display the  
answer, and loop.
```

```
CREDIT: I worked closely with Mike R. on the input  
algorithm. I helped Cliff with outputting negative numbers.
```

```
COMMENTS: I think this assignment was great fun! I would  
like to see square roots implemented in next week's lab.
```

Figure 2 Example README.

Lab 6 Grade, Fall 2001 Tutor _____

_____@cats.ucsc.edu

Checkoff Criteria

minimum specifications

___ rot-13

___ favorite instruction (+5 others in README)

extras

___ toggle mode - rot-13, normal

___ output to LCD - ?

Overall

___ Your overall grade

Favorite instruction

_____ is the student's favorite instruction.

Tutor comments:

Student initials _____ Date _____

Lab 6 Grade, Fall 2001

_____@cats.ucsc.edu

This is the student grade verification. Void unless signed by tutor.

You will be expected to submit your program code and README by the submission deadline. Program code and grade may be re-evaluated at the discretion of the grader.

___ Your overall grade

Tutor signature _____ Date _____

Figure 3 Example checkoff sheet.

2 MIPS and MAL

2.1 MIPS

MIPS is a reduced instruction set computer (RISC) processor that runs SGI workstations, Nintendo 64, Cisco routers, the Aibo, and many other things. MIPS processors are primarily used in small appliances such as digital cameras, printers, routers, and handheld PCs. For more information on where MIPS is used, visit

<http://www.mips.com/coolApps/s3p3.html> [5].

For the purposes of our MIPS adventures, there are five levels of abstraction. A high-level language, like C++ or Java, must be converted very much to become the machine code bit pattern.

High level language → SAL → MAL → TAL → Machine Code

We will see three levels of assembly language while learning the MIPS architecture. The first is SAL, simple assembly language, which looks like an assembly language but does not let you work with registers. This is what your book uses. Unfortunately, the book uses too much SAL. We will use it as little as possible, and will work almost exclusively in MAL—MIPS assembly language. MAL allows you to see the intricacies of the machine while still concealing the details of it. The closest usable assembly language to what the computer actually does is TAL, MIPS true assembly language, which translates directly into machine code. (You may have the adventure of writing machine code yourself!)

MIPS is based on Stanford's RISC project (Sparc is based on Berkeley's) and is used in Computer Engineering 110 and 202 computer architecture classes here at the University of California.

In this section, you will find an introduction to MIPS assembly language, or MAL, the development environment, algorithms, and example MAL code.

2.2 Wonderful World of MAL

The MIPS machine has 32 registers. However, it is a load-store architecture, allowing easy access to memory. We introduce the registers and instruction set, and show some MAL directives. For more information, read this section all the way through!

2.2.1 Registers and instructions

Registers

Registers are physical components of a chip. They are different from and much faster than memory. On our MIPS machine, each register can hold 32 bits of data. There are 32 registers, some with special uses.

The following table shows the 32 registers you can use. The ones that are safe for you to use are \$s0–\$s8 (\$16–\$23), the global registers, and \$t0–\$t9 (\$8–\$15, and \$24–\$25), the temporary registers (these registers may be overwritten by procedure calls). The register named \$zero (\$0) is always zero! Don't use any other registers until you know what they mean.

Instructions

A word is thirty-two (32) bits.

There are three types of MAL instructions: Register to Register, Memory or Immediate, and Branches and Jumps. For a more detailed description of MIPS instructions, look up MIPS instructions in this manual.

1. *Register to Register.* These are instructions that do not access memory at all. The process of executing a register to register instruction is as follows: read the contents of the registers, perform the computation, and store the result in the destination register.

Example: `add $16, $16, $17`

2. *Memory and Immediate.* This category includes all instructions with memory accesses (loads and stores), as well as instructions that use an immediate. The process of executing memory and immediate instructions is as follows: read the contents of the register, add the desired offset, if necessary, go to memory, and store the result of the operation in the destination register.

Example: `add $16, $16, 25`

Example: `lw $16, 12($17)`

Example: `sw $16, 12($17)`

3. *Branches and Jumps.* Branches and jumps are control instructions, much like `goto` statements. They tell your program to go somewhere else and begin executing there. A branch is relative (difference is calculated) and can also be conditional, like an `if` statement. A jump is direct—it will simply find the address and go there.

Example: `beq $16, $0, hello`

Example: `j hello`

2.2.2 MAL directives

A *directive* is a way to tell the assembler what to expect. For instance, `.data` is a directive telling the assembler that variable declarations are coming up. Alternately, `.text` is a directive saying to expect instructions. In lab, we use directives such as `.word` and `.byte` to allocate space in the data segment of memory when we declare variables. The following table shows some common directives and their interpretations.

2.2.3 More information

For more information on registers, see page 244 in the course textbook for descriptions of special registers. The next section explains register usage through MIPS calling conventions. We also see a list of some useful page references in the textbook [1].

Register name (what MIPS calls it)	Alternate name (what you call it)	Use (what to do with it)
\$0	\$zero	Register 0 is always zero!
\$1	\$at	(reserved by the assembler)
\$2-\$3	\$v0-\$v1	Return <i>value</i> registers are used to pass function return values. Use these carefully; the compiler also uses them.
\$4-\$7	\$a0-\$a3	<i>Argument</i> registers are used to pass function call values (arguments). Use these carefully; the compiler also uses them.
\$8-\$15	\$t0-\$t7	Use <i>temporary</i> registers freely, but, by calling convention, they are not saved over procedure calls.
\$16-\$23	\$s0-\$s7	Use <i>global</i> registers freely, but, by calling convention, you should save them over procedure calls.
\$24-\$25	\$t8-\$t9	More <i>temporary</i> registers
\$26-\$27	\$k0-\$k1	(reserved by the OS)
\$28	\$gp	<i>Global pointer</i>
\$29	\$sp	<i>Stack pointer</i> —your friend—is used when you push or pop from the built-in system stack.
\$30	\$s8	Another <i>global</i> register (like the other \$s)
\$31	\$ra	<i>Return address</i> register is used when you do a JAL. The return address is saved in this register.
\$f0-\$f30		<i>Floating point</i> registers We will not use these!

Table 2 MIPS table of registers.

Directive	Meaning	Example
.data	variables coming up!	
.asciiiz	null-terminated string	hello: .asciiiz "Hello, world!\n"
.word	32-bit chunk	myvar: .word 0 #initialized to 0 mywarr: .word 0:100 #100-element array
.byte	8-bit chunk	mybyte: .byte 'a' #initialized to 'a' mybarr: .byte 'a':10 #ten-element array
.space	8-bit space	notme: .space 16 #16-element array
.text	instructions coming up!	

Table 3 MAL directives.

Page No.	Important stuff
102	ASCII table of elements
114	Logical operations (AND, OR, XOR, etc)
220	The MAL instruction set
222	Example MAL code
244	Register usage
268	Directives

Table 4 Useful page numbers from the textbook.

2.3 Development Software

In this part of the course you will use Exceed, a commercial application used to host an X session on one of the CATS servers, and xspimsal, a graphical application running under X that parses MAL and emulates a MIPS machine. This software is available in the following labs: Social Sciences II, Baskin Engineering 213, and Baskin Engineering 105 (xspimsal without Exceed).

2.3.1 SSH, Exceed, and xspimsal

SSH, or secure shell, is the new standard alternative to telnet. It encrypts your password and any data you send to the server.

Exceed is a commercial application used to host an X session. This means that through Exceed, you will be able to run UNIX programs that require Xwindows.

Xspimsal is the simulator which you will be using to run your MAL code. It is a graphical front-end to spimsal that runs under UNIX (but can be accessed off a PC with Exceed).

Here is how to start up and use your environment.

1. Open an SSH session to learn or teach

```
Start -> Internet Tools -> _Unix - Telnet & SSH -> _teach
```

(or whatever server CATS decides to implement today) to allow you to cut & paste a prototype lab into your favorite text editor (pico, vi, emacs). There is a screenshot on the next page.

Type the following line exactly as it appears into your SSH window. You will only need to execute this once—*the first time you ever use xspimsal!*

```
source /afs/cats.ucsc.edu/courses/cmpe12c-rph/newbin/make12cpath
```

This file will write the path of xspimsal to a file it creates called `.cmpe012c` in your home directory, and will also append a line "source ~/.cmpe012c" to the end of your `.login`.

2. Start Exceed.

```
Start -> X-Windows -> Xterm on unix.ic
```

Log in as you normally would, using your CATS username and password. The figure on the following page shows a login screen. This should automatically open an xterm window.

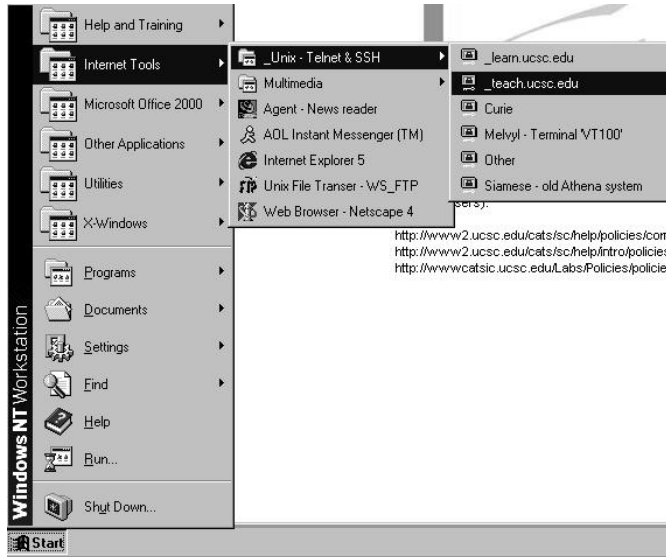


Figure 4 Opening SSH. Use whichever server CATS prefers today.

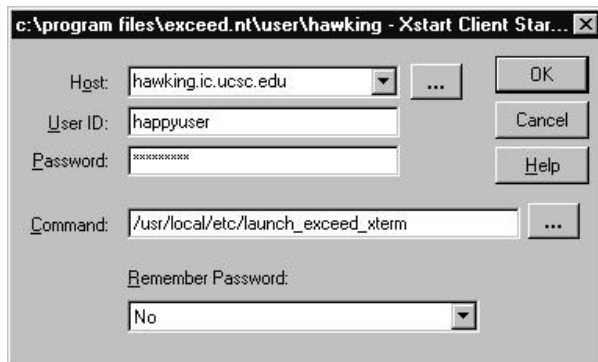


Figure 5 Using Exceed to log in. Note the diagram shows the incorrect host!

3. In the x-term window, start xspimsal.

```
xspimsal -file myfile.mal
```

This will start xspimsal with myfile.mal as the filename of your MAL program in your present working directory (pwd). See the figure on the following page for the open xspimsal window.

4. You will be editing your program directly on your CATS account (on the xterm or SSH window). If you make any changes, be sure to save your work. Then, reload the program by clicking on

```
Clear | Registers & Memory
```

to clear the memory, and Load your program again.

Check the Spim Resources page (off Handouts) for lots of interesting MIPS/Spim information! The address is <http://www.cse.ucsc.edu/classes/cmpe012c/Manual/spim.html>

2.3.2 Xspimsal control buttons, displays, and messages

Xspimsal, a graphical MIPS simulator, has buttons and displays to make your job easier. The next page shows a screenshot of the window that appears when you invoke xspimsal. This table gives an explanation of the buttons. The following paragraphs explain the different display areas.

Button	Use	
<i>Quit</i>	Quits the program.	
<i>Load</i>	Prompts for a filename to load.	
<i>Run</i>	Executes currently loaded program.	
<i>Step</i>	Prompts for number of TAL instructions to step, then executes them. Keep hitting the button to step multiple times. Useful for debugging!	
<i>Clear</i>	<i>Registers</i>	Resets all registers to 0.
	<i>Registers & Memory</i>	Resets all registers and clears the memory of the resident program.

Table 5 Xspimsal control buttons.

Display	What to expect
<i>Register Display</i>	The top part of the window is the table of all registers. Pay attention to the s (global) and t (temporary) registers in the General Registers part of the window.
<i>Text Segments</i> (User and kernel)	This is your code (and kernel code) translated to TAL, including the address of each instruction and its machine code. Remember that most MAL instructions will translate into more than one TAL instruction! Your original instructions are in the comments to the right of the TAL.
<i>Data Segments</i> (Data and stack)	Next is the memory addresses of all of your data segments—this corresponds to the .data portion of your program.
<i>Spim Messages</i>	In this segment are messages from spimsal. <i>Check here for error messages</i> when you first load your program, and when your program does not function properly!

Table 6 Xspimsal display areas.

The screenshot shows the Xspimsal debugger interface. At the top, it displays system information: PC = 00000000, EPC = 00000000, Cause = 0000001c, BadVAddr = 00000000, Status = 00000000, HI = 00000000, and LO = 00000000.

Below this is a section for **General Registers**, listing registers R0 through R31 with their current values. For example, R0 (r0) = 00000000, R1 (at) = 00000000, R2 (v0) = 00000000, R3 (v1) = 00000000, R4 (a0) = 00000000, R5 (a1) = 00000000, R6 (a2) = 00000000, R7 (a3) = 00000000, R8 (t0) = 00000000, R9 (t1) = 00000000, R10 (t2) = 00000000, R11 (t3) = 00000000, R12 (t4) = 00000000, R13 (t5) = 00000000, R14 (t6) = 00000000, R15 (t7) = 00000000, R16 (s0) = 00000000, R17 (s1) = 00000000, R18 (s2) = 00000000, R19 (s3) = 00000000, R20 (s4) = 00000000, R21 (s5) = 00000000, R22 (s6) = 00000000, R23 (s7) = 00000000, R24 (t8) = 00000000, R25 (t9) = 00000000, R26 (k0) = 00000000, R27 (k1) = 00000000, R28 (gp) = 10008000, R29 (sp) = 7ffffeffc, R30 (s8) = 00000000, and R31 (ra) = 00000000.

Next is a section for **Double Floating Point Registers**, showing FP0 through FP30, all with values of 0.0000.

Then is a section for **Single Floating Point Registers**, showing FG0 through FG31, all with values of 0.0000.

Below the registers is a control panel with buttons for: quit, load, run, step, clear, set value, print, breakpoints, help, halt, terminals, and mode.

The **Text Segments** section shows the following code:

```

KERNEL
[0x80000080] 0x3c019000 lui $1, -28672
[0x80000084] 0xac220250 sw $2, 592($1) # sw $v0 s1 # Not re-entrant
[0x80000088] 0x3c019000 lui $1, -28672
[0x8000008c] 0xac240254 sw $4, 596($1) # sw $a0 s2 # Don't need to save
[0x80000090] 0x401a6800 mfc0 $26, $13 # mfc0 $k0 $13 # Cause
[0x80000094] 0x335a00ff andi $26, $26, 255 # and $k0 $k0 0xff# Use just ExcCode
[0x80000098] 0x401b7000 mfc0 $27, $14 # mfc0 $k1 $14 # EPC
[0x8000009c] 0x34020004 ori $2, $0, 4 # li $v0 4 # Print " Exception "

```

The **Data Segments** section shows:

```

DATA
[0x10000000]... [0x10020000] 0x00000000

STACK
[0x7ffffeff8]... [0x80000000] 0x00000000

KERNEL DATA
[0x90000000] 0x20204578 0x63657074 0x696f6e20 0x00206361 - sp sp E x c
[0x90000010] 0x75676874 0x20627920 0x74726170 0x2068616e - u g h t sp

```

At the bottom, the version and copyright information is displayed: SPIM Version 4.4.2, Release: April 22, 1993, Copyright 1990-92 by James R. Larus (larus@cs.wisc.edu), Modified to read SAL code by Scott Kempf (scottk@cs.wisc.edu), See the file COPYING for license information.

Figure 6 Xspimsal screenshot.

2.3.3 Alternatives to xspimsal

Alternatives to xspimsal include spimsal, the non-graphical version of xspimsal. Spimsal is also a UNIX utility, but does not require X or an X emulator such as Xceed. See Spim Resources [4] for detailed instructions on the use of both spimsal and its graphical front-end. Also, click on the `help` button on the xspimsal display, and try the `help` command in the spimsal interface.

Another alternative to xspimsal is Spimsal for Windows. This is linked from the Spim Resources page. We will not be using this in the labs, but you can use this for programming at home. Be careful: Spimsal for Windows may behave differently from Xspimsal in the labs. One of the bugs we have discovered is the *not* instruction gives a *nor* result. Also, students have complained that Spimsal for Windows crashes frequently.

There is another link from the Spim Resources page that goes to a great documentation file on spim/(x)spimsal. Read through it to get a better understanding of the interface and MIPS in general.

2.3.4 Quirks

Sometimes things get weird, even in the nice world of MIPS.

- Registers \$v0 (\$2) and \$a0 (\$4) may get overwritten when performing system calls. See the appendix in this manual for more information.
- In Xspimsal, you will need to clear memory and registers before re-loading a program.
- In Spimsal for Windows, *not* may give a *nor* result.
- More quirks as they are discovered. Make sure to let your lab tutor know if something becomes odd!

2.4 MALgorithms

This section contains some algorithms in MAL that were posted on the Web page and newsgroup in past quarters.

2.4.1 Input and print a number, character by character

The following algorithm shows a C-like pseudocode to get a series of characters and convert them to an integer.

Keep in mind that '0' is ASCII zero. This is what you need to subtract from your number in order to get the actual number, not the ASCII representation. Check the ASCII table in the appendix to verify that ASCII digits are 0x30, or '0', away from their integer counterparts.

We will use the shift-and-add technique for character input. This is similar to saying 682 is 600+80+2. In other words,

Get 6	register: 6
Shift 6 (multiply by 10)	register: 60
Get 8	
Add 8	register: 68
Shift (multiply by 10)	register: 680
Get 2	
Add 2	register: 682

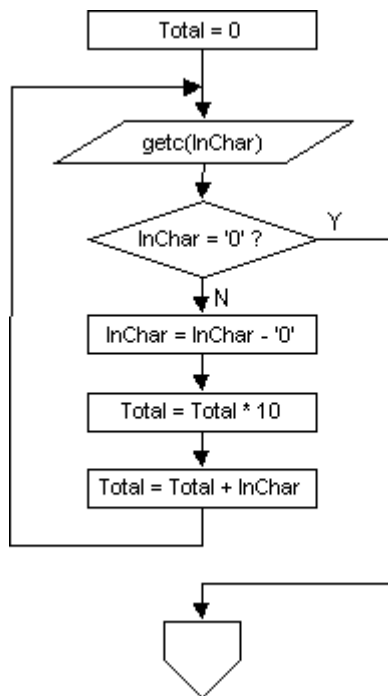


Figure 7 Algorithm: Integer input, character by character.

Character input equivalent pseudocode

