Learn ASM

COLLABORATORS			
	TITLE : Learn ASM		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 23, 2011	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

0.1 Overview

0.1.1 Introduction

The lsa processor project was created for a number of different reasons. Teaching assembly language is but one of them. The architecture was designed to balance real world usability and understandable simplicity.

At the moment, and probably forever, this document is a work in progress. Any feedback you choose to offer will be read and appreciated, doesn't mean that everything suggested or offered will get included. Feedback helps to understand what the reader needs to see or not see to make the learning process successful. Since we do not all learn things the same way the presentation has to be averaged in some form.

0.1.2 Why learn assembly language?

Perhaps the best argument is that it will help a programmer better understand how a computer works. A few Universities still require students to take an assembly course for this reason. This understanding can help you write better programs. It will help you understand what higher level languages do. All programming languages essentially result in the generation of machine code. Assembly language is probably closer to machine code than any other language.

What are the advantages of assembly language over other languages? The simple answer is speed, space and capability. In the early years of computing when programmers were working with a few megabytes of memory and relatively slow processors, speed and space demanded the use of assembly language. Programming languages have evolved to make it easier for the programmer to write programs. This has resulted in more overhead, slower execution and programs that consume large amounts of space. Of course processor speeds and memory capacity have correspondingly increased to accommodate these programs. This leaves capability. Assembly language can do some things that higher languages cannot (even now). The question is do you need to do these things. As a modern day programmer, probably not. This leaves understanding. By learning assembly language you learn the basic operations of computer processors and this will help you understand how programs work at the machine level.

Despite what you may have heard, there was a time when there were accomplished assembly language programmers that could write a 16 megabyte program that would do the same thing that would take a high level language program of 200 megabytes or more to do. The other difference was the assembly program would do it in 1/5th the time. At one time the programmers who developed operating systems and higher level languages (Systems Programmers or Systems Developers) worked exclusively with assembly language. I doubt that there are any of those assembly language programmers left still writing assembly code.

It is important that you start with simpler processor instruction sets to learn assembly before you progress to more complex (not necessarily more powerful) processors such as the x86. So we have developed a tool that can be used to teach assembly language. This tool consists of a simulated (non-physical) processor, lsa-sim, and an associated "assembler", lsa-as, that we will use introduce you to the fundamentals of assembly language. Advance topics in assembly such as masking, instruction cpu cycles, optimization, packed data, logic design, etc. will not be taught here (at least not now).

0.1.3 What Assembly Language is

Assembly Language (asm) is a programming language like C or C++ or Java or Python. You create text files that follow syntax rules and a tool is used to parse that file. The primary difference from other languages like C, Python, etc is that the term Assembly Language is generic, but the language itself is not. Each different processor type or family has a different assembly language. Although the functions that assembly language instructions perform may be similar from one architecture to another, there should be no expectation that one syntax will resemble another. Think about a door on a business or office building. It performs the same function, but think about how many different sizes, shapes and types (revolving, sliding, hinged, etc.) of doors there, are.

Processors operate on bits and bytes known as machine code. Although certainly possible to program in machine code, you would not want to for any significant programming problem. Assembly Language is a human readable (and writable) form of machine code. There is often a one to one relationship between an assembly language instruction and a machine instruction that the processor operates on. The bits and bytes in the instruction are assembled based on information in the assembly language instruction.

Assembly language programming involves more than just the assembly language instructions. The program used to read and process an assembly language program, for lack of a better term, “assembler,” will require other information it needs to do its job.

Unlike other programming languages, assembly language normally does not have an organization or standards body that carefully defines the rules of the language. Normally, the company that invents the processor creates the initial assembly language, along with the machine code instructions, of course.

A company that creates the assembler tool may support the original assembly language and add additional syntax to support the programmer that does not translate to instructions (like a #define or #pragma in C for example). This is not a hard and fast rule. Sometimes the folks creating the assembler tool modify the original or create their own assembly language that translates to the same machine code instructions.

The assembly language concept is so ungoverned that, the terms used here assembly language, machine language and assembler; have all be used to describe the programming language (the text you type in a file). So it is hard to have any kind of discussion about the difference between the ASCII syntax you type in the file, the bits and bytes the processor runs on, and the program you want to run that reads that syntax and turns it into bits and bytes.

The good news though, is once you learn asm for one processor, asm for another is much easier to pick up. If you choose to continue to learn asm for other processors (which I highly recommend) each one gets that much easier. Both assembly languages and other programming languages become easier to learn once you understand how the processor really operates, and are able to break programming problems down into very small sequences of steps.

0.1.4 What Assembly Language is Not

Assembly language only gets you closer to processor core, it will not teach you how to program peripherals like video cards and USB controllers. Often it does not involve on chip peripherals like gpio and uarts and timers. Peripherals are normally programmed using languages like C, if you are learning asm to try to understand the peripherals, you are in the wrong place.

0.1.5 What to expect

This is an attempt to teach by doing. Each lesson should be completed in order as most lessons build upon the foundation built by the lessons before them. Resist the urge to cut and paste, it will be in your best interest to type these programs in manually and debug typos you might make along the way. More than other languages, assembly carries with it serious crashes and chaos and debugging your way through those with the tools is often more important than the language itself. The good news is that you won’t be melting down and letting smoke out of a simulated processor if you make mistakes. On real hardware melt downs can happen.

There is an expectation here that you already have programming experience in some higher level language. Ideally you have had some C programming exposure as the lessons often use C code to show the counterpart to the assembly language. It is also assumed that you understand binary and hex numbers and operations. Likewise, these lessons are not going to teach you how to use a text editor or a command line, your programming experience should already have provided that knowledge.

0.1.6 Some Terminology and Notation

We have adopted some commonly used terminology in these lessons.

If a bit is one it is said to be “set”. Conversely, if a bit is zero it is said to be “clear”. These terms are commonly used with special bits called “flags”.

The term register is commonly used in assembly language for any processor. It is a place where bits are stored, similar to memory but much more intimate with the processor.

The registers are identified with the lower case letter “r” followed by a decimal number between 0 and 127 that identifies the register number. Just like unique variable names in a C program this gives us 128 unique register names.

rs or rd in an instruction refers to the source register and the destination register. The source register us normally read but not modified by an instruction, where the destination is normally both read as an input to the instruction, but is also modified by that instruction. For example an add instruction takes rd and rs as inputs and stores the result in rd. rd = rd + rs Rd can be any one of the general purpose registers r0 - r15 for an add instruction, likewise rs can be any one of r0 - r15. For times when it does not matter what specific register is being used rd or destination register or rs or source register is used as a placeholder. The assembly language itself requires a numbered register.

Often we refer to the contents of the register in the text just by saying “r7”, “r0 is” or “set r6 to” rather than saying “the content of r7” , “the content of r0 is” or “set the content of r6 to”. The registers have an intimate relationship with the processor they are a major player in the logic. When you speak of the Queen of England you normally say "the Queen" instead of "the person who holds the position of Queen". "r7 is the address in this instruction" rather than "the contents of r7 are the address in this instruction".

As to the contents of a register. When it gets down to it, registers hold bits. If you are here from a high level language the syntax of those languages give the impression that there might be a difference between a byte and a character. A pointer or a integer, etc. Registers and memory locations hold bits. Those bits have no meaning of any kind until they are used, and the meaning is temporary for that one operation, after that operation they are just bits again. Even when changing from one high level programming language to the next that can be a struggle for some programmers. If you study twos complement for example, you will find that for add and subtract operations those operations have no notion of signed or unsigned. The logic doing the math itself does not care because the bit patterns coming in and the bit patterns going out are the same (for those operations). You may have experienced some of this if you have programmed in C. For example you may have some pointer int *p; Then at some point you might increment that pointer to the next item p++; For the brief moment that that increment occurs, what was defined as an address held in the variable p breifly becomes just a number, an operand in an add, operation, when the add operation is complete, the bits are returned to the pointed and the pointer is used to point to something those bits have changed from being the sum of an add operation to being an address in memory again.

To help aid in the readability of the language, there are a few symbols used to quickly see what a register or numbers in the language are representing, temporarily. For example when you see brackets around a register, [r6], for that instruction the contents of r6 are an address in the memory space. Memory space .vs. memory is a carefully chosen term the processor does not, at the instruction level, know what the address means. Some addresses, most in fact, do indicate a location in memory a place that holds some bits. A few addresses in the memory space are actually registers to some other hardware affectionatly called peripherals. A video card for example has some addresses that are memory but special memory because after you write them those bits become colors on a monitor. And some of the addresses you write to are control registers in that video card telling the video card how many pixels wide and tall the screen is goigng to be. Another symbol you will see is the pound sign (#), this indicates that there are going to be some numbers that follow. Those numbers are sometimes called an immediate, because that number is encoded directly into the instruction. You might think of them as a constant or an integer, a number of different terms. This instruction set calls them immediates and they are always preceeded by a pound sign.

Various programming languages have ways of representing binary, decimal, octal, hexadecimal, etc numbers. If in a programming manual I wrote these three numbers 10, 10, 10 as is you might assume they were all three the decimal number ten. But if I add something for clarity 0x10, 10, 0b10, well two of those you probably know, 0x10 comes from the C language and it means hexidecimal, base 16, 0x10 is the same as the decimal number 16. 0b10 is rarely if never seen, but when used here it means binary, the numbers that follow are binary. 0b10 is the same as the decimal number 2. 0b10011 is the same as 0x13 is the same as 19. If a number is shown without some comment or one of these prefixes it can be assumed to be decimal.

Other C language symbols are used as well:

```
& = .and.
| = .or.
^ = .xor.
~ = .not.
++ = increment
-- = decrement
= (in label instruction) = “the address of”
```

0.1.7 Instruction set

The lsa instruction set is defined in the file lsa-isa.txt. These lessons will walk you through understanding how to use the instructions defined in that document. Once exposed you can use your programming experience to organize these instructions to write programs that interest you or perform a certain task. There are some additional instructions that the lsa-as assembler allows you to use to save typing or to add to readability. These translate directly into an instruction defined by the instruction set.

The lsa instruction set was invented specifically for educational purposes. It has many goals, teaching assembly language is but one of them. An assembler lsa-as.c has been provided as well as an instruction set simulator (software that simulates the parsing and execution of binary machine code and simulating the operation of those instructions). The lsa processor project has number of other educational goals that are not discussed in this document.

0.1.8 Addressing and Memory Space

Unlike most processors you will experience, the lsa processor is not set in stone or burned into silicon. There are a few choices that can be made when using it, and those choices can and will change depending where it is used and why. For example the simulator allows for almost all of the memory space to be actual memory, but if you were to have this processor implemented in hardware you likely would not have memory behind all of those addresses. For example the computer you are reading this on may have 32 bit addressing, (if not probably the one in the attic that you retired does) but that does not mean that it automatically has 4GBytes of memory, it can operate just fine with 1GByte. Likewise the lsa simulators have only a few memory locations that cause peripheral like things to happen, when implemented in silicon there would need to be many peripherals added and addresses assigned.

The lower address space for the lsa processor is designed to be ram, starting at address 0x0000 and going up. The upper addresses are where the peripherals will start, perhaps just the 0xF000 - 0xFFFF range or perhaps 0xE000 to 0xEFFF. These lessons are going to assume that at a minimum 0x0000 to 0x0FFF are addresses into memory.

Normally when programming in a higher level language there are rules or function calls, etc that are needed to allocate memory and the operating system keeps you corralled in a small, fixed, private subset of the memory space for that system. Assembly language itself does not put limits on what addresses you can or cannot use, it is true that in some systems, even at the assembly language level MMU's (memory management units) will prevent you from going outside a space of memory that has been allocated to you. Normally though and in particular with this processor and simulator, you are free to use any address you like, it is your responsibility as an assembly langauge programmer to not choose addresses which for example might hold instructions, if you happen to change a memory location holding and instruction that you were going to execute, that can and will change the way your program runs, sometimes leading to a crash, sometimes not. Likewise there is no malloc() function or anything like that, from the moment the processor boots and starts running the entire memory space is already allocated to you. It is up to you to manage how much of that memory is for the program, how much is data that you may need to store things and how much is for the stack which you will learn about.

0.2 Lesson 1: Building tools and first simulation

These lessons require the use of the lsa assembler (lsa-as) and instruction set simulator (lsa-sim). These are simple C programs that need to be compiled.

From your computers command line/prompt run the gcc command:

```
> gcc
```

You should see:

```
gcc: no input files
```

Your output may vary, if you get some sort of file not found error then you need to install gcc or some other C compiler. For example clang or cl or llvm-gcc, etc. If you have a C compiler proceed to the heading "Compile lsa" below.

Installing the C compiler to build the tools is beyond the scope of this lesson. In case it helps though, if on windows look for the mingw32 or cygwin projects (which will provde a gcc), or look for llvm (and use clang or llvm-gcc to compile).

If running Ubuntu linux, then run the command:

```
> sudo apt-get install build-essential
```

This will give you a C compiler (gcc) among other things.

Once you have a C compiler compile the lsa-as.c and lsa-sim.c C source code files. If you are using linux run the commands:

```
> gcc lsa-as.c -o lsa-as
> gcc lsa-sim.c -o lsa-sim
```

If you are running on Windows instead of -o lsa-as and -o lsa-sim use -o lsa-as.exe and -lsa-sim.exe. If using llvm then use llvm-gcc or clang instead of gcc. If using Microsoft C then us cl instead of gcc, etc.

For each lesson there will be some assembly language code between lines that look like this: ---- lesson01.s ---- The primary task (other than learning) is to re-type the lines of that program into a text file on a text editor. Do not include the ---- lesson01.s ---- lines themselves. It is time to perform this typing task for lesson 1:

```
---- lesson01.s ----
    b reset
    b reset
reset:
    halt
---- lesson01.s ----
```

For each lesson including this one (do this now) save the file, perhaps using the suggested filename (lesson01.s), not required but suggsted.

The next step is to assemble the program

```
> ./lsa-as lesson01.s
(if running windows use ./lsa-as.exe)
```

The lsa-as program will spit out some information which down the road may or may not be useful. It is all documented in the lsa-as.c source code (grin).

```
mem[0000]=0x8300C000
mem[0001]=0x8300C000
mem[0002]=0x8300C000
mem[0003]=0x8300C000
mem[0004]=0x8000FFFF
lab:
<1> [3][0000][reset]
<2> [3][0001][reset]
<3> [3][0002][reset]
<4> [3][0003][reset]
<5> [0][0004][reset]
mem[0000]=0x8300C003
mem[0001]=0x8300C002
mem[0002]=0x8300C001
mem[0003]=0x8300C000
mem[0004]=0x8000FFFF
```

lsa-as takes lesson01.s and creates the file named lesson01.s.csv (simply tacks a .csv on the end of whatever file name you feed it). And as the name implies the output of lsa-as is a csv file. Most assemblers do not output this format of a file, usually you get some sort of object file or some flavor of binary file. The term binary is a bit of a misnomer as some binary files are ASCII (like intel hex and srec). Others binary file formats do contain binary portions of executable instructions, wrapped by file related stuff (elf, coff, exe, etc). And some binary files are nothing but the bits and bytes you put in a memory of some sort for the processor to execute.

Now we are ready to run the simulator.

```
> ./lsa-sim lesson01.s.csv
```

When the simulation hits the halt instruction it stops and prints out some statistics:

```
fetch_count 2
read_count 0
write_count 0
```

The lessons from here on out will assume you know how to run lsa-as and lsa-sim commands. Note: If/when you change the lesson.s file do not forget to assemble it before running the simulator.

So what does lsa.as and lsa-sim programs do?

All processors have some sort of entry point they use to boot up, a way or place to figure out where in memory the program starts and a way to start executing. Most use some sort of vector table, which can go by many names, an interrupt vector table, an exception vector table, etc. For most processors, that table contains a list of addresses (vectors). The location in the table determines what code can be found at that address. Usually, at a minimum, you will have a reset vector, which is the code that is run as soon as the processor comes out of reset (when it starts to boot). The the other common vector is an interrupt vector, code that is run when a interrupt occurs. Some processors have a vector for each different interrupt line or entries for undefined instructions or data aborts, etc.

The lsa processor defines address 0x0000 as the reset vector and address 0x0001 as the interrupt vector. Note that unlike most processors that use byte based addressing, the lsa processor uses word base addressing address 0x0001 defines the second 16 bit quantity in memory, address 0x0002 is the third, and so on to 0xFFFF which is the last 16 bit memory location or address in the lsa processors address space.

At the moment the lsa processor does not support interrupts, but address 0x0001 is reserved for the interrupt vector when that time comes. So all of the lessons will contain two items in the vector table.