```

EndChar = '\n';
Total = 0;

getc( InChar );
while ( InChar != EndChar ) {
    Total = (Total * 10) +
    (InChar - '0');
    getc( InChar );
}
  
```

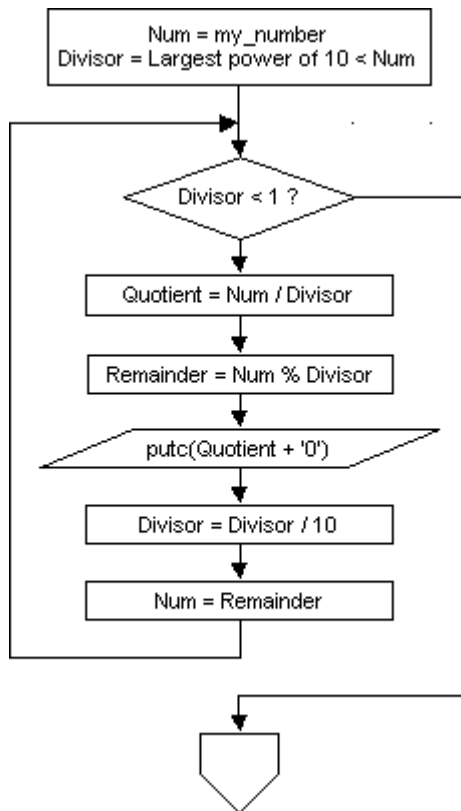
To print a number, use the analogously opposite approach as follows.

We add '0' because we need to change the number into its ASCII representation. If you do not add '0', you will get an unprintable character. (Try this!) Check the ASCII table in the appendix to verify that ASCII digits are 0x30, or '0', away from their integer counterparts.

To find Divisor (the largest power of 10 less than the number), you could implement a while loop, counting up by powers of 10 until the divisor is greater than the number, then backtrack by one step.

We will start from the largest digit of the number and print it. Then, divide the number by the divisor and print the remainder — the next digit. This way, 682 would be

```
(682 / 100) = 6,           print!
(682 % 100) = 82;
(82 / 10) = 8,            print!
(82 % 10) = 2;
(2 / 1) = 2,              print!
(2 % 1) = 0.
```



Character output equivalent pseudocode

```

Num = my_number;
Divisor = (Largest Power of 10 <
Num);

while ( Divisor >= 1 ) {
    Quotient = Num / Divisor;
    Remainder = Num % Divisor;
    putc( Quotient + '0' );
    Divisor = Divisor / 10;
    Num = Remainder;
}
  
```

Figure 8 Algorithm: Printing a number, character by character. Note that pseudocode to find Divisor is not provided.

2.4.2 Extract and print 8 bits

Here we show a C-like pseudocode to print the last 8 bits of a register.

In the bit-printing algorithm shown, we set the mask to be 0x80, or (1000 0000) base 2, because we want to start extracting the bits at the eighth location from the LSB (least significant bit).

The '>>' operator is a logical shift right by one bit.

For example, assume the last 8 bits in my register are 0x61, or 'a'. That's 01100001 in binary. We set the mask to be 1 in the most significant bit (mask is 10000000, or 0x80) and perform the and operation. The result of the and with all the 0 bits is 0; the only bit that will produce an intelligible output is the most significant bit. We print this result (its ASCII equivalent, actually) and shift the mask right by one bit... and repeat! In this way, we traverse the bitstring until we produce all eight bits of output.

Notice we can extend this algorithm to print 32 bits (a whole MIPS register)! To do this, we will need a much bigger mask (the 1 should be at the 32nd position) and a longer loop.

Char	Mask	Tmp (output)
0110 0001	1000 0000	0
	0100 0000	1
	0010 0000	1
	0001 0000	0
	0000 1000	0
	0000 0100	0
	0000 0010	0
	0000 0001	1
	0000 0000	end

Figure 9 Example: Printing the ASCII character 'a', bit by bit.

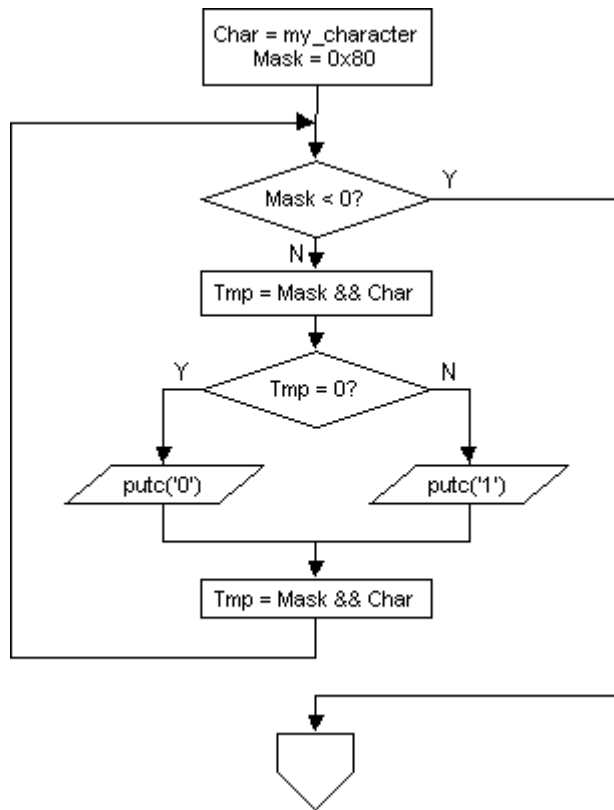


Figure 10 Algorithm: Printing the last 8 bits of a character.

Bit printing equivalent pseudocode

```

Char = my_character;
Mask = 0x80;
while ( Mask > 0 ) {
    Tmp = Mask AND Char;
    If ( Tmp == 0 )
        Then putc('0');
    Else putc('1');
    Mask = Mask >> 1;
}
  
```

2.4.3 What is RPN?

RPN stands for Reverse Polish Notation. It is a method of calculation where the operations are placed after the terms on which to be computed. RPN is used by HP calculators, which are used by geeky professors and students alike! As the calculator reads the input string, it pushes all integers on the stack. As soon as it reaches an operation, it pops the top two integers off the stack and performs the operation on those numbers. Then it pushes the result on the stack and begins reading the next input string. For example,

$$3 + (4 * 5) - 7$$

would be entered into an RPN calculator as

$$4 \ 5 \ * \ 7 \ - \ 3 \ +$$

Example. Trace the stack given the input string

$$22 \ 32 \ 2 \ 4 \ * \ / \ 14 \ + \ -$$

The RPN calculator would read the integers in order, pushing each on the stack. When the calculator reaches the '*', the stack looks like this.

<SP>	
	4
	2
	32
	22

The calculator now pops the top 2 numbers off the stack (2 and 4) and applies the operation (*). It pushes the result (8) on the stack.

<SP>	
	8
	32
	22

The next character that the calculator gets is '/'. Because this is an operation, the calculator pops the top 2 numbers from the stack, operates, then pushes the result back on the stack. On the stack, 32 and 8 are replaced by 4.

<SP>	
	4
	22

The calculator now continues reading and encounters a 14. It pushes it on the stack.

<SP>	

14
4
22

The calculator handles the '+' by popping the two operands (14, 4) and pushing the result (18).

<SP>	
	18
	22

The next item is an operation, '-', so it pops the two operands again and pushes the result (4).

<SP>	
	4

Finally, the calculator encounters a '\n' indicating that the computation is done, so the calculator outputs its answer, which will always be the top element of the stack.

<POP> "Answer: 4"

2.4.4 Tips on procedure calls

Your Friends JAL and JR, or, How to Make Your Program Have Functions

Sometimes you need to get into a section of code, and then bounce back to where you were before. Here's an overview of how to use jal and jr.

```
j   <label>      # jump - jumps directly to address at label
jal <label>      # jump and link - jumps to label AND
                        # automatically stores the return address
                        # in a special register
jr  <register>   # jump return - goes back to address stored
                        # in register specified
```

To the user (you), the address of any instruction is not important. You simply jal to label, then jr to return.

A further dissection of JAL

Jal stores the address of the instruction *after* the jal in a special register, \$31 or \$ra (return address register). The address of the instruction after the JAL is 4 bytes (1 word) away from the address of the jal—the program counter is incremented before you do the actual jump. We will show a simple example using jal and jr.

Words of Warning

Remember that when you jal, the address of the next instruction gets stuck into \$ra (\$31). This means that if you have not yet returned, and you jal again, the previous value will be overwritten! So be careful—jal responsibly. See \$ra getting wiped out below.

```
one:
    jal   three  # stores next address into $31
two:
    # more code
    # more code
three:
    jr   $ra    # go back to whatever is in $31
```

Figure 11 Code fragment showing the use of jal and jr.

```
four:
    jal   six    # stores address of five into $31
five:
    jr   $ra    # return to $31 (which is seven)
six:
    jal   five   # whoops! now the address of seven is in $31
seven:
    # I just lost a bunch of code!!
```

Figure 12 Nested jal wiping out \$ra.

2.4.5 MAL to TAL conversion

This section is a general overview of MAL instructions and their conversions to TAL. The table is meant as a guide, not a concrete reference. That is, you should be able to look at the table a few times, then be able to figure out any similar conversions in the future without the aid of this table.

The table is meant to address instruction equivalencies, not computational issues. It does not include branch and jump calculations.

If any of the TAL instructions are unfamiliar to you, the textbook should explain all of them in one context or another [1].

Classes of MAL to TAL conversions

Simple immediate	add → addi
Load	li → lui, ori
Mostly simple immediate	add → load then add
Kind of odd immediate ²	mul → load then mult, mflo
Logic-twisting branch	ble → slt, beq
Special case	move, j

Branch

beqz \$a, label	beq \$a, \$0, label
bnez \$a, label	bne \$a, \$0, label
b label	beq \$0, \$0, label
blt \$a, \$b, label	slt \$1, \$a, \$b
	bne \$1, \$0, label
ble \$a, \$b, label	slt \$1, \$b, \$a
	beq \$1, \$0, label
bgt, bge are similar	

Load/Store/Move

la \$a, label	lui \$a, label_hi ori \$a, label_lo
li is the exact same as la	
lw \$a, label	lui \$1, label_hi ori \$1, label_lo lw \$a, 0(\$1)
lb, sw, and sb are identical to lw	
move \$a, \$b	add \$a, \$b, \$0

² Remember that mul, div, and rem still need to do the mfhi/mflo stuff even without the immediates!

1. *Register to Register (R-type)*. These are instructions that do not access memory at all. In this kind of instruction, there are three register operands and no immediates. This includes all ALU instructions (add, sub, or, xor, and, neg, and so on).

OP code	Rs	Rt	Rd	Other
6 bits	5 bits	5 bits	5 bits	11 bits

Example: add \$16, \$16, \$17
 OP RD RS RT

2. *Memory and Immediate (I-type)*. This category includes all instructions with memory accesses (loads and stores), as well as instructions that have an immediate. An immediate is a value that is not read from a register, but is needed "immediately" to execute the instruction.

OP code	Rs	Rd	Immediate
6 bits	5 bits	5 bits	16 bits

Example: add \$16, \$16, 25
 OP RD RS IMM

Example: lw \$16, 12(\$17)
 OP RD IMM RS

Example: sw \$16, 12(\$17)
 OP RS IMM RD

* Note which is the source and which is the destination!

3. *Branches (I-type)*. A branch is like a conditional goto statement. It tells your program to go somewhere else and begin executing code there. A branch is relative—that is, it sees where the program is executing (PC+4) and where you want to go (by looking at the label—that's the immediate), takes the difference, and then goes there. There are two understood zeroes at the end of the immediate because each instruction is a multiple of four (100 in binary). Because we don't write the two zeroes, we have a broader branching range. So the next PC is really $(PC+4) - (IMM*4)$. The maximum number of instructions (not bytes) you can branch is $2^{15}-1$ forward and 2^{15} backward in your program.

OP code	Rs	Rt	Relative Offset
6 bits	5 bits	5 bits	16 bits

Example: beq \$16, \$0, hello
 OP RS RT IMM

4. *Jumps (J-type)*. A jump is pseudodirect—it will unconditionally set the PC to the target address. There are 26 bits you can specify in the address field. As in branch, there are two understood zeroes at the end of the instruction (because the address of each instruction is divisible by 4). The top four bits come from the top four bits of the program counter.

OP code	Address
6 bits	26 bits

Example: j hello
 OP ADDR

2.5 Code Examples and Explanations

Here are some code examples you can dissect on your own. Here we present MAL character I/O as well as procedure calls and parameter passing in MAL.

2.5.1 MAL character I/O

An ASCII character walks into a bar. Bartender asks, "What'll you have?" ASCII character says, "Give me a double." Bartender asks, "Having a bad day?" ASCII character says, "Yeah, I have a parity error." Bartender says, "Hmmm. I thought you looked a bit off."

The program `char_io.asm`, is a simple character I/O routine. The program loops, echoing the character entered, until the user presses newline (`'\n'`). Any MAL program starts at the label `__start` (line 19). Notice the use of `branch` as an if statement (line 32).

There are two different ways to print out a character: passing the character's ASCII value (lines 29–31), or simply passing the character (line 36).

The difference between put string (`puts`) and put character (`putc`) is `puts` prints a null-terminated string (defined with `.asciiz`) and takes as an argument a pointer to that string. `putc` will simply print out the character passed to it.

```

1 #####
  # char_io.mal
  # Alexandra Carey 07/11/2000
  # fire@cats.ucsc.edu
5  #
  # This program takes characters as input and prints out the
  # following character in ASCII order.
  # Written for xspimsal.
  #####
10 # The data segment of code is like variable declarations.
   .data
   greeting:      .asciiz "\n\nWelcome to Char IO!\n"
   prompt:        .asciiz "\n\nEnter your characters, newline
15   exits:\n"
   next:          .asciiz "\nThe next char is: "
   bye:           .asciiz "\n\nThank you!\n"

   .text
20  __start:                # the beginning of any program
       puts  greeting
       puts  prompt

   nextchar:
25  # Gets user input, prints +1 (ASCII)
   # Exits on input \n
   # Registers used:
   # s0 -> User input
   # t0 -> '>'
30     li    $t0, '>'      # load immediate
       putc $t0          # print it out
       getc $s0
       beq  $s0, '\n', finish
       puts next
35     add  $s0, $s0, 1
       putc $s0
       putc '\n'          # another way to print an immediate
       b   nextchar

   finish:
40     puts  bye
       done

```

Figure 13 Program: An example of MAL character I/O.

2.5.2 Procedure calls

The program `proc.mal` is an illustration of procedure calls. It is not efficient and does not follow programming etiquette, but is a good example of the different ways to pass parameters.

Argument Registers `$a0` through `$a4` are reserved for passing parameters to functions. I have chosen to pass parameters on the stack for the difference function, and in register `$t2` with the `take_square` function. These are purely a stylistic issues, although generally you should pass parameters with the `$a` registers (or at least be consistent)!

Return Value Registers `$v0`, `$v1` are reserved for passing return values from functions. If you have more than two values to return, pass the rest on the stack.

The *Return Address*, `$ra`, is reserved for the return address from jump-and-link (JAL) instructions. If there is a nested JAL, make sure you save the return address on the stack!

For more information on nested procedures, procedure calls, and `jal/jr`, see [the final section](#).

```

1  #####
   # proc.mal
   # Alexandra Carey 07/11/2000
   # fire@cats.ucsc.edu
5  #
   # A simple program to introduce procedure calls.
   # Input: two characters separated by a comma
   # Output: their ASCII difference
   #####
10
   .data
   greeting: .asciiz      "\n\nWelcome to Proc!\n"
   instruct: .asciiz      "\nEnter two characters separated by a
                           comma.\nOutput will be their difference.\nQ at any time will
                           quit.\n\n"
15  difference: .asciiz     "\nThe difference is: "
   bye:       .asciiz      "\nThank you!\n"

   .text
20  __start:
       puts  greeting
       puts  instruct
       jal   get_input
       jal   find_diff
25  add    $a0, $v0, $0      #move returned val to argument
       jal   output
       b     __start

   get_input:
30  # Expects: nothing
   # Returns: characters 1 and 2 on stack
   # Registers:
   #     t0 -> first input

```

```

#      t1 -> second input
35      putc  '>'
        getc  $t0                #character 1
        beq   $t0, 'Q', finish
        add   $sp, $sp, -4        #push
        sw    $t0, 4($sp)
40      getc  $0                  #comma: discarded
        getc  $t1                #character 2
        beq   $t1, 'Q', finish
        add   $sp, $sp, -4
        sw    $t1, 4($sp)
45      getc  $0                  #newline: discarded
        jr    $ra

find_diff:
# Expects: inputs on stack
50 # Returns: difference in inputs
# Registers:
#      t0 -> second input (first off stack)
#      t1 -> first input (second off stack)
#      t2 -> temporary result
55 #      v0 -> difference (final result)
        lw    $t0, 4($sp)        #restore parameters
        add   $sp, $sp, 4
        lw    $t1, 4($sp)
        add   $sp, $sp, 4
60 # Push return address onto stack
# (anything else that will get clobbered should be saved as well)
        add   $sp, $sp, -4
        sw    $ra, 4($sp)

65      blt   $t1, $t0, negative
        sub   $t2, $t1, $t0      #find difference
continued:
        jal   make_ascii
        add   $v0, $t2, $0
70 # Pop ra and return
        lw    $ra, 4($sp)
        add   $sp, $sp, 4
        jr    $ra

75      negative:
# Dealing with the negative number
        sub   $t2, $t0, $t1
        b     continued