Unlike many processors the contents of address 0x0000 and 0x0001 are not simply addresses to some code for a handler, instead those locations contain instructions. The lsa-as assembler assumes that the first two instructions in the file are the instructions to execute when a reset or interrupt happens.

One part of assembly language programming that is not actual instructions are labels. lsa-as accepts strings of text (with no spaces) that end in a colon as a label. Just like you would see in C. Labels ultimately lead to direct or relative addressing, if you use labels instead of immediates, the lsa-as assembler will take care of computing the immediate for you and inserting it into the instruction. Not that you have ever used a goto in C perhaps, you probably learned about it, imagine if the goto in C instead of using a label had a line number in the program. It would not take you long before you changed some code in that file causing the line of code you wanted the goto to jump to be a different line number, then you would have to go back and fix all the affected gotos without making any mistakes, and this would be quite painful. Although assembly language strives to give you complete control over the creation of the instruction that will be executed by the processor, letting the assembler compute addresses from the labels will help preserve your sanity.

The unconditional branch instruction will be taught in a later lesson, for now the letter b followed by a label name (without the colon) means goto the place in the program where you find the label name (with the colon). So the first branch to lesson01 in the program is the reset vector, and the second is the interrupt vector.

Most processors do not have a halt instruction. The ones that do often use that instruction to put the processor in a low power sleep state. This processor has one for educational purposes only. When the simulator hits that instruction it stops. Otherwise the simulator will execute forever (normally a ctrl-c will get you out of it if you happen to make a programming mistake that leads to an infinite loop). Normally processors are always running, always executing something. If your software has nothing better to do then it will often sit in a tight “while” loop waiting for something to do, but rarely do they actually stop executing, so the concept of a processor halting is a bit strange.

0.3 Lesson 2: Registers and Immediates

0.3.1 Code

```
---- lesson02 ----
    b lesson02
    b lesson02
lesson02:
    llz r5,#0x34
    lhz r7,#0xFF
    stw [r7],r5
    halt
---- lesson02 ----
```

0.3.2 Output

When you simulate this lesson the output will contain:

```
show: 0x0034
```

0.3.3 Description

The lsa processor has a number of general purpose registers. General purpose register is sometimes abbreviated gpr. The first few gpr's (r0,r1,r2) are not completely general purpose as they have special purposes too, for now let's avoid them. r5 and r7 are definitely general purpose. Registers are used like general purpose local variables in a program. Unlike a high level language, though, you have a finite number of these registers and you must re-use them (if your program needs more than a handful of variables).

Being a load/store architecture processor, most of the instructions use only registers for operands. If you want to add 7 to something you have to put a 7 in a register and then perform the add instruction using only registers. So that the bulk of the instructions can use only registers there is a groupl of instructions in the lsa processor that are used for putting constants in registers. For example to perform that add + 7 you need to put a 7 in a register somehow.

The llz instruction stands for load immediate lower (half) and zero upper (half). Because the lsa instructions are a fixed length of 16 bits you are limited as to how big an immediate can be. You cannot squeeze 16 bits of data and some bits to tell the processor this is a load immediate instruction instruction into 16 bits. The load immediate instructions have room for eight bits of data and one bit to indicate whether you want to zero the other eight bits in the register or leave them alone.

There are four flavors of load immediate instructions.

```
ll  rd,#immed   load immediate lower half
lh  rd,#immed   load immediate upper half
llz rd,#immed   load immediate lower half and zero upper half
lhz rd,#immed   load immediate upper half and zero lower half
```

The destination register, rd, is normally the first register specified, before the first comma.

The load immediate instructions modify eight bits of a register by placing those eight bits in the half of the register specified. The zero other half option gives you control over all 16 bits in the register. When the z is used the other half is zeroed, if the z is not present the other half of the register is not modified by the instruction. An ll r5,#0x34 will put 0x34 in the lower bits of r5 and leave the upper eight bits untouched. llz r5,#0x34 will put 0x34 in the lower eight bits of r5 and 0x00 in the upper eight bits.

So knowing all of this, the program sets r5 to 0x0034 and r7 to 0xFF00.

lsa-sim has a few "peripherals", for lack of a better term. Memory location 0xFF00 is for educational purposes when using the simulator. Placing data (for example 0x0010) in memory location 0xFF00 will "print" that data in the output with a "show" label e.g. show: 0x0010. Each time you place or "write" data to location 0xFF00 in a program a new "show:" line will be "printed." This feature will allow us to examine registers and follow the program execution in these lessons. Like the halt instruction this feature is not something you see in the real world. Normally you would compute ASCII characters and send them to a serial port (lsa-sim has a memory mapped I/O address for that as well).

stw is the store word instruction. A word in the lsa instruction set is defined as 16 bits. Store word means store or write a word to memory. There are a number of stw options which this lesson will not get into, the basic stw is used here. The brackets ([]) around the register give us a visual reminder that this register is being used to specify an address in memory for this instruction. In our program, the 0xFF00 in r7 is an address in memory. The second operand in an stw holds the data to be written to that memory location. The stw instruction we are using here is defined in the form:

```
stw [rd],rs
```

When this instruction is executed in the lesson01 program, the value 0x0034 will be written to memory space at address 0xFF00. In C think of r7 as being a pointer and r5 being a variable:

```
unsigned short r5,*r7;

r5=0x0034;
r7=(unsigned short *)0xFF00;
*r7=r5;
```

0.4 Lesson 3: Simple Loads and Stores

0.4.1 Code

```
---- lesson03 ----
   b lesson03
   b lesson03
lesson03:
   llz r5,#0x34
   lhz r7,#0xFF
   stw [r7],r5
   lhz r8,#0x01
   lhz r9,#0x00
   stw [r8],r5
   ldw r9,[r8]
   stw [r7],r9
   halt
---- lesson03 ----
```

0.4.2 Output

```
show: 0x0034
show: 0x0034
```

0.4.3 Description

Most of the memory implemented in the lsa-sim simulator is available for generic read/write use. In an embedded system like this you need to carefully manage your memory allocation. This program is about a dozen instructions, so it is about a dozen memory locations. Address 0x0100 leaves plenty of room for code.

r8 is loaded with the value 0x0100 r9 is zeroed for demonstration purposes r5 is stored in memory at address 0x0100 r9 is read from memory using the ldw (load word) at address 0x0100 r9 is then written to address 0xFF00 (show: address) so we can see its value.

This is the C equivalent:

```
unsigned short r5,*r7,*r8,r9;

r5=0x0034;
r7=(unsigned short *)0xFF00;
*r7=r5; /* show 0x0034 */
r8=0x0100;
r9=0x0000;
*r8=r5;
r9=*r8;
*r7=r9; /* show 0x0034 */
```

0.5 Lesson 4: Load/Store post increment

0.5.1 Code

```
---- lesson04 ----
    b lesson04
    b lesson04
lesson04:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    lhz r8,#0x01

    llz r9,#0x11
    stw [r8++],r9

    llz r9,#0x22
    stw [r8++],r9

    llz r9,#0x33
    stw [r8++],r9

    llz r9,#0x44
    stw [r8++],r9

    lhz r8,#0x01

    ll  r8,#0x00
    ldw r9,[r8]
    stw [r7],r9

    ll  r8,#0x01
    ldw r9,[r8]
    stw [r7],r9

    lhz r8,#0x01

    ldw r9,[r8++]
    stw [r7],r9

    ldw r9,[r8++]
    stw [r7],r9

    ldw r9,[r8++]
    stw [r7],r9

    ldw r9,[r8++]
    stw [r7],r9

    halt
---- lesson04 ----
```

0.5.2 Output

```
show: 0x1234
show: 0x0011
show: 0x0022
show: 0x0011
show: 0x0022
show: 0x0033
show: 0x0044
```

0.5.3 Description

When you use a ++ after the address register in a stw or ldw, the address in the register is used for the store or load, then the register is incremented.

Lsa-as allows for comments in the assembly language. Whenver a semicolon (;) is encountered from the semicolon to the end of the line is a comment. The commented code below illustrates what is going on in the program.

```
    lhz r8,#0x01      ; r8 = 0x0100
    llz r9,#0x11      ; r9 = 0x0011
    stw [r8++],r9     ; ram[0x0100] = 0x0011 then r8 = 0x0100+1 = 0x0101
    llz r9,#0x22      ; r9 = 0x0022
    stw [r8++],r9     ; ram[0x0101] = 0x0022 then r8 = 0x0101+1 = 0x0102
```

The program then goes to verify the memory two ways, one is to manually set the address in r8, the other is to use an ldw post increment.