80      make_ascii:
# Informal procedure.

```

```
    # Expects: input in $t2
    # Returns: $t2 plus 0x30
        add    $t2, $t2, 0x30
85         jr    $ra

output:
    # Prints out result
    # Expects: input in $a0
90     # Returns: nothing
    # Registers:
    #     a0 -> input
    #     t1 -> temp for printing
        add    $t1, $a0, $0
95     puts   difference
        putc   $t1
        jr    $ra

finish:
100    puts   bye
        done
```

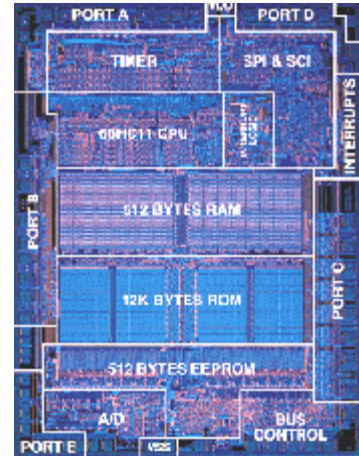
Figure 14 Program: MAL procedure calls with various methods of parameter passing.

3 Motorola HC11

3.1 Introduction to Microcontrollers & HC11

A microcontroller is a microprocessor that is geared towards a specialized purpose, such as controlling everyday appliances like microwaves and wristwatches. A microcontroller typically has memory and I/O on chip. The microkits you will be using contain a Motorola HC11 microcontroller. The HC11 microkits each have 8 input switches, a single interrupt request (IRQ) button, 8 output LEDs, and a two-line liquid crystal display (LCD). The HC11 kits also have a pulse accumulator, or a counter, which is incremented once per clock cycle.

The next few sections introduce the HC11, its hardware and software and explain how to use our development environment. You can find useful algorithms and code examples in in the final sections.



3.2 Beautiful Bounties of HC11

The programs you write will be assembly language (.asm) files, which will later be assembled into s-record (.s19) files. This is the type of file the HC11 knows how to read. The GNU binutils and environment will help you write assembly language files and assemble them into s-record files.

You will use your favorite UNIX editor to create and edit the .asm file, and then use the GNU binutils to assemble them into s-record files.

The pink *Motorola HC11 Reference Manual* (MRM) [7] will be your textbook for this part of the quarter. Although it may be out of the scope of this class, it is a tool you will find quite valuable. But look out! Some things in the MRM are quite different from what we will be using in the labs.

3.2.1 HC11 hardware

Your microkit comes equipped with two buttons, 8 switches, 8 LEDs, and a two-line LCD. Each I/O device is associated with a memory location defined in v2_18 (see Appendix).

Buttons

There are two buttons on the HC11: reset and interrupt (IRQ). The reset button sends a hard-reset signal to the HC11 bootloader. The interrupt button, to the left of the switches, will be useless until you begin to program your own interrupts.

Switches

You will use the eight switches for taking input. You can think of them as an eight-bit unsigned binary number, with the MSB at the far left and the LSB at the far right. The include file, v2_18 [8], maps the switches to a memory location, SWITCHES, to make programming easier.

LEDs

These are eight LEDs used as output devices. They represent an eight-bit binary number with the MSB at the far left and the LSB at the far right. Like the switches, the LEDs also are mapped to a place in memory, LEDS. The lights are active low, that is, writing a zero will turn a light on and a one will turn it off.

LCD

The two-line liquid crystal display can be used for dramatic output! There are 16 characters per line. The include file [8] contains numerous procedures to help you use the LCD. There is documentation on the LCD at <http://www.cse.ucsc.edu/classes/cmpe012c/Manual/lcddisplay/> [9].

3.2.2 And language

Our HC11 programming environment is generally case insensitive, which means ADDA and AdDa will mean the same to the assembler, but note that labels are case sensitive. What was that? Labels are case-sensitive.

Registers

Below is a diagram of the registers you will be able to use. Accumulator D is made up of accumulators A and B: A is the high eight bits, and B is the low eight. This means if you modify A or B, accumulator D will also get modified (and vice-versa)! X and Y are 16-bit index registers, that is, their primary function is to hold addresses (but you can use them any way you want).

You will use the program counter (PC) and condition codes register (CCR) only indirectly, as certain instructions will alter these two registers for you.

The stack

Because you have only three 16-bit registers to play with, you will become friends with the stack very soon! You will use the built-in stack which grows from high to low memory. The stack pointer (SP) is automatically incremented or decremented with each push and pull instruction (how convenient)! For example,

```
PSHA
```

is an inherent instruction (it requires no operands) which will both push accumulator A onto the stack, decrementing the stack pointer. Other stack instructions are pula, pshb, pulb, pshx, pshy, pulx, puly.

Prefixes

Each time you use an immediate in your programming, you will have to use it with its prefix. For example,

```
ADDA  #0x30
ADDB  #30
```

the top instruction adds *hexadecimal* 30 to accumulator A, whereas the bottom adds *decimal* 30 to accumulator B. Integer constants in this assembler work much like they do in C.

The '#' sign prefixing the prefix means "effective address of." In our case, it means the stuff that follows is an immediate, not a memory location.

Valid prefixes are as follows:

Prefix	Meaning	Example
0	octal	040
'	character	'c'
0x	hexadecimal	0x60
none	decimal	42

Table 8 HC11 valid prefixes.

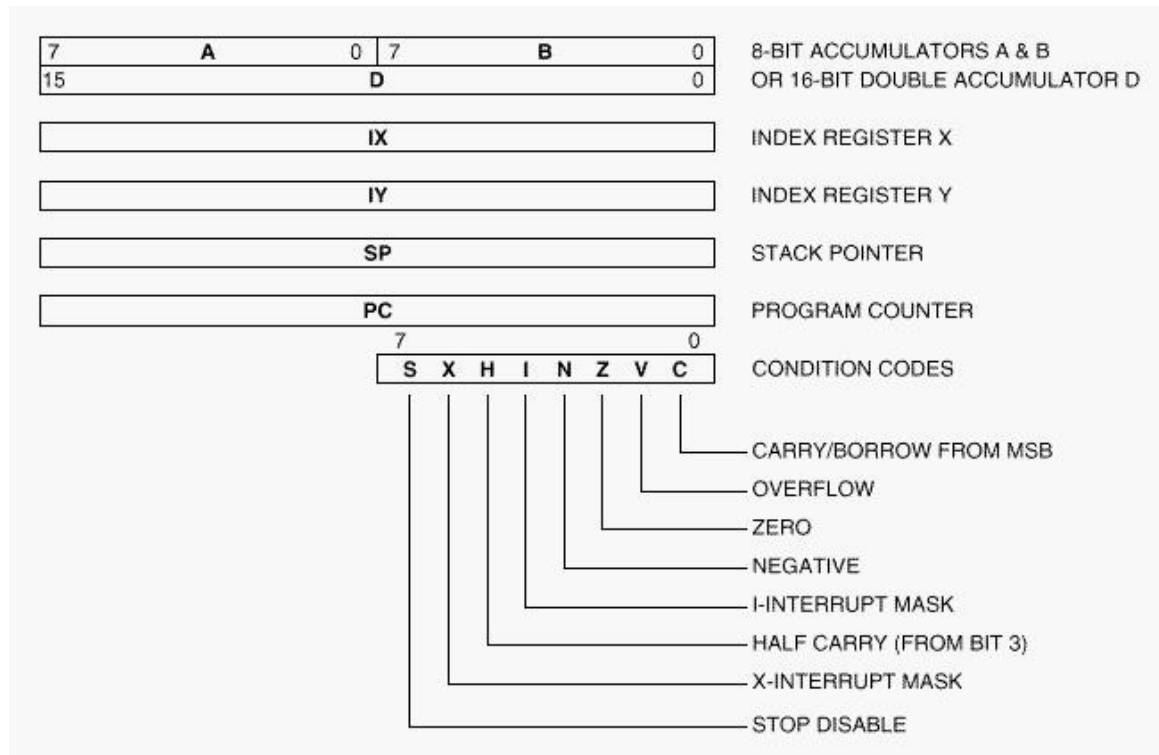


Figure 15 Diagram of HC11 internal registers [7].

Comments

In HC11, we use C-style comment characters: `/*` begins a comment string, and `*/` ends it. One-line comments are supported with `//`.

Instructions and Addressing Modes

There are several types of HC11 instructions and addressing modes. We outline some basic commands and their similarities to MAL.

1. *Inherent*. An inherent instruction is one that requires no operands—that is, the operands are included in the instruction itself. For example,

```
DEX
XGDX
```

decrements the value in register X, and exchanges the contents of register X with accumulator D.

2. *Immediate*. You should be familiar with instructions that require an immediate. For example,

```
CMPA #0b10000010
```

compares the contents of accumulator A to binary 10000010. This instruction sets bits in the condition codes register (CCR).

3. *Index-X*. These types of instructions are useful when indexing into an array, stack, or activation frame. For example,

```
LDD 1, X
```

loads into D a value one byte away from the memory address X points to.

4. *Index-Y*. Similar to Index-X, these instructions are used to index into an array, stack, or frame. For example,

```
STX 0, Y
```

stores the contents of X into the memory location Y points to.

A word about branching

Branching in HC11 is a bit tricky. A branch instruction, such as

```
BEQ    label
```

only contains information about the location, not about the condition of the branch. The branch is conditional on some special bits in the condition codes register (CCR). To set these bits properly, you need to do a comparison. For example, if I want to branch to a label HELLO if accumulator A is less than 0x61 ('a'), I can do this.

```
CMPA  #'a'
BLT   HELLO
```

These two instructions compare accumulator A to 'a' and branch accordingly.

Be sure to read the MRM for more information about comparing and branching.

HC11	MAL	Meaning	Notes
jsr <label>	jal <label>	Jump subroutine, saving return address	Jsr places return address on stack.
rts	jr \$ra	Return to linked location	Rts is inherent, requiring no operands. Rts pulls top 16 bits off stack, jumps to that location.
jmp <label>	j <label>	Unconditional jump	
bra <label>	b <label>	Unconditional branch	In HC11, you must compare before branching.

Table 9 Some HC11 instructions and their MAL "equivalents."

3.2.3 Variables in HC11

Declaring variables in HC11 is similar to MAL. The table below gives an example of a few variable initializations. The `mychar` is a character in memory; `myword` is a word. You can access these variables directly or indirectly. Directly, you can do this.

```
LDAA  mychar
CPX   myword
```

The first instruction loads the contents of `mychar` into accumulator A. The second compares the contents of X to the contents of `myword`.

Note that labels --- in this case, `myword` --- are case-sensitive!

Indirectly, you can use a series of instructions with an offset, like this.

```
LDX   #mychar
STAA  0,X
```

These instructions load the address of `mychar` into X and store the contents of A into the address X points to (`mychar`). For the null-terminated string `mystr`, notice that `mystr` is actually a *pointer* to the string. That is, when referring to `mystr`, you will need to use its address, like this.

```
LDX   #mystr
JSR   OUTSTRING
```

This will output `mystr` to the screen.

Notes!

- Are you used to the old WinIDE/PEMicro environment? We no longer use it! Check some of the differences in the Quirks section.
- Also, keep in mind that all variable declarations must be initialized. Otherwise the assembler will not create space for them.

3.2.4 More information

The Motorola Reference Manual contains the instruction set, which starts on page 6-12 and goes (in table format) for the rest of Section 6. You won't have to use the stuff on 6-17. Get acquainted with Appendix A in the MRM; it provides a detailed explanation of each instruction. Become familiar with the way the MRM displays the information (see B.1 in the Reference Manual).

Declaration Example	Meaning
<code>myshort: .short 8</code>	Declaring short initialized to 8
<code>mychar: .byte 'a'</code>	Declaring an 8-bit variable initialized to 'a'
<code>myword: .word \$ffff</code>	Declaring a 16-bit variable initialized to ffff
<code>mystr: .asciz "Hello!"</code>	Declaring a null-terminated string
	Notice there is no space before or after the comma.

Table 10 Declaring HC11 variables. Note all variables must be initialized.

3.3 *Development Software*

In the second part of the course you will build the HC11 assembly language file with the GNU binutils and then transfer your files to the microkit to execute. *Next*, we examine our development scheme.

3.3.1 **Building and running HC11 executables**

The directions provided in this section are for the PCs in Baskin Engineering 109 computer lab.

From your CATS account:

1. Open a secure-shell connection to CATS.
2. Download source files (such as code templates) you need to your cats account. You can use a web browser or wget to do this:

```
wget http://www.cse.ucsc.edu/classes/cmpe012c/Manual/lab6/echo.asm
```
3. Work on your source files from your cats account. When you are ready to build the executable, run hc11build by typing

```
hc11build echo.asm
```

to generate echo.s19 from echo.asm. The hc11build program will show you the steps it takes to build your files, along with any errors. *Be sure to watch for errors.*
4. Your CATS account is mounted on the X: drive on the desktop. This means that the .s19 file is somewhere inside X: . Find the .s19 file, and keep this folder open.

Now you can download this .s19 file to your HC11 kit

1. Carefully connect your box to power and the serial port of the computer.
2. *Open TeraTerm*: Start | Internet Tools | Athena-UNIX Telnet | Other
Choose Other: COM1 for your connection. *Note*: If you are working from the Social Sciences 2 lab, you may need to check the physical connection into the COM port --- you may need to connect to COM2.
Hit CTRL-X (^X) or press the RESET button on your box.
Type D (and hit enter) to Download at the menu.
3. Download your .s19 file by either dragging the icon from X: (or wherever you put it) to the TeraTerm window, or selecting File | Send File... and selecting your .s19 file. The file transfer will end automatically.
4. Type E (and hit enter) to *Execute* this file. Ctrl-X (^X) will always send a warm-boot the microkit.

Modify your program from your CATS account via an xterm or SSH window, and repeat the steps to assemble and download each new version for testing.

3.3.2 Quirks

There are a few weird things about our programming environment you might have to deal with.

- At first, you may have trouble at first differentiating between a failed build and a successful one. The normal output is a list of commands, a failed build will contain error messages formatted as “filename: error” and the .s19 file will not be produced.
- "Error: Relocation truncated to fit": This error message means your are loading too many bits into a small register. For example, your instruction of `ldaa #myvar` will generate this error if #myvar is larger than the 8 bits that will fit into accumulator A.
- We have several differences between the old environment and our new, improved GNU utilities development environment:
 - We don't use the silly `!`, `%`, `$`, etc. to indicate the base of an immediate. Instead, constants are in "C-style": decimal by default; `0x4e0` is hex; `040` is octal; `0b1011` is binary, and so on.
 - Comments are also c-style; `/*` multi-line comment `*/` or `//` comment instead of using semicolons.
 - All variable declarations must be initialized! Otherwise the assembler will not allocate space for them.
- More quirks as they are discovered. Be sure to mention any problems you have in your lab section!

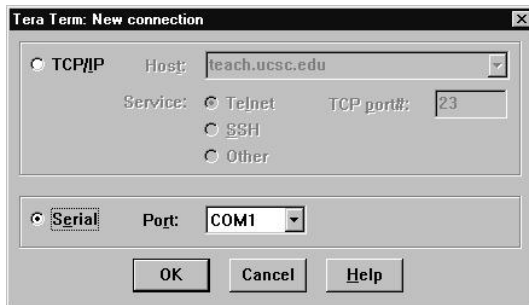


Figure 16 Connecting a microkit to the computer via TeraTerm.

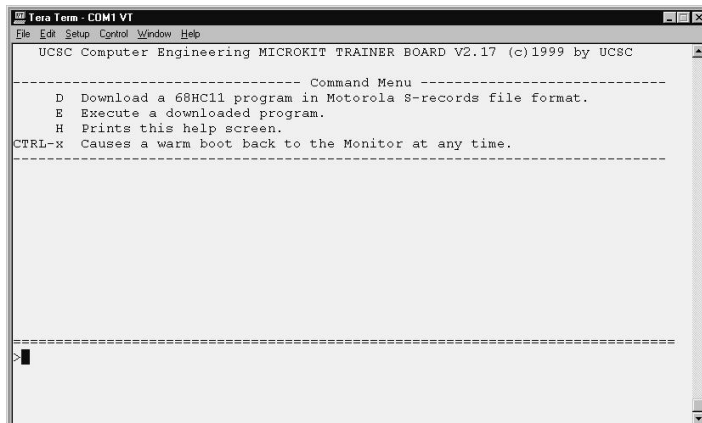


Figure 17 HC11 command menu.

3.3.3 Alternatives to TeraTerm

You can work on your HC11 programs in the comfort of your own home! Baskin Engineering Lab Support (BELS) offers quarterly check-outs of microkits for a fee [10].

The hardware and software requirements for use of the HC11 mikrokit at home are as follows. You must have all of the following:

- a computer with an internet connection
- a COM port
- an SSH client to connect to CATS
- a program to connect to your COM port

For Windows, you can download a free SSH client such as PuTTY [11] from <http://www.putty.nl/> to connect to the CATS server of the week. Next, use HyperTerminal or similar COM software to connect to the HC11 kit via your serial port. Follow the instructions provided to create, modify, build, and transfer your files.

3.4 HC11 Algorithms

This section contains algorithms and general help you may find useful in your HC11 adventures! They are taken from the Web page and previous newsgroup postings.

The following sections explain Rot-13, a simple encoding method, as well as recursion, and an example of recursion using Fibonacci. Next we explore the internals of HC11 procedure calls and talk about interrupts.

3.4.1 Rot-13

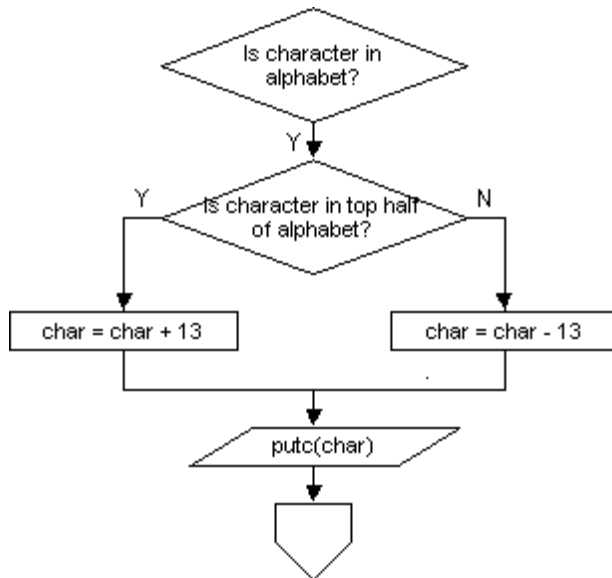
Rotate-13 is a simple encoding method that works like this: If a number lies between A and M in the alphabet, add 13 to it; if it is between N and Z, subtract 13. This gives the impression of "rotating" the character 13 places.

Non-alphabetic characters such as numbers (1,2,3), punctuation (.,?!), and other weird stuff (&#\${}|\, etc) should not be altered before output. That is to say, if I enter '@', I should get '@' back.

Some test strings you may want to try include

```
V yvxr UP11!
jnag gb cebtenz gbtrgure?
```

Rot-13 flow diagram



Rot-13 equivalent pseudocode

```

If ( character is in alphabet )
{
    If ( character is in top half of
        alphabet [A-M|a-m] )
        character = character + 13;
    Else ( character is in bottom half
        [N-Z|n-z] )
        character = character - 13;
}
print (character);
  
```

Figure 18 Algorithm: Rot-13 in pseudo-C.

3.4.2 What is recursion?

In order to solve a problem, a function will need to call itself, which then calls itself again, and so on an indeterminate number of times, until it reaches a simple base case. A function calling itself is said to be *recursive*.

We can think of each function call to a function funcA as an *instance*. The first time funcA is called is called *instance 1*, the second is *instance 2*, and so on. In a typical recursive function, instance1 would call funcA creating instance2, instance2 would call funcA creating instance3, and so on. We will show nested instances in series of recursive calls.

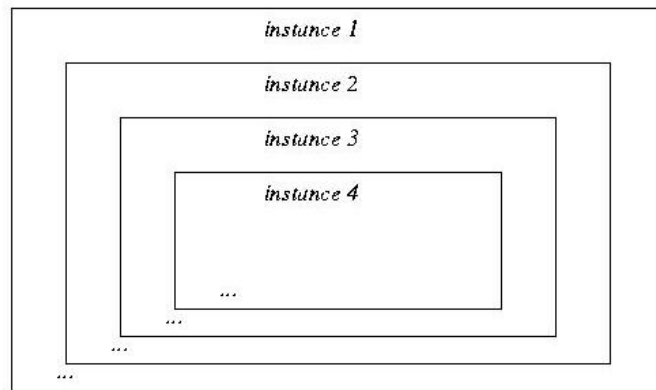


Figure 19 Nested instances in a series of recursive calls.

An example of a problem that can be solved recursively is the calculation of a factorial. A factorial of a natural number n is defined as

$$\begin{aligned}\text{fact}(n) &= n * \text{fact}(n-1) \text{ for } n > 0 \\ \text{fact}(0) &= 1\end{aligned}$$

where factorial of 0 is the base case. We could (and do! see below) code this function recursively.

Next, we show the nested instances and return values associated with call to `fact(5)`.

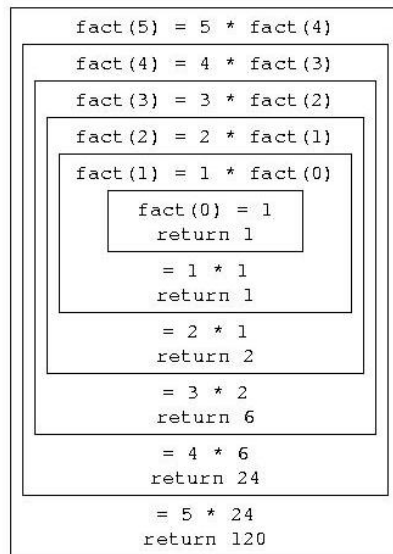


Figure 20 Recursive nested instances of a call to factorial.

Recursive factorial in C

```

int fact(int n) {
    if (n == 0) return 1;
    else
return (n * fact (n - 1));
}

```

Explanation of factorial example

Each successive call of factorial cannot complete until the base case is reached. That is, `fact(5)` cannot produce an output until `fact(4)` is known, and so on.

Note that `fact(0)` is the unique base case. Until we have reached `fact(0)`, none of the solutions are known.

3.4.3 Recursion: Fibonacci in HC11

We will define Fibonacci (like factorial) recursively, as such: each Fibonacci number is the sum of the two previous numbers in the sequence. The base case is $\text{fib}(0)=1$.

For example, the 4th Fibonacci number is 3. Here's how we find that:

$$\begin{aligned}\text{fib}(4) &= \text{fib}(3) + \text{fib}(2) \\ \text{fib}(3) &= \text{fib}(2) + \text{fib}(1) \\ \text{fib}(2) &= \text{fib}(1) + \text{fib}(0)\end{aligned}$$

So,

$$\begin{aligned}\text{fib}(4) &= (\text{fib}(2)+\text{fib}(1)) + (\text{fib}(1)+\text{fib}(0)) \\ &= (\text{fib}(1)+\text{fib}(0) + \text{fib}(1)) + (\text{fib}(1)+\text{fib}(0)) \\ &= ((1+0)+1)+(1+0) \\ &= 3\end{aligned}$$

We will show a Fibonacci function coded in C.

```
fib(n) = fib(n-1) + fib (n-2) for n>1
fib(0) = 0;
fib(1) = 1;
```

Figure 21 Algorithm: Fibonacci defined recursively in pseudo-C.

```
int fib ( int n ) {
  if (n==0) return 0;
  else if (n==1) return 1;
  else return ( fib(n-1) + fib(n-2) );
}
```

Figure 22 Algorithm: Fibonacci coded in C.

3.4.4 HC11 procedure calls

When you jump subroutine (JSR), the return address is saved—just like jump-and-link in MAL. The difference is that in MAL, the return address is saved in \$31 (\$ra), and in HC11 it is stored on the stack. So when you JSR, the return address (16 bits) is pushed onto the stack, and when you RTS, it is pulled.

In other words,

What you do	What's behind the scenes
JSR HELLO	; push address of PC + 1 on stack ; address is 16 bits
HELLO:	
RTS	; pull 16 bits from stack, make this ; pulled value the program counter

3.4.5 Notes on interrupts

Please see Section 5 in the MRM for detailed information of interrupts.

An interrupt is any asynchronous signal to the processor that tells it something important is happening. Some asynchronous interrupts are

- the IRQ button being depressed
- the clock being incremented
- the floppy drive catching fire

Pretend you press the IRQ button, signaling an interrupt. Execution is rushed immediately to the interrupt service routine (ISR)! Here is what happens inside the computer.

- Save all of the important information (stored in registers and accumulators) onto the system stack. (This is happened automatically; you don't have to do this yourself in code.)
- Check against the *interrupt table* to find out where to go. The interrupt table is a vector table in v2_18 that holds information about all sorts of different interrupts that could occur while executing a resident program. (This is also happened automagically.)
- Go to ISR's address stored in vector for IRQ button. The ISR is what you define your interrupt routine to be --- it's code you write.
- Execute all of the code in the ISR. RTI says to return back to the place in the program that was being processed when the interrupt occurred, and pull all that stuff off the stack. See page A-84 in the Motorola Reference Manual for more information on RTI.

Keep in mind that interrupts from the IRQ button are not debounced. That is, one press of the IRQ may spawn several nested interrupts.

3.5 *Code Examples and Explanations*

*Eight bits are in register A
And B does the likewise portray
Though register D
is twice that, you see
Its 16 [bits] can't store an array*

This section contains example code: simple HC11 I/O using echo.asm; procedure calls and parameter passing in HC11; the LEDs and an example light pattern; interrupts and an example interrupt service routine (ISR); and the Pulse Accumulator, which is an internal counter. The final section shows the code for LCDINT, a procedure in v2_18, which prints an integer to the LCD.

3.5.1 **HC11 I/O**

The following program is a simple echo loop. The program takes in a character and prints it out using functions defined in v2_18 (which are available for you to use). This is an include file, like a library, of convenient I/O routines.

The labels on lines 11–12 are variable declarations. You may want to learn about declaring your own variables.

This program uses a number of procedures from v2_18: OUTSTRING (18), GETCHAR (25), OUTCHAR (26), and OUTCRLF (27).

```

1  /* echo.asm
   *
   * Alexandra Carey
   * Winter 2002
5  * Modified from rph's example
   * updated 11/29/01 (talby)
   *
   * A simple program to introduce the 68HC11 development
   * environment.
10 /*
   #include <v2_18.asm>

       .sect .data
   signon: .asciz  "CMPE12C Welcome!"
15 prompt: .asciz  ">"

       .sect .text

   /* Your program starts here indicated by the label "main:".
20  * After startup code in v2_18.asm is done, it jumps to this
   * label and continues running.  (see Motorola Ref. Manual
   * 6-12 for instruction set). */

   main:  ldx      #signon
25         jsr      OUTSTRING      // write a null-terminated
                                       // string to serial port

   /* Get char --> accumulator A, print it.
   * The serial port will output will go to your computer
30  * monitor.  Serial port input will come from the keyboard.
   * There is no local echo; you will not see your own input. */

   echo_loop:
       ldx      #prompt      // address of prompt
35         jsr      OUTSTRING      // write out to serial port
         jsr      GETCHAR      // getchar returns in ACCA
         jsr      OUTCHAR      // write ACCA to screen
         jsr      OUTCRLF      // write '\n' to serial port
         jmp      echo_loop

```

Figure 23 Program: Simple HC11 I/O.

3.5.2 Procedure calls

The program below implements the factorial function non-recursively for eight bits. It starts with the input value and progressively counts down to 0, multiplying along the way. Lines 9–11 are variable declarations. Lines 14–16 demonstrate printing out to the LCD. Lines 18–19 are procedure calls.

In the factorial function, there are a few initializations (24–25), and then the actual factorial starts on line 27. Notice the `dec` instruction (line 27) decrementing the value at memory location `input`. This way it is not necessary to load the value before doing the decrement. Accumulator D gets the result of the `mul` instruction on line 30. That means both A and B are overwritten! We must load the value at `input` again each time through the loop.

The `stop` instruction on the last line tells the processor to freeze in its current state. All output remains on the LCD. The only way to recover from a stopped state is by hitting the RESET button on the HC11 board.

This is a good opportunity to learn more about procedure calls!

```

1  /* factnr.asm
   * Alexandra Carey 07/18/2000
   * updated 11/30/01 (talby)
   * Non-recursive factorial program to illustrate function
5  * calls. */
   #include <v2_18.asm>

       .sect .data
   signon: .asciz "Non-recrsive fact"
10  bye:  .asciz "bye!"
   input: .byte  0

       .sect .text
   main: ldaa  #1           // display on first line
15      ldx   #signon
        jsr   LCDLINE
        ldab  SWITCHES    // input from switches -> b
        jsr   fact
        jsr   finish

20
   /* Expects: value in B
   * Returns: 8-bit factorial of input in D */
   fact: tba
        staa  input       // save it
25  getfact:dec  input
        ldaa  input
        beq   return_to_main
        mul               // D <- a * b
        jmp   getfact

30  return_to_main:
        rts
   finish: ldx  #bye
        jsr   OUTSTRING
        stop

```

Figure 24 Program: Non-recursive factorial in HC11.

3.5.3 Light patterns

We will show an example light pattern which you *cannot use!* It is a binary counter. Keep in mind that the lights are active low, which means that a light is ON when it is zero, and OFF when it is one. The routine stores the pattern on the stack because otherwise it will be overwritten by the loading of the delay for the WAIT, defined in v2_18. Wait takes a value in accumulator D as its delay time.

There is an example lab in this manual that requires the following order of operations.

1. Start first light pattern.
<pushing IRQ sends you to your interrupt routine>
2. Read value from switches
3. Display Fact or Fib—your choice
4. Cycle to next light pattern
5. Wait for next interrupt by displaying light pattern infinitely

```
        ldaa #0           ;initialize pattern
pattern:
        staa LEDS        ;from v2_18
        inca             ;a binary counter
        psha             ;save my pattern
        ldd #0xff        ;a medium-length delay
        jsr WAIT         ;from v2_18
        pula
        jmp pattern     ;reloop
```

Figure 25 Program: Code fragment of a binary-counter light pattern.

3.5.4 Interrupts

Haiku, in MAL:
*Interrupt will send
 next PC (plus four) to the
 Exception PC*

This program appears to loop infinitely (lines 16–19). However, when you press the interrupt button, execution immediately transfers to the interrupt routine (alex_intr). On lines 12–13, I store the address of my interrupt routine to the label associated with the interrupt button (each piece of hardware has an address defined in v2_18) so that the program would know what to do when it encounters this type of interrupt.

The interrupt routine simply displays "Hi" to the monitor and returns. When entering an interrupt, all useful information is stored on the system stack (condition codes register, other registers, program counter). The instruction to return from an interrupt is RTI, which recovers all of the registers off the stack.

See A-84 in the Motorola Reference Manual for more on RTI, and the section in this manual for more on interrupts.

```

1  /* A simple interrupt routine
   *
   * alexandra carey 05/22/2000
   * updated 11/14/01 (talby)
5  */
   #include <v2_18.asm>

       .sect .text
   /* initialize stuff for the interrupt! store the address
10  * of my interrupt procedure to the IRQ (button). */
   main: ldx    #alex_intr // addr of my interrupt routine
         stx    ISR_JUMP15 // initialize the intr button

       // time-wasting loop
15  loop: jmp    loop

       // prints 'Hi' to monitor
   alex_intr: ldd  #0x4869
             jsr  OUTCHAR
20             tba
             jsr  OUTCHAR
             rti    // return from interrupt

```

Figure 26 Program: Example interrupt service routine (ISR).

3.5.5 Pulse accumulator

The pulse accumulator is basically a counter that gets incremented very frequently by the system clock. This program has two interrupt service routines (ISR)—one for the interrupt button (IRQ) and one for the pulse accumulator overflow (PA_ISR).

You could use the pulse accumulator to generate a more infrequent interrupt (PA_ISR), or make it a random seed. When accessed asynchronously (such as from within an interrupt) the pulse accumulator can generate a pseudorandom number.

PA_ISR. The computer executes the code at PA_ISR every time the pulse accumulator overflows. The procedure clears the pulse accumulator overflow bit in an internal status register and returns.

IRQ. The computer executes code at IRQ every time you press the interrupt button. The procedure loads the pulse accumulator counter into accumulator D, loads the number 5 into index register X, and performs an integer divide to get a number between 1 and 5. The remainder is in D, which it stores back to MYRANDOM. After finishing this procedure, MYRANDOM contains a pseudorandom number between 1 and 5.

```

1  /* PACC_1
   * 1 Dec 99/scp
   * This program uses the Pulse Accumulator to produce a
   * periodic time tick.
5  * updated 11/29/01 (talby)
   */
   #include <v2_18.asm>

   #define PACTL   REGBASE+0x26
10  #define PACNT   REGBASE+0x27
   #define TMSK2   REGBASE+0x24
   #define TFLG2   REGBASE+0x25

       .sect .data
15  m1:   .asciz  "Pulse Accumulator Test Program\n\n"
       m2:   .asciz  "PACTL"

       .sect .text
       main:  ldx    #RAMTOP           // setup user stack
20         txs
           ldx    #m1                // pointer to line-1
           jsr    OUTSTRING          // write program name out
           ldaa   #1                  // reset LCD
           jsr    LCD_CMD            // write to command reg in LCD
25         ldaa   #0x0f              // initialize LCD for two lines
           jsr    LCD_CMD
           ldaa   #1                  // specify line-1
           ldx    #m2                // logo-2 string
           jsr    LCDLINE            // write the line to the LCD
30
           ldx    #IRQ               // address of ISR
           stx    ISR_JUMP15         // IRQ hook;
           ldx    #PA_ISR            // address of Pulse-Accumulator
           stx    ISR_JUMP4         // PACNT hook; ibid
35         ldaa   #0xf0              // Turn on pulse accumulator
           staa   PACTL
           ldaa   #0x30              // Clear PAOVF and PAIF

```

```
        staa    TMSK2

40  /* Event spin loop */
    loop:  ldaa    PACNT
        bne     loop
        ldaa    #0x61
        jsr     OUTCHAR
45      jmp     loop

    /* Interrupt Service Routine called each time the Pulse
    * Accumulator overflows or every 32 us. */
PA_ISR: ldaa    TFLG2
50      anda    0xd0          // clear PAOVF
        staa    TFLG2
        rti

    /* ISR called each time the IRQ pin is pulled low
55  * (button pushed). */
IRQ:
    /** Do something with LEDs and switches
        rti
```

Figure 27 Program: Using the pulse accumulator to generate a periodic time tick and pseudorandom numbers

3.5.6 LCDINT

We will explore a procedure that displays a positive integer to the HC11's LCD. It was written by Cliff McIntire and Alexandra Carey in hopes of making HC11 output a bit easier for you. Now, it is a part of the include file, so you can use the procedure freely.

LCDINT displays a 16-bit positive integer to the LCD from accumulator D. It preserves all registers on the stack and uses two memory locations to hold the 16-bit remainder and quotient.

```

_DIVISOR:      .short      0x00
_REMAINDER:   .short      0x00

LCDINT:
//Expects: 16-bit unsigned integer in ACCD
//Returns: nothing
//Notes:  prints ACCD to LCD.  If input is negative, prints 0.
    pshx
    pshy
    psha
    pshb
    std     _REMAINDER
    ldd     #0
    cpd     _REMAINDER
    blt     _NOTZERO
    ldaa   #'0'
    jsr    LCD_CHAR
    bra    _PRINTLOOPEND

_NOTZERO:
//find the greatest divisor
    ldd     #10000
    ldx     #10
_FINDDIV:
    cpd     _REMAINDER
    ble     _ENDFINDDIV
    idiv
    xgdx                    // d->quo  x->rem
    ldx     #10
    bra    _FINDDIV

_ENDFINDDIV:
    std     _DIVISOR
    ldd     _REMAINDER
    ldx     _DIVISOR

_PRINTLOOP:
//decrement divisor, get next remainder & quotient
//print each quotient in turn

```

```
    idiv
    xgdx
    tba
    adda    #'0'
    jsr     LCD_CHAR
    stx     _REMAINDER

    ldd     _DIVISOR
    ldx     #10
    idiv
    cpx     #0                // exit if quo=0
    beq     _PRINTLOOPEND
    stx     _DIVISOR

    ldd     _REMAINDER
    bra     _PRINTLOOP
_PRINTLOOPEND:
    pulb
    pula
    puly
    pulx
    rts
```

Figure 28 Program: Displaying a 16-bit positive integer to the LCD.

4 Example Labs

4.1 *Are Only Examples*

Here are some labs that were posted in previous quarters. Keep in mind that labs change at the discretion of the instructor. The labs in this manual are examples only! The following two sections contain MAL labs and fun HC11 labs. (Oops --- the MAL labs are fun as well!)

4.2 *MAL*

This section contains exciting labs for MIPS assembly language programming! You may encounter some of them in your lab, but expect them to change. You can use these labs in this section as practice labs and references.

4.2.1 *Efficiency*

Posted is a very short C program, `lab1.c`. If you compile a C program with the command "`gcc -S lab1.c`", you will get an output that is a `.s` file, or assembly language.

Posted is a cleaned up version of `lab1.s` called `lab1.mal`, which will run on the MAL simulator, and which is commented so you can see what is going on.

Problem: Your job is to take `lab1.mal` and make it more efficient by changing the MAL instructions within the specified block of code. You may delete, add, change, or move the instructions to make the code run more efficiently (i.e., fewer total instructions executed).

Notes: You *do not* need to change any branch instructions! Do not worry about suboptimal branching.

No collaboration with other students is allowed on this lab assignment!

Evaluation: your lab tutor will assign one of following grades: check +, check, check -, 0.

- For a 0, do nothing.
- For a X-, you will need to submit a MAL program which is a more efficient version of `lab1.mal`.
- For a X, you will need to submit a maximally efficient version of `lab1.mal`. This means that your lab tutor or I will not be able to make your program more efficient.
- For a X+, you will need to submit a program worthy of a check, plus an efficient version of `lab1Xtra.mal` (more complex).

lab1.c