The C equivalent would be:

```
unsigned short r5,*r7,*r8,r9;

r5 = 0x1234;
*r7 = 0xFF00;
*r7 = r5; /* show 0x1234 */
*r8 = 0x0100
r9 = 0x0011;
*r8++ = r9;
r9 = 0x0022;
*r8++ = r9;
r9 = 0x0033;
*r8++ = r9;
r9 = 0x0044;
*r8++ = r9;
*r8 = 0x0100
r9 = *r8;
*r7 = r9 /* show 0x0011 */
*r8 = 0x0101
r9 = *r8;
*r7 = r9 /* show 0x0022 */
*r8 = 0x0100
r9 = *r8++;
*r7 = r9 /* show 0x0011 */
r9 = *r8++;
*r7 = r9 /* show 0x0022 */
r9 = *r8++;
*r7 = r9 /* show 0x0033 */
r9 = *r8++;
*r7 = r9 /* show 0x0044 */
```

0.6 Lesson 5: Load/Store post decrement

0.6.1 Code

```
---- lesson05 ----
    b lesson05
    b lesson05
lesson05:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    lhz r8,#0x01

    llz r9,#0x11
    stw [r8--],r9

    llz r9,#0x22
    stw [r8--],r9

    llz r9,#0x33
    stw [r8--],r9

    lhz r8,#0x01
    ll  r8,#0x00
    ldw r9,[r8]
    stw [r7],r9

    llz r8,#0xFF
    ldw r9,[r8]
    stw [r7],r9

    llz r8,#0xFE
    ldw r9,[r8]
    stw [r7],r9

    lhz r8,#0x01

    ldw r9,[r8--]
    stw [r7],r9

    ldw r9,[r8--]
    stw [r7],r9

    ldw r9,[r8--]
    stw [r7],r9

    halt
---- lesson05 ----
```

0.6.2 Output

```
show: 0x1234
show: 0x0011
show: 0x0022
show: 0x0033
show: 0x0011
show: 0x0022
show: 0x0033
```

0.6.3 Description

Just like the post increment, after the address register is used, the address is decremented and saved in that register.

```
    lhz r8,#0x01    ; r8 = 0x0100
    llz r9,#0x11    ; r9 = 0x0011
    stw [r8--],r9   ; ram[0x0100] = 0x0011, then r8 = r8 - 1 = 0x00FF
    llz r9,#0x22    ; r9 = 0x0022
    stw [r8--],r9   ; ram[0x0101] = 0x0022, then r8 = r8 + 1 = 0x00FE
```

After writing a few locations, the program reads back those locations using manually set addresses and by using the post decrement with the ldw instruction.

0.7 Lesson 6: Load/Store pre increment and pre decrement

0.7.1 Code

```
---- lesson06 ----
    b lesson06
    b lesson06
lesson06:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    lhz r8,#0x01

    llz r9,#0x11
    stw [++r8],r9

    llz r9,#0x22
    stw [++r8],r9

    llz r9,#0x33
    stw [++r8],r9

    stw [r7],r8

    lhz r8,#0x01
    ll  r8,#0x01
    ldw r9,[r8]
    stw [r7],r9

    ll  r8,#0x02
    ldw r9,[r8]
    stw [r7],r9

    ll  r8,#0x03
    ldw r9,[r8]
    stw [r7],r9

    lhz r8,#0x01
    ll  r8,#0x04

    ldw r9,[--r8]
    stw [r7],r9

    ldw r9,[--r8]
    stw [r7],r9

    ldw r9,[--r8]
    stw [r7],r9

    stw [r7],r8
    halt
---- lesson06 ----
```

0.7.2 Output

```
show: 0x1234
show: 0x0103
show: 0x0011
show: 0x0022
show: 0x0033
show: 0x0033
show: 0x0022
show: 0x0011
show: 0x0101
```

0.7.3 Description

As you can guess, using a ++ before the address register in an stw or ldw instruction, causes the address to increment before it is used by that instruction. Likewise using a -- before the address register in an stw or ldw instruction causes the address to decrement before it is used.

0.8 Lesson 7: Load/Store PC Relative

0.8.1 Code

```
---- lesson07 ----
    b lesson07
    b lesson07
lesson07:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    lpc r9,jack
    stw [r7],r9

    llz r9,#0x77
    spc jack,r9

    llz r9,#0x88
    spc jill,r9

    lpc r9,jack
    stw [r7],r9

    stw [r7],r0

    lpc r9,#0x03
    stw [r7],r9
    halt

jack:   .word 0xABCD
jill:   .word 0
---- lesson07 ----
```

0.8.2 Output

```
show: 0x1234
show: 0xABCD
show: 0x0077
show: 0x000F
show: 0x0088
```

0.8.3 Description

Early on it was mentioned that not all of the general purpose registers are really general purpose. Register r0 in the lsa processor is also known as the program counter or pc. It is used by the processor to hold the address in memory where the instructions are being fetched. If you think about it in terms of the steps the processor has to do. The processor needs to fetch, or read, from memory the instruction to be executed. After reading the instruction it is incremented to the next location. If nothing happens to cause it to change that is the address/instruction that will be fetched next.

If you use r0 directly in an instruction as we do in stw [r7],r0

```
    MEM,0x000E,0x4708,END,,
```

The output shows that r0 contains the current instruction address + 1

```
    show: 0x000F
```

assuming the program was entered exactly as written above.

The keyword .word tells the lsa-as assembler that you want to fill that memory location with the item specified after the keyword.

```
jack:   .word 0xABCD
```

The label is used so that our program can access that memory location but allow the assembler to compute the address for you.

This program has two memory locations defined with static values, jack and jill.

```
jack:   .word 0xABCD
jill:   .word 0
```

As with instructions, the assembler determines, by its location in the program, the address for the variable. For this program the address for jack is 0x0012 and for jill 0x0013 (if you assemble the program exactly as written above).

```
    <27> [0][0012][jack]
    <28> [0][0013][jill]
```

```
    MEM,0x0012,0xABCD,END,,
    MEM,0x0013,0x0000,END,,
```

The first time we try to load jack is on line 9 of the lesson.

```
    lpc r9,jack
```

Which will be an instruction at address 0x0006.

```
    <9> [1][0006][jack]
```

Once encoded this instruction becomes:

```
    MEM,0x0006,0x090B,END,,
```

The lpc (load pc relative) instruction takes the program counter (r0) which is the address of the instruction plus one at that time. It adds the constant encoded in the instruction and uses that sum as the address to read from memory (the sum is not stored anywhere it is discarded after the memory read is complete). The constant or immediate encoded in the instruction for the lpc instruction uses the lower eight bits of the instruction. In this case 0x0B is the lower eight bits of the 0x090B instruction.

So for this instruction, 0x090B, at address 0x0006, that means the memory location to be read is 0x0006 + 1 + 0x0B = 0x0012. Which is the address for the label jack. The assembler did what we asked it to do.

The spc (store pc relative) instruction is the same as lpc except it is a store. When used like this in the lesson

```
    spc jill,r9
```

tells the assembler that we want to write the contents of r9 to the address that is defined with the jill label.

When label names are used, the assembler takes care of figuring out the address for you. But, there is a rarely used form of these instructions that allows you to force the immediate that is encoded in the instruction.

```
[0x000F]:0x0903         lpc r9,#0x3
[0x0010]:0x4798         stw [r7],r9
[0x0011]:0xFFFF         halt
[0x0012]:0xABCD     jack: .word #0xABCD
[0x0013]:0x0000     jill: .word #0x0000
```

It is difficult to use this form of the instruction properly, if you add or remove anything between the lpc instruction and the desired memory location the constant will be wrong or you will have to keep adjusting it.

This is only a syntax difference for the lsa-as assembler, the lpc function is exactly the same. The the address used by the instruction is the current value of r0 + the constant. The current value of r0 is the address of the instruction plus one. In this case the lpc instruction is at address 0x000F, so the instruction computes the address 0x000F+1+0x03 which is 0x0013 which happens to be the address for the jill label.