```
main() {
int a = 0;
while(a!=100) { a++; };
}
```

lab1Xtra.c

```
main() {
int a = 0;
int b;
int c;
while(a!=100) {
for (b = a; b <= 100; b ++ ) {
c = b;
}
a++;
}
}
```

lab1.mal:

```
#####
# lab1.mal
# CMPE12C Lab 1
#
# Cliff McIntire

# A section of data.  You can have as many .data segments as you
# would like, intermixed with as many .text segments as you would like.
.data
mem: .word

.text
__start:

### YOU MODIFY FROM HERE...
sw $0, mem                # Initialize memory location to zero

loop:
    lw $s1, mem           # Load value from mem into $s1
    bne $s1, 100, continue # If register is greater than 100,
    continue              # continue
    b escape              # Else exit while loop

continue:
    lw $s1, mem           # Load value from memory into register
    add $s1, $s1, 1       # Increment register
    sw $s1, mem           # Store register back into memory
    b loop                # Start while again

escape:
    done                  # Done

### ... TO HERE
```

lab1Xtra.mal:

```
#####
# lab1Xtra.mal
# CMPE12C Lab 1
#
# Cliff McIntire

.data
aa: .word
bb: .word
cc: .word

.text
```

```
__start:

### YOU MODIFY FROM HERE...
sw $0, aa          # Initialize memory aa to zero

while:
    lw $s1, aa      # Load value from aa into $s1
    bne $s1, 100, while_continue
                    # If $s1 is not equal to 100, continue
    b while_exit    # Else branch to while_exit
while_continue:
    lw $s1, aa
    sw $s1, bb      # Store value from $s1 back into bb
for:
    sw $s1, bb
    ble $s1, 100, for_continue # If ($s1 ≤ 100) continue
    b for_exit      # Else exit for loop
for_continue:
    lw $s1, bb      # Load value from bb into $s1
    sw $s1, cc      # Store value from $s1 into cc

    lw $s1, bb      # Load value from bb into $s1
    add $s1, $s1, 1 # Increment $s1
    sw $s1, bb      # Store value from $s1 into bb
    b for           # Branch to for
for_exit:
    lw $s1, aa      # Load value from aa into $s1
    add $s1, $s1, 1 # Increment $s1
    sw $s1, aa      # Store value from $s1 into aa
    b while         # Branch to 'while'
while_exit:
    done           # Done
### ... TO HERE
```

4.2.2 Displaying the Register (MAL I/O)

Some of the most useful code you will develop this quarter is integer I/O routines. This lab will require you to write loops to input ASCII chars and translate them into a number, and then translate a number into a sequence of chars for printing. You will also use bit operations to print the contents of registers as ones and zeroes. *The code you write in this lab will be reused!*

Submit: malio.mal, README

Files:

- character i/o algorithms
- char_io.mal—an example MAL program

Problem: Write a program that does two things :

- a) Input a series of characters of the following format (variable length input):

```
#####+#####
```

Convert the characters into the appropriate numbers, add them, and then print the result (convert the result BACK into a series of ASCII characters).

- b) Input a character and store it in a register. Then print each bit of the character (total of 8 bits).

Notes/Requirements: Your integer input/output routines should be general—they should work for values up to $2^{32}-1$. You will want to implement this as a loop. You may use any algorithms you wish to do this, but I suggest using the algorithms provided.

Collaboration with other students is allowed on this lab assignment! Make sure you give other students and tutors credit for any help you receive and any code you develop together. Even if you develop some code with someone else and give credit, it still must be your own! This means if you can't write it, you can't use it! Give credit to any help you receive. Rewrite all your code in your own words to guarantee that it really is yours. And give credit. Give credit.

Evaluation: your lab tutor will assign one of following grades: check +, check, check -, 0

- For a 0, do nothing.
- For a X-, you will need to submit a MAL program which inputs numbers and adds them correctly, then outputs them correctly.
- You will get a X- for any of the following:
 - Your program is not general—it works for most inputs but not all.
 - You do not implement the bit printing part of the lab.
 - Bad or no README. See the section on READMEs to see how to make a good README.
- For a X, you will need to submit a fully functional program up to the specifications listed in the program description.
- For a X+, you will need to submit a program worthy of a check, plus you will need to implement the following features:
 - Instead of just handling a '+' for all inputs, also be ready to handle '-', '/', and '*'.
 - Do bitwise printing for both 8-bit characters and 32-bit integers. You can implement the interface for that in any way you wish (i.e., allow the user a choice of which to do OR do one, then the other, etc).

4.2.3 Self-modifying code

*...Memory is something contiguous
So write where you will, don't mistake!
For if you should write where you shouldn't
Your program could certainly break.*

This lab will further familiarize you with coding in MAL. One of the important points of the lab is getting familiar with the idea that memory is a contiguous chunk of storage, and that your program resides in memory while executing.

Submit: self-mod.mal, README

Problem: Write a program which does the following:

- Output a prompt for an operation (And/Or/Xor/Sub/Add).
- Input a character to determine operation.
- Modify a line of code in ArithmeticLoop to do the selected operation.

ArithmeticLoop:

```
Output a prompt for two numbers
Input two numbers (use last week's code...)
(*) And/Or/Xor/Sub/Add numbers (depends on user's choice above)
Print out number (use last week's code...)
Continue? Loop to ArithmeticLoop
```

(*) You will want to write a generic arithmetic operation as a placeholder in the program until you know which operation you will want to perform. For example, you should code 'add \$s1, \$s2, \$s3' on the arithmetic computation line. If the user enters /, -, or *, you can then change that line of code to:

```
'xor $s1, $s2, $s3'
'or $s1, $s2, $s3'
'and $s1, $s2, $s3'
'sub $s1, $s2, $s3'
```

Notice that since you must wait for the user's input to make the change in your code, you must write code that modifies itself—this is what is meant by the term *self-modifying code*. In order to change code in your program, you must do all the same things as you would to change any other part of memory. You must find the address of the location you need to write to, then get the value you want to write, and then write that value to the memory location.

Last Hints:

In machine code add/xor/or/and/nor/sub each is one word of memory.

In machine code,

```
xor $s1, $s2, $s3 = 0x02538826
or $s1, $s2, $s3 = 0x02538825
and $s1, $s2, $s3 = 0x02538824
sub $s1, $s2, $s3 = 0x02538822
```

Warning: NEVER write self-modifying code unless required to in a lab by your supervisor! Self-modifying code can seriously damage your computer!

Notes/Requirements: Your integer I/O routines should be general—they should work for values up to $2^{31}-1$. You can use the routines you developed previously.

Self-modifying code will not actually work on MIPS, but your simulator will allow it. Self-modifying code may work on older architectures (x86).

Collaboration with other students is allowed on this lab assignment! Make sure you give other students and tutors credit for any help you receive and any code you develop together. Even if you develop some code with someone else and give credit, it still must be your own! This means if you can't write it, you can't use it! Give credit to any help you receive. Rewrite all your code in your own words to guarantee that it really is yours. And give credit. Give credit.

Evaluation: your lab tutor will assign one of following grades: check +, check, check -, 0

- For a 0, do nothing.
- For a X-, you must have absolutely everything working except for the code modification.
- For a X, you must have your program working up to the specifications of this write-up.
- For a X+, you must figure out how to implement this program with additional operations that expand to multiple instructions.

```
MUL $s1, $s2, $s3
    0x02530018
    0x00008812
DIV $s1, $s2, $s3
    0x0253001a
    0x00008812
REM $s1, $s2, $s3
    0x0253001a
    0x00008810
```

You may need a NOP (no operation—pretty much, do nothing) instruction. You may use `add $0, $0, $0` as a NOP. The machine code for this is `0x20`.

4.2.4 Stack

*Your stack grows from high to low memory
So when you push words you subtract
The stack pointer's nought but a register
That points to your structure abstract*

This lab will build upon what you have learned in previous labs, as well as familiarize you with the stack.

Submit: `rpn.mal`, `README`

Problem: Write a non-recursive RPN calculator that takes a variable number of integer inputs and character operations.

- Output a prompt.
- Get positive integers and operations until `\n`.
- Push integers onto stack.
- Perform operations as operations are encountered.
- Output result using code from your previous work.

Example:

```

Welcome to my RPN Calculator!
Enter a string of positive integers and operations separated by
spaces!
> 72 23 8 13 + - *
The result is 144
The square root of 144 is 12! (*)

(*) just kidding! You don't have to do square roots.

```

Internals: This is how your stack looks at the end of your input loop:

```

<sp>
    13
    8
    23
    72

```

Here is how to calculate the result:

```
(72 (23 (8 13 +) -) *)
```

If you need a refresher on what an RPN calculator is, see RPN help.

Notes/Requirements:

- All parameters **MUST BE** pushed on the stack until used.
- Your integer input/output routines should be general—they should work for values up to $2^{31}-1$.
- You can use the routines you developed in the MAL I/O lab.

Collaboration with other students is allowed on this lab assignment! Make sure you give other students and tutors credit for any help you receive and any code you develop together. Even if you develop some code with someone else and give credit, it still must be your own! This means if you can't write it, you can't use it! Give credit to any help you receive. Rewrite all your code in your own words to guarantee that it really is yours. And give credit. Give credit.

Evaluation: your lab tutor will assign one of following grades: check +, check, check -, 0

- For a 0, do nothing.
- For a X-, your program must work for single-digit input or only one operation. Bad or no README.
- For a X, your program must work for any legal sequence of positive integers and operations. Output must be correct. Clear, concise README.
- For a X+, implement negative numbers.

4.2.5 Procedure calls

*Your procedures must save all your registers
Your JALs must all save the \$ra
For when you return to your program
"I know where to go!" it should say.*

*The s registers are all global
They should be preserved, given back
But push them and pop them correctly
And count them by words on your stack.*

So far (most of) you have been implementing procedures by branching/jumping to a target and branching/jumping back to a designated return point. This makes routines not very general (you cannot easily call them from different places) and it results in a very tangled and unreadable program (technically called spaghetti).

This lab will require you to implement REAL procedures using JAL and JR which may be safely called from anywhere in your program. Also, you will be re-implementing solutions you have already worked through, so you will be able to see the value of the improved procedural calling method as a result.

Submit: procedures.mal, README

Files:

- Tips on procedure calls

Problem: I have written a program, procedures.mal. It is only half-written. I have done the easy part. In my slothen laziness, I have left all the procedure calls unfinished, so that you can implement them for me.

Your assignment is:

- a) Read through the posted program (procedures.mal) and a sample output (procedures.out) and understand what each part of the program is supposed to do.
- b) Implement all of the unwritten procedures at the bottom of the file. Because the procedures are not yet finished, the program will not run in its current state. You may wish to fill in the functions with stub procedures to enable you to run and debug as you work.
- c) Document ALL your work. This means meaningful comments in the code and a clear, concise README.

Notes/Requirements: You MAY NOT change any of the main program. Your procedures must work as called by my program.

Your integer input/output routines should be general—they should work for values from $-(2^{31})$ to $(2^{31}-1)$. For a check, you must implement your functions to handle negative numbers!

It is not possible to compute a factorial on a negative number. Therefore, your factorial and sum of factorial functions should print error messages when you try to feed them negative values.

Collaboration with other students is allowed on this lab assignment! Make sure you give other students and tutors credit for any help you receive and any code you develop together. Even if you develop some code with someone else and give credit, it still must be your own! This means if you can't write it, you can't use it! Give credit to any help you receive. Rewrite all your code in your own words to guarantee that it really is yours. And give credit. Give credit.

Evaluation: your lab tutor will assign one of following grades: check +, check -, 0

- For a 0, do nothing.
- For a X, you must submit a fully functional program up to the specifications listed in the program description.
- For a X+, you will need to submit a program worthy of a check, plus you will need to implement the functions designated in lab5.mal as X+ material. This will involve uncommenting the marked lines of code in the main program (okay, so you'll have to change main a little, but that's it!).

Failure to implement any of the X procedures or a missing or bad README *will* result in a X- or 0 for the lab! It is advised that you double-check your lab with a tutor before submitting to make sure your program REALLY works!

procedures.out

```

Please Enter a Number
6
The Binary Representation of 6 is 00000110
Please Enter a Number
12
The Factorial of 12 is 479001600
Please Enter 20 Numbers Separated by Spaces
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Your Values in Unsorted Order Are:
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Please Enter a Number
5
The Binary Representation of 5 is 00000101
Please Enter a Number
9
The Factorial of 9 is 362880
Please Enter 20 Numbers Separated by Spaces
1 2 9 8 4 5 8 7 6 5 1 3 5 7 9 10 6 2 5 8
Your Values in Unsorted Order Are:
1 2 9 8 4 5 8 7 6 5 1 3 5 7 9 10 6 2 5 8

Please Enter a Number
--and so on--

```

procedures-x+.out

```

Please Enter a Number
57
The Binary Representation of 57 is 00111001
Please Enter a Number
5
The Factorial of 5 is 120
Please Enter a Number
6

```

```

The Sum of Factorials of 6 is 873
Please Enter 20 Numbers Separated by Spaces
20 19 18 17 16 15 14 13 121 11 10 9 8 7 6 5 4 3 2 1
Your Values in Unsorted Order Are:
20 19 18 17 16 15 14 13 121 11 10 9 8 7 6 5 4 3 2 1
Your Values in Sorted Order Are:
1 2 3 4 5 6 7 8 9 10 11 13 14 15 16 17 18 19 20 121

Please Enter a Number
127
The Binary Representation of 127 is 01111111
Please Enter a Number
8
The Factorial of 8 is 40320
Please Enter a Number
8
The Sum of Factorials of 8 is 46233
Please Enter 20 Numbers Separated by Spaces
1 1 1 9 9 9 2 2 2 8 8 8 3 3 3 7 7 7 4 4
Your Values in Unsorted Order Are:
1 1 1 9 9 9 2 2 2 8 8 8 3 3 3 7 7 7 4 4
Your Values in Sorted Order Are:
1 1 1 2 2 2 3 3 3 4 4 7 7 7 8 8 8 9 9 9

Please Enter a Number
4
The Binary Representation of 4 is 00000100
Please Enter a Number
--and so on--

```

procedures.mal

```

#####
# lab5.mal
# CMPE12C Lab 5
# Cliff McIntire
#
# Procedures:  The procedures called in the main program are not yet
#              implemented.  You need to add code under each matching
#              label at the bottom of this file to implement each
#              procedure.  You MAY add any procedures you wish.  You
#              MAY NOT change any of the driver program
#              (between "START MAIN" and "FINISH MAIN")
# For a X+:  The marked lines commented out in main may be
#            uncommented and the designated procedures may be
#            implemented for a X+

.data
array:      .word  0:20

```

```

NumPrompt:  .asciiz "Please Enter a Number\n"
BinPrompt:  .asciiz "The Binary Representation of "
FactPrompt: .asciiz "The Factorial of "
SumFactPrompt: .asciiz "The Sum of Factorials of "
NumArrPrompt: .asciiz "Please Enter 20 Numbers Separated by Spaces\n"
UnsortedPrompt: .asciiz "Your Values in Unsorted Order Are:\n"
SortedPrompt: .asciiz "Your Values in Sorted Order Are:\n"
is:         .asciiz " is "

.text
__start:
# START MAIN (DO NOT MODIFY THIS SECTION!!!!)

# Integer in, bit string out
# Prompt for an integer, get it
puts NumPrompt
jal  getnum

# Output integer in base-2 format with prompts
# and newline
puts BinPrompt
jal  putnum
puts is
jal  putbin
putc '\n'

# Integer in, factorial out
# Prompt for an integer, get it
puts NumPrompt
jal  getnum

# Compute factorial on integer and output with
# prompts and newlines
puts FactPrompt
jal  putnum
puts is
jal  factorial
jal  putnum
putc '\n'

#X+# Integer in, sum of factorials out
#X+  # Prompt for integer, get it
#X+  puts NumPrompt
#X+  jal  getnum
#X+
#X+  # Calculate sum of factorials and output it with
#X+  # prompts and newlines.
#X+  puts SumFactPrompt

```

```

#X+  jal  putnum
#X+  puts  is
#X+  jal  sumoffactorials
#X+  jal  putnum
#X+  putc  '\n'

# NumArray in, NumArray out
# Prompt for 20 integers, get them, stuff array
puts  NumArrrPrompt
jal  getnumarray

# Output array with prompts and newlines
puts  UnsortedPrompt
jal  putnumarray
putc  '\n'

#X+  # Sort array and output it with prompts and newlines
#X+  puts  SortedPrompt
#X+  jal  sortarray
#X+  jal  putnumarray
#X+  putc  '\n'

putc  '\n'

b    __start

# END MAIN (YOU MAY NOT MODIFY THIS SECTION!!!!)

# START PROCEDURES (YOU MAY (MUST) MODIFY THIS SECTION)
getnum:    # Strip leading spaces from input, then get an
           # ascii number up to a space or newline, then
           # store in $s0.  Must be able to handle negative
           # numbers ( denoted by a number immediately followed
           # by a ! - 3! is really -3 )

putnum:    # Print the value in $s0 as an ascii string.  Must be able
           # to handle negative numbers ( denoted by a number
           # immediately followed by a ! - 3! is really -3)

putbin:    # Print the last 8 bits of $s0

factorial: # Calculate the value of ($s0)! and store in $s0

sumoffactorials: # Calculate the sum of factorials
                 # on 1 to n, where n is the value
                 # in $s0, then store result in $s0
                 # Must call 'factorial'
                 # IMPLEMENT FOR A X+

```

```

getnumarray: # Call getnum 20 times to fill 'array' with values

putnumarray: # Print each value in 'array' in stored (not sorted)
              # order using putnum

sortarray:   # Sort the elements of 'array' in ascending order
              # IMPLEMENT FOR A X+

```

4.2.6 Negabinary arithmetic

Base -2 is one of my favorites! As with any other bases, the weight of position i is b^i , so the positions for an 8-bit negabinary number are:

... -128 64 -32 16 -8 4 -2 1 . $-(1/2)$ $1/4$ $-(1/8)$...

In our binary 32-bit MIPS world, the easiest thing to do on input is to separate the positive and negative parts (using a masking logical operation) and subtract them to get an internal 2s complement representation.

To output a negabinary number, we can use a divide by -2 algorithm, but with adjustments to ensure that the remainder is either 0 or 1 (i.e., never -1):

```

37/ -2 = -18 r 1
-18/ -2 = 9 r 0
9/ -2 = -4 r 1
-4/ -2 = 2 r 0
2/ -2 = -1 r 0
-1/ -2 = 0 r -1 => -1 r 1
1/ -2 = 0 r 1

64 -32 16 -8 4 -2 1
1 1 0 0 1 0 1

37 = 1100101

```

```

7/ -2 = -3 r 1
-3/ -2 = 1 r -1 => 2 r 1
2/ -2 = -1 r 0
-1/ -2 = 0 r -1 => 1 r 1
1/ -2 = 0 r 1

16 -8 4 -2 1
1 1 0 1 1

7 = 11011

```

Division by 2 in signed integers corresponds to right-shift with some corrections, so you can examine the bottom bit and use knowledge of the bottom bit and the iteration number (i.e., even or odd = positive or negative bit position) to determine the next dividend and the next bit.

Problem: Write a program to convert between the following formats with input format specified by user.

```
Formats: (u) unsigned binary
         (s) sign/mag binary
         (1) 1s complement
         (2) 2s complement
         (n) negabinary
         (d) decimal integer
```

Notes:

- Use `getc` and `putc` for all the [nega]binary bases.
- Use code you wrote in your MAL I/O lab for integers.

Example:

```
<helpful help message and greeting>
<Enter number of bits for output>: 8
<Enter input format and number>: n1010
<print out 1010 (base -2) in the other formats>
```

You may want to talk to TAs and classmates about the algorithms needed for this at the C level, or the use of specific instructions in MAL examples unrelated to this problem. Feel free to post negabinary conversions on the newsgroup to see if they're correct. Similarly, feel free to post questions about MAL syntax like how to print a string, and the like.

Submit: `nega.mal`, `README`

README should contain:

- a) Name and e-mail address
- b) Overview of your code
- c) Thoughts on negabinary:
 - What is the range of an n-bit negabinary number?
 - How would you add 2 negabinary numbers?
 - In what circumstances might negabinary be useful?
- d) Any other comments on MAL.

Evaluation: your lab tutor will assign one of following grades: check +, check, check -, 0

- For a 0, do nothing.
- For a X, you must submit a fully functional program up to the specifications listed in the program description.
- For a X+, you will need to submit a program worthy of a check, plus you will need to have great prompts, error messages and the like, inclusion of excess notation with user-entered excess, or something else that's really cool.
- For a X-, everything but the negabinary output should work. Alternately, you program can only handle 8-bit numbers. (X+ items may help move either of these to a check.)

4.2.7 Jump table calculator

Problem: Construct an adding machine with at least the following functions:

1. Clear accumulator
2. Add a number (typed in by user) to the accumulator
3. Subtract a number (typed in by user) from the accumulator
4. Print out the accumulator
5. Quit

This program will use the number printing routine of your previous assignment.

The selection of which function to perform is to be implemented using a jump table (described below).

Purpose:

1. Increase familiarity with addresses as data
2. Illustrate something that is easier in assembly language than C/C++

Submit: jump.mal (your assembly language file), README (including what you think about jump tables and their uses)

Example: Input would look something like the following (program output shown in angle brackets):

```

*****
<Welcome to the Jump Table Calculator!!>
1
2
<Enter number to add:> 10
2
<Enter number to add:> 12
4
<Result is now 22>
3
<Enter number to subtract:> 9
4
<Result is now 13>
5
<Have a nice day!>
*****

```

Implementation:

1. The calculator's accumulator is just a register. Decide which one to use, and only use it for that purpose.
2. For the jump table, create a table in memory like:


```

jt0: .word a0          #clear_func
jt1: .word a1          #add_func
jt2: .word a2          #sub_func
jt3: .word a3          #print_func
jt4: .word a4          #quit_func

```

Here, jt1–jt4 are labels for the four entries in the jump table, and a1–a4 are labels in your code for each of the routines. Memory address jt0 contains the address of function a0, jt1 contains the address of a1, and so on.

To use this jump table you will need to:

1. Load the base address of your table into a register (i.e., the address jt0).

2. Read in a character and convert to a number between zero and four, printing an error message if it is an incorrect character. Your menu should have calculator instructions labeled 0–4.
3. Convert the number between zero and four to an offset in the table. Each MIPS .word is 4 bytes, so you need to multiply the input value (between 0 and 4) by 4 to create an offset between 0 and 16. This can be done with a shift or simply by adding it to itself twice.
4. Add the base address (from 1) to the offset (from 3). Now, if you typed in '2', you should have the address of `jt2` in a register.
5. Load the value in the table into a register. Thus, in the above example, you now have the instruction address `a2` loaded in a register.
6. Perform a jump register (`jr`) on that register. You are now executing the code labeled `a2`.

Your main program will look somewhat like this,

```
while (1) {
    read a character
    check for errors & jump to appropriate routine
endwhile:
}
```

while all of the routines will look like this.

```
myfunc:    do stuff
          goto endwhile
```

In many ways, jump tables in assembly language are much simpler than jump tables in C because addresses are not typed. If you are to perform this in C, you would need to remember the syntax for the address of a function, as in the following.

```
extern int f1(int), f2(int), f3(int);
int (*jtable[])(int) = {&f1, &f2, &f3};
```

Extra: Things you can do towards a +

- a) Computed `goto`. If all of your routines are exactly the same number of lines, you can skip the jump table and add an offset to the base routine. For example, if all routines were exactly 2 instructions, that would be 4 bytes. You could calculate `'a0 + index * 8'` and do a `jr` to get to the appropriate `a-index` routine. Computed `gotos` are available in FORTRAN, but not in C or C++. To receive extra credit under this option, hand in an additional program, `goto.mal`, and discuss what you think of the technique in your README.
- b) Use a 256-element jump table. Initialize a block of memory to a routine that prints an error message (use a loop in MAL), and then go back and fill in the individual codes you want to use. You can make a much better calculator, for example, if '+' indicated addition rather than '2'. With a 256-element table, you do not need to do any error correction or ASCII bias adjustment to the input character (why??). To receive extra credit under this option, implement more operations than those listed above, and discuss what you think of the technique in your README.

Collaboration with other students is allowed on this lab assignment! Make sure you give other students and tutors credit for any help you receive and any code you develop together. Even if you develop some code with someone else and give credit, it still must be your own! This means if you can't write it, you can't use it! Give credit to any help you receive. Rewrite all your code in your own words to guarantee that it really is yours. And give credit. Give credit.

Collaboration with other students is allowed!! Make SURE to give credit!!! Code can NEVER be directly copied, electronically or by other means. You can develop the structure of your routines with other members of your lab, but be sure to rewrite ALL code "in your own words". One way to ensure code is "in your own words" is to rewrite it several hours after your collaborative effort without reference to ANY notes from the collaborative effort.

Evaluation: your lab tutor will assign one of following grades: check +, check, check -, 0

- For a 0, do nothing, or use `puti` or `geti`.
- For a X-, you may write the accumulator without using jump tables, or skip the README
- For a X, you must submit a fully functional program up to the specifications listed in the program description.
- For a X+, you will need to submit a program worthy of a check, plus you will need to implement the computed goto or 256-element jump table.

4.2.8 Variable-sized stack operations

In this lab you will create a stack that pushes variable-sized inputs.

Submit: `rpn.mal`, README

Files: What is RPN?

Problem: Create an stack that can push two different data types:

Integer ranging from -2^{31} to $2^{31}-1$. This will be a 32-bit data type. Reuse your integer input/output code.

String of variable length from 1 to n characters³. This means that the size of this data type can vary from 1 byte (8 bits) to n bytes ($8n$ bits).

Here's a problem—how do you know what kind of data you're dealing with, if there are different data types? Each input that goes onto the stack will have a tag (unsigned, size = 1 byte) associated with it.

Tag = 0 means 4-byte integer.

Tag = n (where $n \neq 0$) means string of n chars.

Notes/Requirements: The functions of this variable-sized stack should be the following⁴.

- (R) Read in integers (including negatives) and strings of variable length: Separate the inputs by a space or a newline character. For integers, handle negatives in the same way as last week—4! is -4. For strings, you may restrict your string input to lowercase letters only.
 - Push input: Each input that you read in needs to be pushed onto the stack. You will need to keep track of what kind of input it is (string or integer) with the tag.
 - Pop input: You will need to be able to pop the input from the stack in order to perform operations on it.
- (V) Convert string to integer length: Pop string, find its length, and push the integer result back on.

³Notice that you can only store a string of 256 characters because the tag is only 8 bits. You will not need to do any error checking for this, but keep it in mind and explain it in your README if your tutor gives you trouble.

⁴Wouldn't this be easy (and fast!!) if you were to use your jump table from Lab 3?

- (P) Print out entire stack: Print stack, but do not alter it. Indicate bottom of stack, current stack pointer, the size in bytes of the stack, and the number of elements in the stack (you can just use a counter that gets incremented with each push and decremented with each pop). Print out negative integers preceded by a dash (-).⁵
- (C) Clear stack: Pop all inputs and reset the stack pointer.
- Integer operations: Pop top two elements off the stack, perform the specified operation on them, and push the result.
 - + , - , * , / , the basics.
 - (N) Negate integer: Negate the integer at the top of the stack.

Would you like to do more calculations? Then here is the RPN calculator you can implement for X+.

Basic stack operations:

- (D) Duplicate top of stack: Pop the top of the stack and push the result twice.
- (S) Swap top of stack: Pop the top two elements on the stack, reverse their order, and push them again.

For strings: operator overloading! Doing these will give you a note on your narrative evaluation about how crazy you are for doing these operations. Maybe you'll get a candy bar, too.

- string + string = string. This will concatenate the strings and place them together. So, "thislab" + "isfun" would yeild "thislabisfun". Re-push your result back onto the stack.
- string + int = string. This will shift each character of the string int (mod 26) number of times. For example, "abcd" + 13 will do a ROT-13 and will yeild "mnop".

Some words of warning

- Push and pop bytes only or else bad stuff will happen⁶. Remember that the smallest element you can have on your stack is the tag, which is only 1 byte, and everything else is in multiples of bytes. That means you will use lbu, sb (load byte unsigned and store byte—there is no unsigned for store byte—to prevent sign-extending).
- Decide in which order you will be pushing your elements! Which byte goes on first? Which last? What does a reversed integer look like?
- Decide whether you want to push the tag before or after the input. Only one of these makes sense. (Why?)⁷

Collaboration with other students is allowed on this lab assignment! Make sure you give other students and tutors credit for any help you receive and any code you develop together. Even if you develop some code with someone else and give credit, it still must be your own! This means if you can't write it, you can't use it! Give credit to any help you receive. Rewrite all your code in your own words to guarantee that it really is yours. And give credit. Give credit.

⁵ For printing out your entire stack: Are you going to use the system stack or a stack you create (mystack: .word 0:xxx)? It's up to you, but remember that you will need to keep track of the top and the bottom of your stack. You can do this by storing the address of the bottom (before you push anything on) into the register of your choice, including \$fp, the frame pointer. That's its purpose.

⁶ Bad stuff like a Memory Unaligned Error

⁷ Do you push the tag first or last? Think about how you will pop stuff off the stack. How do you know how much to pop (variable length strings)?

Evaluation: your lab tutor will assign one of following grades: check +, check, check -, 0

- For a 0, do nothing, or use geti or puti.
- For a X, you must submit a fully functional program up to the specifications listed in the program description.
- For a X+, you will need to submit a program worthy of a check, plus you will need to implement the integer operation listed in the program description.

For a X+ and a candy bar, you will need to prove how crazy you are by submitting a program worthy of a check, and of a check plus, and you will need to do the string operations, too.

4.3 HC11

This section has some fun example labs for the HC11! You may encounter some of them in your lab, but expect them to change. You can use these labs in this section as practice labs and references.

4.3.1 I/O: ROT-13

The HC11 has three (3) 16-bit registers you can use: D, X, and Y. Accumulator D is made up of two 8-bit accumulators, A (the top half) and B (the bottom half). You should not need more than an 8-bit register for this lab.

Submit: rot13.asm, README

Files:

- introduction to HC11
- starting IDE—how to use the programming environment
- rot-13 help
- echo.asm, an example HC11 program

Problem: I have given you a program, echo.asm, which reads in a character and prints it out onto the monitor screen. Unfortunately, I am typing from a book of rot-13 jokes, so I can't make sense of the output. Your assignment is:

- a) Read through the program (echo.asm) to understand what it is doing.
- b) Download the program to the HC11 training board. Make sure you understand what it's doing!
- c) Modify the code to create a rot-13 translator.
- d) Document ALL your work. This means meaningful comments in the code and a clear, concise README.

Notes:

- Start by compiling and running echo.asm, then add the compare and branch for rot-13.
- In your README, include your favorite HC11 instruction, as well as 5 others you find interesting!
- The output will not be to the LCD, but to your computer's monitor. To use LCD routines, check the v2_18 file for appropriate functions.
- Read about rot-13!

Collaboration with other students is allowed on this lab assignment! Make sure you give other students and tutors credit for any help you receive and any code you develop together. Even if you develop some code with someone else and give credit, it still must be your own! This means if you can't write it, you can't use it! Give credit to any help you

receive. Rewrite all your code in your own words to guarantee that it really is yours. And give credit. Give credit.

Evaluation: your lab tutor will assign one of following grades: check +, check, check -, 0

- For a 0, do nothing.
- For a X, you must submit a fully functional program up to the specifications listed in the program description.
- For a X+, you will need to submit a program worthy of a check, plus include a toggle feature. When the user enters '#' as input, your rot-13 program will switch to 'normal' mode (that is, the output will be the same as the input). Entering '#' again will switch back to 'rot-13' mode.

A missing or bad README *will* result in a X- or 0 for the lab! It is advised that you double-check your lab with a tutor before submitting to make sure your program REALLY works!

Example X+ output:

```
Welcome to my rot-13 program!
>U -> H
>p -> c
>l -> l
>l -> l
>` -> `
># -> Normal
>u -> u
>p -> p
>l -> l
>l -> l
># -> Rot-13
>p -> c
>b -> o
>b -> o
>y -> l
...
```

4.3.2 Procedure calls

This lab will familiarize you with the mechanisms on the HC11 for performing function calls, and will require you to develop good function calling methodology.

Submit: factorial-recursion-for-check_Fibonacci-for-checkplus_lab-7.asm (*), README

(*) Just kidding! Call your program fact.asm

Problem: You must *recursively* implement a factorial function on the HC11. For a description of what is recursive and what is not, see *What is recursion?* in this manual. Given is an incomplete program, fact.asm. The main portion of the program and an integer printing routine have been implemented, but I have left the implementation of the factorial function to you.

- a) Read through the posted program (fact.asm) to understand what it is doing.
- b) Implement the *recursive* factorial function.
- c) Download your program to the HC11 kit and test your work.
- d) Document ALL your work. This means meaningful comments in the code and a clear, concise README.

Notes:

- The result of factorial must be capable of 16-bit output. This means that your function must go up to $\text{fact}(6) = 720$.
- Fibonacci, if you choose to implement it for a X+, needs to also be recursive and capable of 16-bit output.

Collaboration with other students is allowed on this lab assignment! Make sure you give other students and tutors credit for any help you receive and any code you develop together. Even if you develop some code with someone else and give credit, it still must be your own! This means if you can't write it, you can't use it! Give credit to any help you receive. Rewrite all your code in your own words to guarantee that it really is yours. And give credit. Give credit.

Evaluation: your lab tutor will assign one of following grades: check +, check, check -, 0

- For a 0, do nothing.
- For a X, you must submit a fully functional program up to the specifications listed in the program description.
- For a X+, you will need to submit a program worthy of a check, plus you will need to uncomment the Fibonacci section and implement the Fibonacci function. See Fibonacci help for a description of Fibonacci.

A missing or bad README *will* result in a X- or 0 for the lab! It is advised that you double-check your lab with a tutor before submitting to make sure your program REALLY works!

fact.asm:

```
/*
 * LAB 7
 * factorial and fibonacci, with LCDINT
 * updated 11/29/01 (talby) */

#include <v2_18.asm>
```

```

        .sect .data
// START DO NOT MODIFY *****
numprompt:
        .asciz "ENTER when ready"
factprompt:
        .asciz "Fact("
fibprompt:
        .asciz "Fib("
spaces: .asciz "          "

        .sect .text
main:   ldx    #spaces          // a bunch of initializations
        ldaa   #1
        jsr    LCDLINE
        ldx    #spaces
        ldaa   #2
        jsr    LCDLINE
mainloop:
        jsr    getnum          // returns B
        jsr    clearend
        jsr    putfact         // expects B
        jsr    clearend
// UNCOMMENT (and implement) FOLLOWING CODE FOR A X+
//      jsr    getnum
//      jsr    clearend
//      jsr    putfib
//      jsr    clearend
        bra    mainloop

getnum:
//Expects: nothing
//Returns: SWITCHES in ACCB
//Notes: prompts, waits for a keypress. clobbers A,X
        ldx    #numprompt
        ldaa   #1
        jsr    LCDLINE        // display a null-terminated string
        jsr    GETCHAR        // faking an interrupt
        clra
        ldab   SWITCHES
        rts

putfact:
//Expects: user input in ACCB
//Returns: nothing
//Notes: prints ACCB (user input), then calculates & prints fact(B)
        ldx    #factprompt
        ldaa   #2
        jsr    LCDLINE

```

```

        clra
        jsr    LCDINT          // expects :-D
        ldaa  #' )'
        jsr    LCD_CHAR
        ldaa  #' ='
        jsr    LCD_CHAR
        tba

        jsr    factorial      // expects: B
        jsr    LCDINT        // expects: D
        rts

putfib:
//Expects: user input in ACCB
//Returns: nothing
//Notes: prints ACCB (user input), then calculates & prints fib(B)
        ldx    #fibprompt
        ldaa  #2
        jsr    LCDLINE
        clra
        jsr    LCDINT
        ldaa  #' )'
        jsr    LCD_CHAR
        ldaa  #' ='
        jsr    LCD_CHAR
        tba
        jsr    fibonacci
        jsr    LCDINT
        rts

clearend:
//Expects: nothing
//Returns: nothing
//Notes: prints spaces to clear the end of a previously written
line
        ldaa  #' '
        jsr    LCD_CHAR
        jsr    LCD_CHAR
        jsr    LCD_CHAR
        jsr    LCD_CHAR
        rts

// END DO NOT MODIFY *****

factorial:
//Expects: unsigned integer in ACCA
//Returns: fact(A) in ACCD
//Notes: If ACCA contains a negative number, return 0.
        rts

```