Another way to look at it is illustrated below. The pc+n comment on the side is relative to the lpc instruction and can be used to figure out how to visualize looking ahead in the program. At the time that lpc executes r0 holds the address for the stw (the address for lpc + 1), so from the perspective of the lpc instruction the stw instruction is at the address pc+0. Halt, relative to lpc, is pc+1 and so on. To access the memory at label jill the lpc instruction needs to encode a 3 in the instruction.

```
    lpc r9,#0x3
    stw [r7],r9     ; pc+0
    halt            ; pc+1
jack: .word #0xABCD ; pc+2
jill: .word #0x0000 ; pc+3
```

You will likely never use the lpc or spc instructions in this form. The reasons for going through this exercise is to demonstrate that the lpc and spc instruction has eight unsigned bits available to add to the program counter (r0). This means that these instruction can only reach 0 - 255 instructions greater than the program counter (1 - 256 locations relative to the lpc or spc instruction). These instructions cannot reach addresses less than or equal to the address of the lpc or spc instruction.

Similar instructions used in other processors often use a signed constant. You can only reach half as far but you can reach forward and backward.

0.9 Lesson 8: Long reach and backward addressing

0.9.1 Code

```
---- lesson08 ----
    b lesson08
    b lesson08
lesson08:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    lpc r8,backward
    ldw r9,[r8]
    stw [r7],r8
    stw [r7],r9
    halt

backward: .word =lesson08
---- lesson08 ----
```

0.9.2 Output

```
show: 0x1234
show: 0x0002
show: 0x8534
```

0.9.3 Description

This program demonstrates how to overcome the fact that lpc only allows forward addressing. Using an equals sign (=) in front of a label name in a .word definition means to the assembler "the address of"

```
    backward: .word =lesson08
```

The value at the backward label will be the address of the label lesson08. The assembler, lsa-as, takes care of filling in these numbers for you. The output of lsa-as, lesson08.s.csv if written exactly as above will be:

```
MEM,0x0000,0xC001,END,,
MEM,0x0001,0xC000,END,,
MEM,0x0002,0x8534,END,,
MEM,0x0003,0xB512,END,,
MEM,0x0004,0x97FF,END,,
MEM,0x0005,0x4758,END,,
MEM,0x0006,0x0804,END,,
MEM,0x0007,0x4980,END,,
MEM,0x0008,0x4788,END,,
MEM,0x0009,0x4798,END,,
MEM,0x000A,0xFFFF,END,,
MEM,0x000B,0x0002,END,,
END,,
```

This lines up with the program in this manner:

```
MEM,0x0000,0xC001,END,,    b lesson08
MEM,0x0001,0xC000,END,,    b lesson08
                       lesson08:
MEM,0x0002,0x8534,END,,    llz r5,#0x34
MEM,0x0003,0xB512,END,,    lh  r5,#0x12
MEM,0x0004,0x97FF,END,,    lhz r7,#0xFF
MEM,0x0005,0x4758,END,,    stw [r7],r5
MEM,0x0006,0x0804,END,,    lpc r8,backward
MEM,0x0007,0x4980,END,,    ldw r9,[r8]
MEM,0x0008,0x4788,END,,    stw [r7],r8
MEM,0x0009,0x4798,END,,    stw [r7],r9
MEM,0x000A,0xFFFF,END,,    halt
MEM,0x000B,0x0002,END,, backward: .word =lesson08
END,,
```

The label lesson08 is at address 0x0002 in memory. Lsa-as was asked to reserve a word (.word) containing the address to the lesson08 label (=lesson08). The label backward shows up at address 0x000B, and it does contain the address for the lesson08 label, 0x0002.

The lpc r8,backward instruction loads the contents of memory at the label backward, 0x0002, into r8. The ldw r9,[r8] instruction uses the address 0x0002 and loads what is essentially the contents of memory at the lesson08 address into r9. It would be the same as doing this

```
  lpc r9,lesson08
```

But lesson08 is at an address smaller than the lpc instruction, it is backward in memory and not allowed. Lpc can only read from memory at addresses greater than the location where that lpc instruction lives. This lesson demonstrates how you get around that problem if you have a reason to reach a lower address.

Now you ask, why would the lsa processor not allow for loading from memory at greater than or less than addresses? That is the second part of this lesson. Eight bits of the lpc instruction are used to provide a constant to be added to the program counter to compute the address being read from. Most processors will use a twos complement number here giving -128 to +127 locations. The way the lsa processor implements this gives 0 to 255 locations. No matter how you implement this you are only able to reach a small percentage of memory away from the current instruction. Useful programs are going to have more than 255 instructions so you are going to have to manage variables or memory locations shared by various parts of that code.

Some instruction sets have a separate far load instruction which often encodes the whole address of what you are trying to load. In the above case this would have essentially been load r9,[lesson08], and the assembler for that kind of instruction set would have basically encoded it as load r9,[0x0002]. That requires a varaible word length instruction set.

For the instruction sets, like lsa, that do not have a far load, then you are required to use this two step, load the address from a location in memory then use that address to perform the load you were really interested in. Choosing to allow a short forward and backward reach or allowing a longer forward only is a design choice and in this case the choice was forward only. So the answer to the why not allow forward and backward loads using lpc is because this is a fixed word length instruction set, which means there is no way to have an immediate based far load instruction, -128 to +127 vs 0 to 255 would still require the code demonstrated in this example.

0.10 Lesson 9: ALU operations

0.10.1 Code

```
---- lesson09 ----
    b lesson09
    b lesson09
lesson09:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    llz r8,#0x01
    llz r9,#0x02
    add r9,r8
    stw [r7],r8
    stw [r7],r9

    llz r8,#0xEF
    dna r8,r9
    stw [r7],r8

    halt
---- lesson09 ----
```

0.10.2 Output

```
show: 0x1234
show: 0x0001
show: 0x0003
show: 0x00EC
```

0.10.3 Description

This demonstrates a couple of the several arithmetic logic unit (alu) operations. The first one is an add instruction which is defined as

```
    add rd,rs
```

and the function performed is rd = rd + rs. As used in this program r9 = r9 + r8.

The second operation is the dna instruction. The dna instruction is like an and instruction except the source bits are inverted before the and operation happens. The and, dna, or, xor, and tst instructions are logical, bitwise, operations. Which means bit 0 of one operand and bit 0 of the other operand are computed to create bit 0 of the result. The truth table for the logical and operation and logical dna operation are shown to understand how the dna operation compares to an and operation.

```
a b and dna
0 0  0   0
0 1  0   0
1 0  0   1
1 1  1   0
```

The columns a and b represent the two input bits. With an and operation any bit in the rd register anded with a zero in the rs register results in a zero. Any bit in the rd register anded with a one in the rs register results in itself. With a dna operation any bit in the rd register dnaed with a zero in the rs register is itself. Any bit in the rd register dnaed with a one in the rs register is a zero. Both the dna instruction and and instruction are used to zero bits in the rd register, the difference is you might save an instruction by using one operation instead of the other. For example if we want to zero all but the lower three bits of a register we can do this:

```
    llz r2,#0x07
    dna r3,r2
```

or this

```
    lhz r2,#0xFF
    ll  r2,#0xF8
    and r3,r2
```

or this

```
    llz r2,#0x07
    not r2,r2
    and r3,r2
```

if the number of bits you want to zero is more than eight and you need to use a load immediate to specify those bits, then use the dna instruction, if the number of bits you want to zero is less than eight and you need to use an immediate instruction then use the and instruction.

There are quite a few different alu instructions in the lsa processor. The full list and their functions are shown here:

```
add rd,rs      rd = rd + rs
sub rd,rs      rd = rd - rs
and rd,rs      rd = rd & rs
dna rd,rs      rd = rd & (~rs)
or  rd,rs      rd = rd | rs
```

```
xor rd,rs      rd = rd ^ rs
neg rd,rs      rd =  0 - rs
not rd,rs      rd =  ~ rs
inc rd,rs      rd = rs +  1
dec rd,rs      rd = rs -  1
cmp rd,rs         = rd - rs
tst rd,rs         = rd & rs
```

Note that cmp (compare) and tst (test) instructions do not modify the destination register. They only update the flags, which we will get to in a few lessons.

0.11 Lesson 10: Unconditional branch

0.11.1 Code

```
---- lesson10 ----
    b lesson10
    b lesson10
lesson10:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    llz r6,#0x00
    inc r6,r6
    inc r6,r6
    inc r6,r6
    b lab01
    inc r6,r6
    inc r6,r6
lab01:
    inc r6,r6
    inc r6,r6
    inc r6,r6
    stw [r7],r6

    halt
---- lesson10 ----
```

0.11.2 Output

```
show: 0x1234
show: 0x0006
```

0.11.3 Description

This program demonstrates the unconditional branch instruction (b). Unconditional just as the name implies, just do it. A branch is also known as a jump in other instruction sets or a goto in C. We have actually, obviously, been using the unconditional branch instruction in the vector table in each of these lessons. This lesson formally introduces the branch instruction as well as demonstrates that it can be used anywhere, not just in the vector table.

There are eight increment instructions in this program, but two are branched over and not executed, so only six of them are executed causing r6 to end up as a 0x0006 when we show it.

This processor does not have any "pipeline" rules that force the execution of instructions after a branch. Pipelining is an interesting and important topic to understand but beyond the scope of this document.

0.12 Lesson 11: Two simple flags

0.12.1 Code

```
---- lesson11 ----
    b lesson11
    b lesson11
lesson11:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    lhz r6,#0xFF
    ll  r6,#0xFE
    stw [r7],r6
    inc r6,r6
    stw [r7],r1
    stw [r7],r6
    inc r6,r6
    stw [r7],r1
    stw [r7],r6
    dec r6,r6
    stw [r7],r1
    stw [r7],r6
    dec r6,r6
    stw [r7],r1
    stw [r7],r6

    stw [r7],r5

    halt
---- lesson11 ----
```

0.12.2 Output

```
show: 0x1234
show: 0xFFFE
show: 0x0004
show: 0xFFFF
show: 0x0003
show: 0x0000
show: 0x0004
show: 0xFFFF
show: 0x0006
show: 0xFFFE
show: 0x1234
```

0.12.3 Description

Like the program counter, r0, there are other registers that are not so general purpose. r1, the status register (sometimes called the program status register psr), is one of these registers. The lower four bits of r1 are special, they hold status flags related to alu operations. The other 12 bits are reserved for future use.

The four status flags are:

```
r1_bit_3 v signed overflow
r1_bit_2 n negative
r1_bit_1 c carry (unsigned overflow)
r1_bit_0 z zero
```

Overflow flags (v, c) are a little more complicated so we will talk about them later. The n and z flags are relatively simple though.

The z flag represents a zero result. If the result of the alu operation is 0x0000 then the z bit will be set (one). If the result of the alu operation is not 0x0000 then the z bit will be clear (zero).

The n flag is even simpler, the n flag gets a copy of bit 15 of the result. If bit 15 of the result is set (one) then the n flag will be set (one). If bit 15 of the result is clear (zero) then the n flag will be clear (zero). The n flag is useful for twos complement numbers when wanting to see if a result was negative or positive. Your prior programming experience should have exposed you to twos complement numbers. Describing them is beyond the scope of this document.

Since the n bit has a connection to twos complement numbers it might make more sense to only update it for math operations that might use twos complement numbers like the add instruction for example. The reality is that the lsa processor copies bit 15 of the result to the n bit for all alu operations. Likewise the z flag is computed for all alu operations.

The program for this lesson shows that 0xFFFE + 0x0001 results in 0xFFFF. 0xFFFF is not zero and has bit 15 set, so the n flag is one and the z flag is zero. Things change with 0xFFFF + 0x0001, the result is 0x0000, the z flag is set and the n flag is not.

0.13 Lesson 12: Our first conditional branch

0.13.1 Code

```
---- lesson12 ----
    b lesson12
    b lesson12
lesson12:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    llz r6,#0x04
lab01:
    stw [r7],r6
    dec r6,r6
    bz lab02
    b lab01
lab02:

    stw [r7],r5

    llz r6,#0x04
lab03:
    stw [r7],r6
    dec r6,r6
    bnz lab03

    stw [r7],r5

    halt
---- lesson12 ----
```

0.13.2 Output

```
show: 0x1234
show: 0x0004
show: 0x0003
show: 0x0002
show: 0x0001
show: 0x1234
show: 0x0004
show: 0x0003
show: 0x0002
show: 0x0001
show: 0x1234
```

0.13.3 Description

We learned what an unconditional branch (b) is and what the z flag represents. Now we get to see what the status flags/bits are used for. This program loads the value 0x0004 into r6. We show r6, decrement it to 0x0003 then execute the new instruction bz, which means branch if zero (branch if the z bit is set at the time we execute this instruction). We know that dec is an alu operation and the result of that operation modifies the status register. If the result of the dec is a 0x0000 then the z bit is set, otherwise it is not. So for the case where the result is 0x0003 the z flag is not set, and the branch does not happen. What that means is you do not branch to the address specified you simply continue on to the next instruction in memory. So for this case of the deci instruction resulting in 0x0003 we execute the next instruction which is an unconditional branch to lab01.

The unconditioal branch takes us back to show r6, and the decrement again. This show and decrement loop continues until the dec instruction results in 0x0000. At that time the z bit is set and the bz branch to lab02 happens which takes us to an stw that shows r5 (0x1234). We can see in the debug output below that the last show before showing r5 is a 0x0001.

```
show: 0x0004
show: 0x0003
show: 0x0002
show: 0x0001
show: 0x1234
```

So r6 was a 0x0001 and the dec instruction made it a 0x0000, the z bit was set, the bz happened and 0x1234 was the next thing we see.

Using a different instruction we can make that loop more efficient. The bnz instruction means branch if not zero (branch if the zero flag is not set).

As with the first loop we start with r6 = 0x0004. The dec instruction decrements r6 to 0x0003. 0x0003 is not 0x0000 so the z flag is not set after the dec instruction. The bnz instruction says to branch if the z flag is not set, so it does, back to the to of the loop. As with the other loop this continues until the dec instruction results in a zero, leaving the z bit set. When the bnz instruction executes with the z bit set it does not branch, execution continues with the stw instruction after the bnz instruction.

The full list of branch instructions are listed below. Future lessons will get into the c and v bits in more detail.

```
b   unconditional branch
bz  branch if zero (z = 1)
bnz branch if not zero (z = 0)
bc  branch if carry (c = 1) (unsigned greater or equal)
bnc branch of not carry (c = 0) (unsigned less)
bn  branch if negative (n = 1)
bnn branch if not negative (n = 0)
bv  branch if signed overflow (v = 1)
bnv branch if not signed overflow (v = 0)
bsg signed greater than or equal (n .xor. v) = 0  (n == v)
bsl signed less than             (n .xor. v) = 1  (n != v)
```

0.14 Lesson 13: Another conditional branch

0.14.1 Code

```
---- lesson13 ----
    b lesson13
    b lesson13
lesson13:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    llz r6,#0x04
lab01:
    stw [r7],r6
    dec r6,r6
    bnn lab01

    stw [r7],r5

    halt
---- lesson13 ----
```

0.14.2 Output

```
show: 0x1234
show: 0x0004
show: 0x0003
show: 0x0002
show: 0x0001
show: 0x0000
show: 0x1234
```

0.14.3 Description

This program uses the n flag for what it is, bit 15 of the result. Very similar to the prior lesson where we were counting down to zero this one goes one more time through the loop because from 0x0004 down to 0x0000 bit 15 is not set, when we decrement from 0x0000 to 0xFFFF (a negative one). We get bit 15 set and the bnn (branch if not negative, branch if n is not set) does not happen because the n bit is set.

0.15 Lesson 14: Carry bit (unsigned overflow)

0.15.1 Code

```
---- lesson14 ----
    b lesson14
    b lesson14
lesson14:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5
    llz r6,#0xFE
    lh  r6,#0xFF
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5
    inc r6,r6
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5
    inc r6,r6
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5
    inc r6,r6
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5
    dec r6,r6
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5
    dec r6,r6
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5
    dec r6,r6
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5
    dec r6,r6
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5

    halt
---- lesson14 ----
```

0.15.2 Output

```
show: 0x1234
show: 0xFFFE
show: 0x0000
show: 0x1234
show: 0xFFFF
show: 0x0004
show: 0x1234
show: 0x0000
show: 0x0003
show: 0x1234
show: 0x0001
show: 0x0000
show: 0x1234
show: 0x0000
show: 0x0003
show: 0x1234
show: 0xFFFF
show: 0x0004
show: 0x1234
show: 0xFFFE
```

```
show: 0x0006
show: 0x1234
show: 0xFFFD
show: 0x0006
show: 0x1234
```

0.15.3 Description

Think about elementary school and you were learning to add. And you would line up numbers like 97 + 5. Add the ones column you get "two carry the one". Then the tens column "zero carry the one" and that last one drops down for the hundreds column. Well think about those carry bits taking you from one column to the next. Unlike grade school imagine if there was a rule about the number of digits you had available to store the result. If the rule was two digits and you tried to add 97 + 5 the result would be 04 but you would also have a carry bit in the status register indicating there was not enough room to store the complete answer in the register.