```
fibonacci:                // For X+
//Expects: unsigned integer in ACCA
//Returns: fib(A) in ACCD
//Notes: If ACCA contains a negative number, return 0.
                Rts
```

4.3.3 Interrupts

This lab will accustom you to the idea of interrupts, and will teach you how to program your own interrupt routines!! You will be writing to the LEDs and the LCD, and reading from the SWITCHES.

Submit: intr.asm, README

Files:

- lights.asm—an example light pattern.
- intrex.asm—an example interrupt service routine.

Problem: This is a multi-part lab. Make sure that your program works each step along the way! You will be writing this in HC11.

- I) *Light patterns*—Create three (3) interesting light patterns to display onto the LEDs. You will need to have these in a loop. The example shows a simple pattern that is not very interesting (which you can't use!). You will need to store these patterns to memory.
- II) *Factorial*—Make sure your recursive factorial function from the previous lab works for 16-bit values!
- III) *Interrupts*—This is where you will put parts one and two together. Every time you hit the interrupt button, two things should happen:
 - a) You switch light patterns (if you were on pattern 1, you would go to pattern 2, etc), reading them and writing them to memory.
 - b) You display the factorial of the value on the switches.
- IV) *Documentation*—Document ALL your work. This means meaningful comments in the code and a clear, concise README.

Notes:

- You may use your Fibonacci procedure instead of or in conjunction with the Factorial procedure if you want.

Collaboration with other students is allowed on this lab assignment! Make sure you give other students and tutors credit for any help you receive and any code you develop together. Even if you develop some code with someone else and give credit, it still must be your own! This means if you can't write it, you can't use it! Give credit to any help you receive. Rewrite all your code in your own words to guarantee that it really is yours. And give credit. Give credit.

Evaluation: your lab tutor will assign one of following grades: check +, check, check -, 0

- For a 0, do nothing.
- For a X, you must submit a fully functional program up to the specifications listed in the program description.
- For a X+, you will need to submit a program worthy of a check. There will be a competition in each lab where each of your light patterns will be judged based on creativity, complexity, and originality. If you win any of these categories, you will get a X+!!

A missing or bad README *will* result in a X- or 0 for the lab! It is advised that you double-check your lab with a tutor before submitting to make sure your program REALLY works!

4.3.4 Hi-Low Game

Stump your tutor! Program a neat game for the HC11!

Submit: game.asm, README

Files:

- Help on the pulse accumulator for eventually generating random numbers.
- pacc_ex.asm, pulse accumulator routine.

Purpose: Do something fun with your HC11 kit using LEDs, the LCD, and the switches.

Problem: Program an 8-bit hi-low game in your kit.

- a) Print a nice prompt to the LCD and display your favorite pattern on the LEDs.
- b) Generate a number internally.
 - Basic:* Have an initial starting value that you rotate each new game.
 - For some credit toward a X+,* use a more sophisticated random number generator.
- c) Prompt the user to input a number on the switches. As the user toggles the switches, the LEDs should represent that number. When user presses the interrupt button, the program proceeds...
- d) Display on the LCD the guess and whether it was too large or too small, compared to your original number.
- e) Continue looping until the guess is correct!
- f) Do something appropriately exciting on the LCD and the LEDs when the user
- g) guesses correctly for either a specific amount of time (several seconds) or until the user presses the interrupt button to start a new game.

Example: Flash "YOU WON!!" on the LCD for 64 iterations, or simply display "Good Job!" and go on with the program when the user hits the button.

Example I/O:

Computer's secret 8-bit number is 0001 0100 (20).

```
I enter 16 on the switches: 0001 0000.
Computer says: Too low 16
I enter 32 on the switches: 0010 0000.
Computer says: Too high 32
I enter 24 on the switches: 0001 1000.
Computer says: Too high 24
I enter 20 on the switches: 0001 0100.
Computer says: You win!! 20 and flashes the lights.
```

Then, I hit the interrupt button and Computer has a new number for me!

No collaboration is allowed on this lab assignment.

By now, you should be quite familiar with the HC11 programming.

Exception: you may discuss tricks for using the LCD with your classmates.

Evaluation: your lab tutor will assign one of following grades: check +, check, check -, 0

- For a 0, do nothing.
- For a X-, complete the program to specifications, but not working well or not very interesting.
- For a X, you must submit a fully functional program up to the specifications listed in the program description, with a great README and LCD instructions.
- For a X+, you will need to submit a program worthy of a check, plus a more interesting game.

A bad or missing README *will* result in a X- for the lab! It is advised that you double-check your lab with a tutor before submitting to make sure your program REALLY works!

5 Appendix: Summaries and Classwork Help

This appendix contains summaries of logical operations and binary number conversion.

5.1 Logical Operations

Haiku:

*A strange instruction,
XOR gives odd parity:
zero, one, zero*

You will need to use logical operations to "mask" bits in a register—to get only certain specific bits out of a whole string of them. This section shows a summary of logical operations you may find useful.

The main operations you will use are *and*, *or*, *xor*, *nand*, and *nor*. *And* is used to extract certain bits to check their value. *Or* is used to fill a register with a certain bit pattern, or to extract a whole series of bits. *Xor* can be used with all ones to flip the bits, changing ones to zeros and zeros to ones. *Nor* and *nand* are simply opposites of *or* and *and*.

<i>and</i>	True when <i>A and B</i> are true simultaneously.
<i>or</i>	True when one of <i>A or B</i> is true, or when both are true.
<i>xor</i>	Exclusive <i>or</i> . True when exactly one of <i>A or B</i> are true.
<i>nand</i>	Negated <i>and</i> —true when result of <i>and</i> is false.
<i>nor</i>	Negated <i>or</i> —true when result of <i>or</i> is false.

To see how this works with actual bits, see the following table.

Inputs		Outputs				
A	B	<i>A and B</i>	<i>A or B</i>	<i>A xor B</i>	<i>A nand B</i>	<i>A nor B</i>
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	1	0
1	1	1	1	0	0	0

Table 11 Summary of logical operations. *A* and *B* are input values; the last five columns show outputs of logical operations.

5.2 Binary Number Conversion Summary

Haiku:
*2's comp negative:
 XOR input with all ones,
 add one to LSB.*

You will study various ways of representing binary numbers (and other number bases) that we review in this section. You will be expected to know all of the methods of binary representation, but keep in mind that a computer usually uses two's complement!

5.2.1 Binary

In this section we will look at some forms of binary representation.

Unsigned binary

You cannot represent negative numbers in unsigned binary representation. An unsigned number is always positive! This representation has no sign bit, giving more room to store a larger number. For example, in 8-bit unsigned binary,

$$0110\ 1101_2 = 109_{10}$$

$$1101\ 0111_2 = 215_{10}$$

Signed Magnitude

MSB (most significant bit) is sign bit. If MSB is 0, the number is positive; if it is 1, the number is negative. The remaining bits are the number in unsigned representation, called the *magnitude*. Notice that you lose one bit of information against unsigned, but you can now represent negative numbers! For example, in 8-bit signed magnitude,

$$0110\ 1101_2 = 109_{10}$$

$$1101\ 0111_2 = -87_{10}$$

One's Complement

MSB is the sign bit. If the number is positive (MSB=0), it is the same as unsigned. If number is negative (MSB=1), find its complement by flipping all the bits. Now, you have the number's magnitude as an unsigned number. For example, in 8-bit one's complement,

$$0110\ 1101_2 = 109_{10}$$

$$1101\ 0111_2 = -40_{10}$$

Two's Complement

MSB is the sign bit. If the number is positive (MSB=0), it is the same as unsigned. If number is negative (MSB=1), add one (1) to the one's complement representation (this is called finding the additive inverse). This will give you the number's magnitude as an unsigned number. For example, in 8-bit two's complement,

$$0110\ 1101_2 = 109_{10}$$

$$1101\ 0111_2 = -41_{10}$$

Bias/Excess

Bias representation is useful when you want to store a number in a certain range. Add the bias to the number to get into the bias representation. Subtract the bias from the number to get out of the bias representation (or add the bias' additive inverse!). For example, in 8-bit bias-127,

$$0110\ 1101_2 = -18_{10}$$

$$1101\ 0111_2 = 88_{10}$$

Radix Point

When confronted with a number that has a radix point⁸ (a decimal point somewhere in the number), separate the number into two problems. First, convert whole number to binary; next convert the radix point into binary.

5.2.2 Example converting decimal to radix point

Let's convert 45.25 to binary. This is two problems: the whole number, and the radix part. We get the whole part by the usual division by 2⁹ algorithm: divide 45 by 2. You get a quotient and a remainder. The remainder (1) is the first (least significant) bit of the answer. Divide the quotient by 2 and take the remainder. Repeat until the quotient is 0.

$$\begin{array}{r} 45 / 2 = 22 \quad r 1 \\ 22 / 2 = 11 \quad r 0 \\ 11 / 2 = 5 \quad r 1 \\ 5 / 2 = 2 \quad r 1 \\ 2 / 2 = 1 \quad r 0 \\ 1 / 2 = 0. \quad r 1; \text{ done} \end{array}$$

Reading this backwards, we get $45 = 101101$.

For the radix part, we multiply the radix by the base we're converting to: in our case, base 2. Continue multiplying using only the *fractional part* of the answer. The whole part is the radix part of the answer. We know we're done with this when one of three things happens:

1. The number terminates (i.e., the result of the multiplication is zero).
2. The number repeats (you'll know if this happens).
3. Your hand falls off (i.e., either you meet the expectations of the number of bits necessary for the answer, you get bored, or you realize you have made a mistake).

$$\begin{array}{l} 0.25 * 2 = 0.50 \\ 0.50 * 2 = 1.0 \text{ (done)} \\ 0.00 * 2 = 0.0 \text{ (still done)} \end{array}$$

Reading this forwards, we get 0.01.

The final answer: $45.25 = 101101.01$

⁸ "Radix point" is the politically correct term for decimal point. It's non number-base discriminant.

⁹ Division by 2, or whichever base you are converting into. For base 3, divide by 3. Everything else remains the same.

5.2.3 Octal

($2^3 = 8$) Octal, or base-8, is a way to represent binary by grouping base-2 numbers into threes.

To get an octal number from binary, start at LSB (least significant bit) or at the decimal point and *sign-extend* if needed. Sign-extending means to duplicate the MSB (most significant bit) for as many places as necessary. In octal, digits range from 0 through 7. For example,

$$0110\ 1101_2 = 255_8$$

$$1101\ 0111_2 = 727_8$$

we sign-extend the one in the second example to get $111\ 010\ 111_2 = 727_8$.

5.2.4 Hexadecimal

($2^4 = 16$) Hexadecimal, or base-16, is another way to represent binary. This is what we will usually use in labs; it is the easiest for human eyes to process. Hex is base two grouped into fours.

To get a hex number from binary, start at LSB or at the decimal point and sign-extend if needed. Digits range from 0 through 9, a through f. For example,

$$0110\ 1101_2 = 6D_{16}$$

$$1101\ 0111_2 = D7_{16}$$

5.2.5 Examples

This section shows number conversions assuming 8-bit registers. Remember that octal and hexadecimal values are just another way to represent a string of zeroes and ones! In the table below, they are taken from 2's complement representation. Because we are dealing with 8-bit numbers, 136 cannot be represented in any of the signed forms due to overflow. Similarly, -136 cannot be represented in unsigned because it is negative, and any of the other forms due to overflow.

Dec	Unsigned	Signed Mag	1's Comp	2's Comp	Octal	Hex
10	0000 1010	0000 1010	0000 1010	0000 1010	12	A
-10	N/A	1000 1010	1111 0101	1111 0110	766	F6
136	1000 1000	N/A	N/A	N/A	N/A	N/A
-136	N/A	N/A	N/A	N/A	N/A	N/A
27	0001 1011	0001 1011	0001 1011	0001 1011	33	1B
-27	N/A	1001 1011	1110 0100	1110 0101	745	E5

Table 12 Number conversions of a few numbers

5.3 *Floating Point Summary*

During this quarter, you will be expected to learn about floating point numbers. The IEEE standard calls for single- and double-precision floating point numbers. Single-precision numbers are 32 bits long; double-precision are (imagine that!) 64 bits.

5.3.1 **Floats are not reals!**

Floating point numbers are an *approximation* of real numbers. Hence, you cannot represent π as a float, but you can represent 3.14, an approximation of π .

In other words, they do not have infinite precision.

Floats are also not integers: integers are unable to express fractions of a number. Floats sure can.

5.3.2 **Normalized numbers**

A normalized binary number is one of the form

$$1.01110101001\dots,$$

where there is exactly one 1 to the left of the decimal point, and all sorts of stuff to the right. Any number can be normalized if you just move the decimal point, much like scientific notation. For example,

$$111001010101 = 1.11001010101 \times 2^{11}$$

$$0.00101001 = 1.01001 \times 2^{(-3)}$$

so the binary number on the right of each above equation is normalized.

5.3.3 **SPFP**

The IEEE standard for single-precision floating point representation has three parts: the sign, the exponent, and the fractional part (mantissa).

The sign is a single bit. A zero sign implies the number is positive. If the sign bit is set, the number is negative.

The exponent is an 8-bit number represented in bias 127 (ack! what's that?). That means when converting a number into its SPFP format, you must add 127 to get into the bias.

The mantissa is a 23-bit unsigned binary field. It is simply the number after the decimal point, once the binary number is in normalized form.

5.3.4 **The conversion process**

Let's start with your average decimal number.

1. First, convert it to binary.
2. Next, given this binary number, make sure it's normalized. That means moving the decimal point until you have exactly one 1 to the left of the decimal point.
3. Find the sign bit. That's easy: if the number is positive, the sign bit is 0. Else the sign bit is 1.
4. The exponent field is formed from the exponent of the normalized number. Add 127 to the exponent; convert the number to binary. That's the exponent field of the floating point number.

5. The mantissa is an unsigned value formed by sticking everything (or as much as you can) after the decimal point into the 23-bit field¹⁰.
6. Where did the "1." go? The 1 is called the hidden bit and is not actually represented in the format. That saves space for one more bit of important information.
7. Double-check that you have all 32 bits accounted for.

Remember these steps only work if the number is normalized!

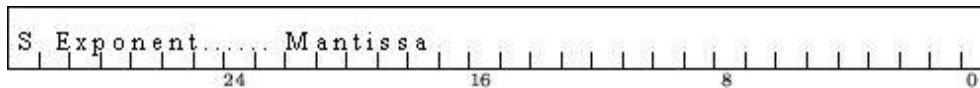


Figure 29 The sign, exponent, and mantissa of a SPFP number.

5.3.5 Example: converting from decimal to SPFP

Here is an interesting example: Let's convert 154.3 to SPFP.

First, we convert it to binary. Converting separately the whole number and the radix point (that's the stuff after the decimal), we get that $154 = 10011010$. For the radix part,

$$0.3 * 2 = 0.6$$

$$0.6 * 2 = 1.2$$

$$0.2 * 2 = 0.4$$

$$0.4 * 2 = 0.8$$

$$0.8 * 2 = 1.6 \text{ (repeats)}$$

the binary representation is $0.0100110011001\dots$ (1001 repeating).

Next, we need to normalize the result.

$$10011010.0100110011001\dots = 1.001101001001100\dots \times 2^7$$

By inspection, the sign bit will be 0.

The exponent will be

$$7 + 127 \text{ (I prefer to do this in binary)} = 134 = 10000110.$$

And the mantissa is the rest of the floating point number, for 23 bits. Make sure you use the mantissa of the normalized radix point number!

So the final SPFP number is

$$0 \ 10000110 \ 00110100100110011001100.$$

That wasn't so bad...

¹⁰ That's not exactly right. The IEEE 754 floating point standard says there must be four rounding modes: round to $+\infty$ (always round up), round to $-\infty$ (always round down), round to even (that's the tricky one), and round to 0 (also known as truncation --- that's the one we're using). Luckily, the first four are (usually) out of the scope of this class.

6 Appendix: Important Tables

This appendix contains important tables and references: the MIPS registers, the ASCII table, the MAL instruction set, the TAL instruction set and machine code, as well as a table of the HC11 registers, and a summary of the procedures in v2_18 (the include file for the HC11).

6.1 MIPS Table of Registers

Register name (what MIPS calls it)	Alternate name (what you call it)	Use (what to do with it)
\$0	\$zero	Register 0 is always zero!
\$1	\$at	(reserved by the assembler)
\$2-\$3	\$v0-\$v1	Return <i>value</i> registers are used to pass function return values. Use these carefully; the compiler also uses them.
\$4-\$7	\$a0-\$a3	<i>Argument</i> registers are used to pass function call values (arguments). Use these carefully; the compiler also uses them.
\$8-\$15	\$t0-\$t7	Use <i>temporary</i> registers freely, but, by calling convention, they are not saved over procedure calls.
\$16-\$23	\$s0-\$s7	Use <i>global</i> registers freely, but, by calling convention, you should save them over procedure calls.
\$24-\$25	\$t8-\$t9	More <i>temporary</i> registers
\$26-\$27	\$k0-\$k1	(reserved by the OS)
\$28	\$gp	<i>Global pointer</i>
\$29	\$sp	<i>Stack pointer</i> —your friend—is used when you push or pop from the built-in system stack.
\$30	\$s8	Another <i>global</i> register (like the other \$s)
\$31	\$ra	<i>Return address</i> register is used when you do a JAL. The return address is saved in this register.
\$f0-\$f30		<i>Floating point</i> registers. We will not use these!

Table 13 MIPS table of registers.

6.2 MIPS System Calls

Are you upset that \$v0 (\$2) is being overwritten, and you don't know why? Well, \$2 is a special register used for system calls like puts, putc, and getc. Note that \$2 is used to pass the unique value of the system call to be executed, whereas \$4 is used to hold the argument to the system call.

The table below will explain how \$2 is used to pass parameters and execute system calls.

For example, let's consider the instruction

```
putc $12
```

which will output the least significant 8 bits of \$12 as a character¹¹. The equivalent TAL code for this pseudoinstruction is

```
addi $2, $0, 11
add $4, $12, $0
syscall
```

which will load the value for the putc instruction into \$2, load the value to be printed into \$4, and execute the system call.

Another example is the pseudoinstruction done used to terminate a program's execution. The equivalent TAL code is

```
addi $2, $0, $10
syscall.
```

Note that done does not take an argument; hence \$4 is not used.

For more information, see the Chapter 10, The Assembly Process, in the course textbook [1].

Function	\$v0 (\$2)
puts	4
putc	11
getc	12
puti	
geti	

Table 14 System calls and the associated value in \$v0 (\$2).

¹¹ That means that if the contents of \$12 are 0x8a098f61, the result of the putc will give 0x61='a'. Only the last byte is printed because MIPS is a Little-Endian machine, referring to its internal byte order.

6.3 ASCII Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Char	Dec	Hx	Oct	Char	Dec	Hx	Oct	Char
0	0	000	NUL	32	20	040	SPACE	64	40	090	@	96	60	140	`
1	1	001	SOH	33	21	041	!	65	41	091	A	97	61	141	a
2	2	002	STX	34	22	042	"	66	42	092	B	98	62	142	b
3	3	003	ETX	35	23	043	#	67	43	093	C	99	63	143	c
4	4	004	EOT	36	24	044	\$	68	44	094	D	100	64	144	d
5	5	005	ENQ	37	25	045	%	69	45	095	E	101	65	145	e
6	6	006	ACK	38	26	046	&	70	46	096	F	102	66	146	f
7	7	007	BEL	39	27	047	'	71	47	097	G	103	67	147	g
8	8	010	BS	40	28	050	(72	48	100	H	104	68	150	h
9	9	011	TAB	41	29	051)	73	49	101	I	105	69	151	i
10	A	012	LF	42	2A	052	*	74	4A	102	J	106	6A	152	j
11	B	013	VT	43	2B	053	+	75	4B	103	K	107	6B	153	k
12	C	014	FF	44	2C	054	,	76	4C	104	L	108	6C	154	l
13	D	015	CR	45	2D	055	-	77	4D	105	M	109	6D	155	m
14	E	016	SO	46	2E	056	.	78	4E	106	N	110	6E	156	n
15	F	017	SI	47	2F	057	/	79	4F	107	O	111	6F	157	o
16	10	020	DLE	48	30	060	0	80	50	110	P	112	70	160	p
17	11	021	DC1	49	31	061	1	81	51	111	Q	113	71	161	q
18	12	022	DC2	50	32	062	2	82	52	112	R	114	72	162	r
19	13	023	DC3	51	33	063	3	83	53	113	S	115	73	163	s
20	14	024	DC4	52	34	064	4	84	54	114	T	116	74	164	t
21	15	025	NAK	53	35	065	5	85	55	115	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	116	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	117	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	120	X	120	78	170	x
25	19	031	EM	57	39	071	9	89	59	121	Y	121	79	171	y
26	1A	032	SUB	58	3A	072	:	90	5A	122	Z	122	7A	172	z
27	1B	033	ESC	59	3B	073	;	91	5B	123	[123	7B	173	{
28	1C	034	FS	60	3C	074	<	92	5C	124	\	124	7C	174	
29	1D	035	GS	61	3D	075	=	93	5D	125]	125	7D	175	}
30	1E	036	RS	62	3E	076	>	94	5E	126	^	126	7E	176	~
31	1F	037	US	63	3F	077	?	95	5F	137	_	127	7F	177	DEL

Table 15 ASCII table of elements.

6.4 MAL Instruction Set

MAL Instruction	Effect	Notes
<i>Arithmetic</i>		
move D, T	Copy T into D	D (destination) is a general register that stores the result of a computation. S (source) is the contents of a general register. T (second source) is either contents of a general register or an immediate.
add D, S, T	$D = S + T$	
sub D, S, T	$D = S - T$	
and D, S, T	$D = S \text{ and } T$	
or D, S, T	$D = S \text{ or } T$	
xor D, S, T	$D = S \text{ xor } T$	
nor D, S, T	$D = S \text{ nor } T$	
not D, S	$D = \text{not } S$	
sll D, S, T	Shift S left by T bits	
srl D, S, T	Shift S right by T bits	
sra D, S, T	Shift S right by T bits	
mul D, S, T	$D = S * T$	
div D, S, T	$D = S / T$	
rem D, S, T	$D = S \% T$	
<i>Memory</i>		
la R, label	R gets address of label	Word Lowest byte, sign-extend Lowest byte, no sign-extension Word Lowest byte (no sign-extension)
li R, constant	R gets constant	
lw R, address	R gets stuff at address	
lb R, address	R gets stuff at address	
lbu R, address	R gets stuff at address	
sw R, address	Stuff in R stored to address	
sb R, address	Stuff in R stored to address	
<i>Branches</i>		
b label	Unconditional branch to label	Otherwise, do not branch
beq S, T, label	If (S = T) branch to label	
bne S, T, label	If (S ≠ T) branch to label	
blt S, T, label	If (S < T) branch to label	
bgt S, T, label	If (S > T) branch to label	
ble S, T, label	If (S ≤ T) branch to label	
bge S, T, label	If (S ≥ T) branch to label	
bltz S, label	If (S < 0) branch to label	
bgtz S, label	If (S > 0) branch to label	
blez S, label	If (S ≤ 0) branch to label	
bgez S, label	If (S ≥ 0) branch to label	
bnez S, label	If (S ≠ 0) branch to label	
beqz S, label	If (S = 0) branch to label	

<i>Jumps</i>			
j	address	Jump to address	RA is return address, the address of the next instruction after the jump-and-link. \$ra (\$31) is an internal general register used to store RA after jump-and-link.
jal	address	Jump to address and save RA in \$ra	
jr	S	Jump, using contents of S as address	
jalr	D, S	Jump, using contents of S as address and save RA in D	
<i>System Calls</i>			
getc	S	Get character into S	S points to a null-terminated string
putc	S	Output character from S to stdout	
puts	S	Output string	
puti	S	Output integer from S	

Table 16 MAL instruction set. Modified from [1].

6.5 TAL Instruction Set, Machine Code

Machine Code	TAL Instruction	Effect
<i>Register</i>		
0000 00ss ssst tttt dddd d000 0010 0000	add D, S, T	$D = S + T$
0000 00ss ssst tttt dddd d000 0010 0010	sub D, S, T	$D = S - T$
0000 00ss ssst tttt dddd d000 0010 0001	addu D, S, T	Overflow ignored
0000 00ss ssst tttt dddd d000 0010 0011	subu D, S, T	Overflow ignored
0000 00ss ssst tttt dddd d000 0010 0100	and D, S, T	$D = S \text{ and } T$
0000 00ss ssst tttt dddd d000 0010 0101	or D, S, T	$D = S \text{ or } T$
0000 00ss ssst tttt dddd d000 0010 0110	xor D, S, T	$D = S \text{ xor } T$
0000 00ss ssst tttt dddd d000 0010 0111	nor D, S, T	$D = S \text{ nor } T$
0000 0000 000t tttt dddd diii ii00 0000	sll D, S, I	Shift S left by I bits
0000 0000 000t tttt dddd diii ii00 0010	srl D, S, I	Shift S right by I bits, shifting in zeros
0000 0000 000t tttt dddd diii ii00 0011	sra D, S, I	Shift S right by I bits, sign-extending
0000 00ss ssst tttt dddd d000 0000 0100	sllv D, S, T	$D = S$ shifted left T bits
0000 00ss ssst tttt dddd d000 0000 0110	srlv D, S, T	$D = S$ shifted right T bits, shifting in 0s
0000 00ss ssst tttt dddd d000 0000 0111	srav D, S, T	$D = S$ shifted right T bits, sign-extending
0000 00ss ssst tttt 0000 0000 0001 1000	mult S, T	HI LO = $S * T$
0000 00ss ssst tttt 0000 0000 0001 1011	div S, T	LO = S / T HI = $S \% T$
0000 00ss ssst tttt 0000 0000 0001 1001	multu S, T	Overflow ignored
0000 00ss ssst tttt 0000 0000 0001 1011	divu S, T	Overflow ignored
0000 0000 0000 0000 dddd d000 0001 0000	mfhi D	D gets contents of HI
0000 0000 0000 0000 dddd d000 0001 0001	mthi S	S is placed in HI
0000 0000 0000 0000 dddd d000 0001 0010	mflo D	D gets contents of LO
0000 0000 0000 0000 dddd d000 0001 0011	mtlo S	S is placed in LO
<i>Immediate</i>		
0011 1100 000t tttt iiii iiii iiii iiii	lui T, I	T = top 16 bits of I
0011 01ss ssst tttt iiii iiii iiii iiii	ori T, S, I	$T = S \text{ or } I$
0010 00ss ssst tttt iiii iiii iiii iiii	addi T, S, I	$T = S + I$
0010 01ss ssst tttt iiii iiii iiii iiii	addiu T, S, I	Overflow ignored
0011 00ss ssst tttt iiii iiii iiii iiii	andi T, S, I	$T = S \text{ and } I$
0011 10ss ssst tttt iiii iiii iiii iiii	xori T, S, I	$T = S \text{ xor } I$

<i>Memory</i>										
1000	11bb	bbbt	tttt	iiii	iiii	iiii	iiii	lw	T, I(B)	T gets word at B+I
1000	00bb	bbbt	tttt	iiii	iiii	iiii	iiii	lb	T, I(B)	T gets byte at B+I, sign-extend
1001	00bb	bbbt	tttt	iiii	iiii	iiii	iiii	lbu	T, I(B)	No sign-extending
1010	11bb	bbbt	tttt	iiii	iiii	iiii	iiii	sw	T, I(B)	T (a word) placed at address B+I
1010	00bb	bbbt	tttt	iiii	iiii	iiii	iiii	sb	T, I(B)	T (last byte) placed at address B+I
<i>Branches</i>										
0000	00ss	ssst	tttt	dddd	d000	0010	1010	slt	D, S, T	If (S < T) then D = 1
0010	10ss	ssst	tttt	iiii	iiii	iiii	iiii	slti	T, S, I	If (S < I) then D = 1
0001	00ss	ssst	tttt	iiii	iiii	iiii	iiii	beq	S, T, I	Branch I instructions. I is a relative value telling how many instructions to branch.
0001	01ss	ssst	tttt	iiii	iiii	iiii	iiii	bne	S, T, I	To find I,
0000	01ss	sss0	0000	iiii	iiii	iiii	iiii	bltz	S, I	1. Address of label – address of next instruction
0001	01ss	sss0	0001	iiii	iiii	iiii	iiii	bgez	S, I	2. Shift off last 2 bits for word-alignment
0001	10ss	sss0	0000	iiii	iiii	iiii	iiii	blez	S, I	
0001	11ss	sss0	0000	iiii	iiii	iiii	iiii	bgtz	S, I	
<i>Jumps</i>										
0000	10ii	iiii	iiii	iiii	iiii	iiii	iiii	j	I	Jump to address I.
0000	11ii	iiii	iiii	iiii	iiii	iiii	iiii	jal	I	To find I, 1. Discard top 4 bits of label 2. Shift off last 2 bits for word-alignment
0000	00ss	sss0	0000	0000	0000	0000	1000	jr	S	Jump, using contents of S as address
0000	00ss	sss0	0000	dddd	d000	0000	1001	jalr	D, S	Jump, using contents of S as address and saving RA in D.

Table 17 TAL and machine code. Modified from [1].

6.6 HC11 Registers

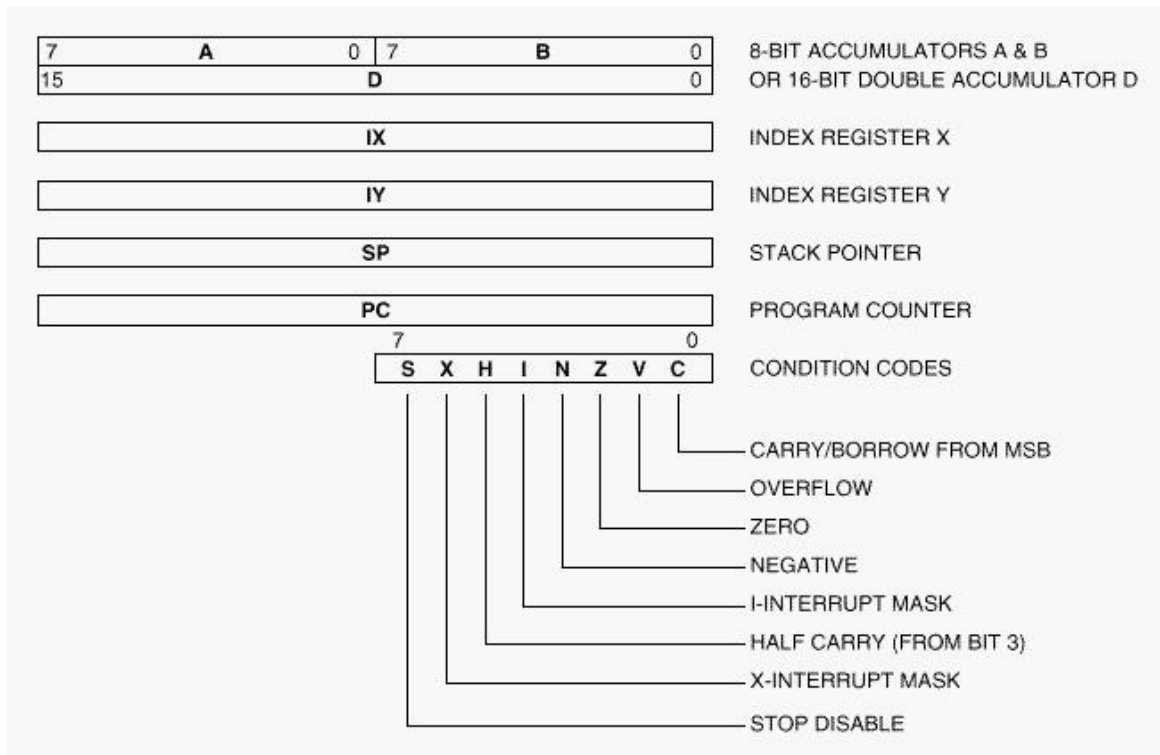


Figure 30 HC11 Registers.

6.7 v2_18 Summary

Procedure	Description
GETCHAR	Reads a single character from the keyboard. A = Character
OUTCHAR	Write a character to the screen. A = Character
OUTSTRING	Write a null-terminated string to monitor. X = Pointer to first character of string
OUTCRLF	Write a carriage-return and line-feed to screen.
CONSOLEINT	Display an unsigned integer to the terminal. D = Unsigned integer
LCD_CHAR	Write a character to LCD. A = Character
LCDLINE	Write a null-terminated string to the LCD. A = 1 goes to line1, or A = 2 goes to line2 X = Pointer to first character of string
LCDINT	Display an unsigned integer to LCD. D = Unsigned integer
WAIT	Software delay loop with 4.096ms resolution. D = length of delay
SETUPSCI	Set up HC11 UART for 9600bps.
LCD_CMD	Write to the LCD Command Register.

Table 18 Summary of procedures in v2_18.

7 References

*Thine permissions are sweeter than sevens
Thine files are as big as your heart
And when you su, it's the heavens!
Yet when you logout—we're apart.*

*Don't shutdown -h now, my darling
And don't recompile right just yet
For when you partition, it's magic...
And when you logout—I forget.*

- [1] Goodman, James and Miller, Karen. *A Programmer's View of Computer Architecture*. New York, Oxford. Oxford University Press. 1993.
- [2] Getting a CATS account.
<http://www2.ucsc.edu/cats/cgi-bin/register.pl>
- [3] *Happy Assembly Class, The* (CE12c Lab Manual).
<http://www.cse.ucsc.edu/classes/cmpe012c/Manual/>
- [4] Computer Engineering 12c Web page.
<http://www.cse.ucsc.edu/classes/cmpe012c/Winter01/>
- [5] MIPS Applications. Accessed 07/2000.
<http://www.mips.com/coolApps/> and <http://www.mips.com/coolApps/s3p3.html>
- [6] Spim Resources and Documentation. Accessed 02/2001.
<http://www.cse.ucsc.edu/classes/cmpe012c/Manual/spim.html>
- [7] *HC11: M68HC11 Reference Manual*. © Motorola, Inc. 1991.
- [8] HC11 v2_18.asm. Accessed 02/2001
http://www.cse.ucsc.edu/classes/cmpe012c/Manual/v2_18.asm
- [9] HC11 LCD Documentation. Accessed 02/2001.
<http://www.cse.ucsc.edu/classes/cmpe012c/Manual/lcddisplay/>
- [10] Baskin Engineering Lab Support. Accessed 12/2002.
<http://www.soe.ucsc.edu/administration/labs/>
- [11] PuTTY: a free Win32 telnet/SSH client. Accessed 12/2002.
<http://www.putty.nl/>

Alex's12c Web Page:

<http://www.cse.ucsc.edu/Manual/Alex/>

8 Index

Academic honesty.....	4	LCDINT.....	57
algorithms		light pattern.....	53
bit printing.....	22	microkit checkout.....	43
factorial.....	45	pointers.....	39
Fibonacci.....	47	prefixes.....	36
integer input.....	20	procedure calls.....	48
integer output.....	21	programming environment.....	36
interrupts.....	48	pulse accumulator.....	54
MIPS.....	20	quirks.....	41
pseudorandom number generator....	56	recursion.....	44, 47
recursion.....	44	registers.....	36
rot-13.....	43	relocation truncated to fit.....	41
RPN calculator.....	24	s-record.....	35
assembly language.....	1	stack.....	36, 48
bits.....	1	stop.....	51
CATS account.....	5	v2_18.....	49
checkoff.....	7, 10	variables.....	39
collaboration.....	4	WinIDE/PEMicro.....	39
comments		interrupts.....	<i>See</i> HC11:interrupts
block.....	5	jokes	
in-line.....	6	ASCII character.....	30
examples		little endian.....	93
echo.....	49	locker.....	6
factorial.....	50	logical operations.....	86
pulse accumulator.....	57	microcontroller.....	35
Exceed.....	14	MIPS.....	11
floating point.....	90	algorithms.....	<i>See</i> algorithms
grading.....	7	code examples.....	30
HC11		character I/O.....	30
addressing modes.....	37	procedure calls.....	32
BELS.....	43	development software.....	14
binutils.....	40	directive.....	12
branching.....	38	done.....	93
comments.....	37	function calls.....	26
downloading s19.....	40	instructions.....	11
failed build.....	41	jal26	
GNU binutils.....	35, 41	little endian.....	93
hardware.....	35	MAL to TAL.....	27
hc11build.....	40	processors.....	11
instruction set.....	39	putc.....	<i>See</i> 93
instructions.....	37	quirks.....	19
interrupts.....	48, 53, 54	register usage.....	32

registers.....	11	haiku in MAL	54
SAL.....	11	How sweet are your numbers.....	1
system calls.....	93	programming style	5
normalized number	90	Thine permissions are sweeter	101
number conversion.....	87	secure shell	14, 40
binary	87	spimsal.....	17, 19. <i>See</i> xspimsal
binary with radix point.....	88, 91	for Windows	19
floating point.....	90, 91	SSH.....	<i>See</i> secure shell
hex	89	stack	
octal	89	HC11.....	36
plagiarize	4	RPN <i>See</i> algorithms:RPN calculator	
README	6	submit	6
Reverse Polish Notation.....	<i>See</i>	xspimsal	14
algorithms:RPN calculator		alternatives.....	19
rhymes		buttons	17
Eight bits are in register A	49	xterm.....	14