Look at the program above, when we add 0x0001 to 0xFFFE we get 0xFFFF, and the carry bit is zero. Next 0xFFFF + 0x0001 is really 0x10000 but the registers are 16 bit so it is 0x0000 with the carry bit set. 0x0000 + 0x0001 is 0x0001 no problem, no carry bit.

The first decrement is a bit tricky and perhaps confusing. Subtracting one is the same as adding a minus one. Since processors normally use twos complement numbers, negating a number or "taking the twos complement" of the number means invert and add one. So 0x0001 - 0x0001 is the same as 0x0001 + 0xFFFE + 1, and the result of that would normally be 0x10000 but we only have 16 bits so the result is 0x0000 with the carry flag set. We know that 1 - 1 = 0, it is interesting that the carry flag is set. When we add the carry flag being set means we overflowed the (for unsigned numbers), when we subtract, the carry flag being set means we did not overflow the register (for unsigned numbers). The next dec instruction takes the 0x0000 in the register and decrements. 0x0000 - 0x0001 is the same as 0x0000 + 0xFFFE + 1 which is 0xFFFF with the carry bit clear. 0xFFFF decrements to 0xFFFE with no carry flag and so on.

Just like that boundary between 99 and 00 in a two digit decimal world is where we would see the carry bit or digit in that case want to go to the hundreds column. In binary the boundary between the largest unsigned number 0xFFFF and the smallest unsigned number 0x0000 is where we will see the carry flag show some activity. Not limited to the inc instruction, an add instruction or the other math instructions that dance around this boundary will cause changes in the carry flag. Bitwise logical operations like the dna instruction or xor instruction zero the carry flag because there is not much else they can do with it.

0.16 Lesson 15: Signed overflow bit (v)

0.16.1 Code

```
---- lesson15 ----
    b lesson15
    b lesson15
lesson15:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    llz r6,#0xFE
    lh  r6,#0x7F
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5
    inc r6,r6
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5
    inc r6,r6
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5
    inc r6,r6
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5
    dec r6,r6
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5
    dec r6,r6
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5
    dec r6,r6
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5
    dec r6,r6
    stw [r7],r6
    stw [r7],r1
    stw [r7],r5

    halt
---- lesson15 ----
```

0.16.2 Output

```
show: 0x1234
show: 0x7FFE
show: 0x0000
show: 0x1234
show: 0x7FFF
show: 0x0000
show: 0x1234
show: 0x8000
show: 0x000C
show: 0x1234
show: 0x8001
show: 0x0004
show: 0x1234
show: 0x8000
show: 0x0006
show: 0x1234
show: 0x7FFF
show: 0x000A
show: 0x1234
show: 0x7FFE
show: 0x0002
show: 0x1234
show: 0x7FFD
show: 0x0002
show: 0x1234
```

0.16.3 Description

Just like the carry bit indicated unsigned addition (or subtraction) had overflowed the storage capabilities of a register, the v bit, the signed overflow indicates that a math operation overflowed. The c flag is used when the numbers are considered to be unsigned and the v flag is used when the numbers are considered to be signed.

Considered to be signed, what does that mean? Binary addition and subtraction logic does not care about signed numbers vs unsigned numbers. That is the beauty of twos complement, it makes it so the math just works. (from a hardware perspective).

Think about 0x0000 + 0xFFFF, that can be thought of as 0 plus 65535 or it could be thought of as 0 plus -1. And in both cases the result is 0xFFFF, which can be thought of as 65535 if the numbers are considered unsigned or -1 if the numbers are considered signed. The hardware takes the same operands 0x0000 and 0xFFFF and generates the same result 0xFFFF, without caring if these are signed or unsigned numbers.

So this program starts with 0x7FFE which is a 32766, we add one we get 0x7FFF (32767). Add 0x7FFF + 0x0001 we get 0x8000 which is a -32768, note the sign changed, and note that the v bit was set on this one, we don't have the ability to store a twos complement +32768 (+32767 + 0x0001) in 16 bits so we had a signed overflow.

0x8000 + 0x0001 gives 0x8001 which is a -32767, no problem storing that one so the v bit is zero. Now we go back the other way 0x8001 - 0x0001 is 0x8000 or -32768 which we can store, then -32768 - 1 is ideally -32769 but we don't have room to store that so we get 0x8000 - 0x0001 = 0x7FFF which is +32767 and the v flag is set. +32767 - 1 = +32766 and we have room for that and so on.

Just like the carry bit dealt with the boundary between the largest unsigned number (0xFFFF) and the smallest (0x0000), the v bit deals with that boundary between the largest signed number (0x7FFF) and the smallest signed number (0x8000).

As with the z and n bits there are conditional branch instructions for the c and v bits being 0 or 1 (bc, bnc, bv, bnv).

0.17 Lesson 16: Unsigned and signed conditional branches

0.17.1 Code

```
---- lesson16 ----
    b lesson16
    b lesson16
lesson16:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    llz r10,#0x06
    llz r11,#0x05
    cmp r10,r11
    stw [r7],r1

    dec r10,r10
    cmp r10,r11
    stw [r7],r1

    dec r10,r10
    cmp r10,r11
    stw [r7],r1

    lhz r10,#0x80
    lhz r11,#0x7F
    ll  r11,#0xFF
    cmp r10,r11
    stw [r7],r1

    dec r10,r10
    cmp r10,r11
    stw [r7],r1

    dec r10,r10
    cmp r10,r11
    stw [r7],r1

    llz r10,#0x06
    lhz r11,#0xFF
    ll  r11,#0xFB
    cmp r10,r11
    stw [r7],r1

    cmp r11,r10
    stw [r7],r1

    cmp r10,r10
    stw [r7],r1

    cmp r11,r11
    stw [r7],r1

    halt
---- lesson16 ----
```

0.17.2 Output

```
show: 0x1234
show: 0x0002
show: 0x0003
show: 0x0004
show: 0x000A
show: 0x0003
show: 0x0004
show: 0x0000
show: 0x0006
show: 0x0003
show: 0x0003
```

0.17.3 Description

First the program leads up to a cmp (compare) instruction. Which is identical to a subtract except the result is discarded and the rd register unmodified. Like a sub instruction it updates the flags.

```
    sub r10,r11
```

Computes r10 - r11 and stores the result in r10.

```
    cmp r10,r11
```

Computes r10 - r11 but discards the mathematical result.

This program performs six comparisons. The goal here is to see if the conditional branch instructions make sense. Branch if unsigned greater than or equal, signed less than, etc.

Thinking in terms of unsigned numbers:

```
0x0006 - 0x0005 v = 0, n = 0, c = 1, and z = 0  6 - 5
0x0005 - 0x0005 v = 0, n = 0, c = 1, and z = 1  5 - 5
0x0004 - 0x0005 v = 0, n = 1, c = 0, and z = 0  4 - 5
```

```
0x8000 - 0x7FFF v = 1, n = 0, c = 1, and z = 0  32768 - 32767
0x7FFF - 0x7FFF v = 0, n = 0, c = 1, and z = 1  32767 - 32767
0x7FFE - 0x7FFF v = 0, n = 1, c = 0, and z = 0  32766 - 32767
```

```
0x0006 - 0xFFFB v = 0, n = 0, c = 0, and z = 0  6 - 65531
0xFFFB - 0x0006 v = 0, n = 1, c = 1, and z = 0  65531 - 6
0x0006 - 0x0006 v = 0, n = 0, c = 1, and z = 1  6 - 6
0xFFFB - 0xFFFB v = 0, n = 0, c = 1, and z = 1  65531 - 65531
```

The conditional branch, bc, branch if carry also says that it means branch if unsigned greater than or equal. We see above that the c bit is set if rd >= rs. It so far matches the claim. This also means the bnc, branch if not carry, comment about branch if unsigned less than, is also true, if rd < rs the carry bit is clear for unsigned numbers.

Where is the branch if unsigned greater (but not equal) instruction? You dont need one, if

```
    cmp r2,r3
    bnc somewhere
```

branches for an unsigned r2<r3. And we want to figure out how to do an unsigned greater than but not equal to branch. That means we want to know what to do when r2>r3. Well we know how to handle r3<r2 right isnt that the same thing?

```
    cmp r3,r2
    bnc somewhere
```

branches for an unsigned r3<r2, r2 is greater than but not equal to r3.

Back to the program but thinking in terms of signed numbers

```
0x0006 - 0x0005 v = 0, n = 0, c = 1, and z = 0  6 - 5
0x0005 - 0x0005 v = 0, n = 0, c = 1, and z = 1  5 - 5
0x0004 - 0x0005 v = 0, n = 1, c = 0, and z = 0  4 - 5
```

```
0x8000 - 0x7FFF v = 1, n = 0, c = 1, and z = 0  -32768 - 32767
0x7FFF - 0x7FFF v = 0, n = 0, c = 1, and z = 1   32767 - 32767
0x7FFE - 0x7FFF v = 0, n = 1, c = 0, and z = 0   32766 - 32767
```

```
0x0006 - 0xFFFB v = 0, n = 0, c = 0, and z = 0  6 - -5
0xFFFB - 0x0006 v = 0, n = 1, c = 1, and z = 0  -5 - 6
0x0006 - 0x0006 v = 0, n = 0, c = 1, and z = 1  6 - 6
0xFFFB - 0xFFFB v = 0, n = 0, c = 1, and z = 1  -5 - -5
```

These two instructions are in the list of conditional branches

```
bsg signed greater than or equal (n .xor. v) = 0  (n == v)
bsl signed less than             (n .xor. v) = 1  (n != v)
```

Does that match what we are seeing? When n == v

```
0x0006 - 0x0005 v = 0, n = 0, c = 1, and z = 0   6 - 5
0x0005 - 0x0005 v = 0, n = 0, c = 1, and z = 1   5 - 5
0x7FFF - 0x7FFF v = 0, n = 0, c = 1, and z = 1   32767 - 32767
0x0006 - 0xFFFB v = 0, n = 0, c = 0, and z = 0   6 - -5
0x0006 - 0x0006 v = 0, n = 0, c = 1, and z = 1   6 - 6
0xFFFB - 0xFFFB v = 0, n = 0, c = 1, and z = 1  -5 - -5
```

rd is a signed greater than or equal to rs. And when n and v do not match:

```
0x0004 - 0x0005 v = 0, n = 1, c = 0, and z = 0   4 - 5
0x8000 - 0x7FFF v = 1, n = 0, c = 1, and z = 0  -32768 - 32767
0x7FFE - 0x7FFF v = 0, n = 1, c = 0, and z = 0   32766 - 32767
0xFFFB - 0x0006 v = 0, n = 1, c = 1, and z = 0  -5 - 6
```

rd is signed less than rs.

Just like with unsigned numbers, you do not need a signed greater than but not equal to instruction. bsl branches when rd<rs which can be interpreted as rd is less than rs or rs is greater than (but not equal to) rd.

0.18 Lesson 17: Register based branching

0.18.1 Code

```
---- lesson17 ----
    b lesson17
    b lesson17
lesson17:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    lpc r8,lab02add
    stw [r7],r8
    llz r6,#0x02
lab01:
    stw [r7],r6
    dec r6,r6
    bz r8
    b lab01
lab02:
    stw [r7],r5
    halt
lab02add: .word =lab02
---- lesson17 ----
```

0.18.2 Output

```
show: 0x1234
show: 0x000D
show: 0x0002
show: 0x0001
show: 0x1234
```

0.18.3 Description

Up until now the branch instructions we have been using were pc relative. The destination address (if the branch occurs) is computed using the current value of the pc plus a signed constant encoded in the instruction.

This is similar to the lpc and spc instructions except the branch instructions can branch forward or backward in the program. But their range is limited.

The solution for lpc or spc's limited reach was a two step approach lpc for example would be used to load something it was able to reach but that was actualy an address than an ldw used to perform a longer reach.

The branch solution to the limited reach is basically the same, an lpc instruction is used to load an address into a register, the branch instruction uses that register as an operand instead of a label or constant.

To implement this we have a loop that counts down, when it gets to zero we want to branch out of the loop with a bz. In order to use a register branch in this way, the registe bz is using must have had the desitnation register loaded into it. These lines in the program result in

```
    lpc r8,lab02add
    ...
lab02add: .word =lab02
```

r8 containing the address to lab02. If and when this branch

```
    bz r8
```

occurs the destination will be lab02.

0.19 Lesson 18: Branch table

0.19.1 Code

```
---- lesson18 ----
    b lesson18
    b lesson18
lesson18:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    llz r8,#0x02

    llz r11,#0x03
    and r8,r11
    llz r10,#0x00
    lpc r6,branch_table_add
    add r6,r8
    ldw r9,[r6]
    b r9
    halt

lab00:
    llz r10,#0x30
    b done
lab01:
    llz r10,#0x31
    b done
lab02:
    llz r10,#0x32
    b done
lab03:
    llz r10,#0x33
    b done

done:
    stw [r7],r10
    halt

branch_table_add: .word =branch_table
branch_table:
    .word =lab00
    .word =lab01
    .word =lab02
    .word =lab03
---- lesson18 ----
```

0.19.2 Output

```
show: 0x1234
show: 0x0032
```

0.19.3 Description

This is an example of how you would implement a branch or jump table. Say for example you had a small, simple switch statement in C, you might implement it using a jump table. (provided the number of cases are small enough to be practical and the case values are such that you can use simple math to uniquely identify each item).

This particular example assumes that r8 is one of the values 0, 1, 2, or 3. We hard code r8 to a 0x02, but let's pretend we didn't know its value. To make sure that r8 is limited to that range we use an and instruction with a 3.

Next we get the address to the beginning of the table in r6, add r8 to r6 so that it holds the address of the item in the table we are interested in. The load loads that item into r9. And then the branch using r9 branches to that address. If you have made no programming mistakes you cannot get to the halt after b r9.

Each of the four addresses contain code specific to that r8 input condition. That code is executed and converges on a common point to continue.

This method of using a table of items and indexing into them is also called a look up table, or more simply an array. In this case our array is an array of 4 addresses and we have looked up the address at index 2 and then used it.

0.20 Lesson 19: Move and swap

0.20.1 Code

```
---- lesson19 ----
    b lesson19
    b lesson19
lesson19:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    llz r10,#5
    llz r11,#6
    swap r10,r11
    stw [r7],r10
    stw [r7],r11
    mov r10,r11
    stw [r7],r10
    stw [r7],r11
    halt
---- lesson19 ----
```

0.20.2 Output

```
show: 0x1234
show: 0x0006
show: 0x0005
show: 0x0005
show: 0x0005
```

0.20.3 Description

The swap instruction swaps the contents of the two registers. Swap is not an alu operation so it does not modify the flags register (unless of course r1 is one of the registers).

A mov instruction simply moves (well actually it copies) the source contents to the destination. Unlike the swap instruction, the source register is not modified.

There is a variant of the lsa processor that has up to 128 general purpose registers. All instructions except for the mov instruction are limited to register r0 - r15. These extra registers r16 on up are called the high registers. These can be thought of as extra, easy to get at memory. Normally when you access memory you have to use a register and put an address in it and then perform a memory operation, with these registers you can do all of that with a single mov. Due to the fixed instruction set length nature of the lsa processor one of the two registers has to be a low register or r0 - r15, the other can be any (supported) register. It is better to assume that only r0 - r15 have been implemented. The hardware engineer can choose to have 32 or 64 up to 128 registers, so if there are high registers you need to find out exactly how many before you use them.

Many processors use the mov instruction for moving from register to register, immediate to register, memory to register, register to memory, etc. And the syntax like brackets or pound signs, etc tell the assembler what flavor of mov instruction was desired. The lsa processor has separate instructions for loading immediates, loading and storing words from and to memory and copying the contents of one register to another.

0.21 Lesson 20: Shifts

0.21.1 Code

```
---- lesson20 ----
    b lesson20
    b lesson20
lesson20:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    llz r6,#0x88
    lsl r6,#1
    stw [r7],r6
    lsl r6,#3
    stw [r7],r6

    lhz r6,#0xFF
    lsl r6,#4
    stw [r7],r6

    lsr r6,#2
    stw [r7],r6

    lhz r6,#0x90
    asr r6,#2
    stw [r7],r6

    lsr r6,r5
    stw [r7],r6

    halt
---- lesson20 ----
```

0.21.2 Output

```
show: 0x1234
show: 0x0110
show: 0x0880
show: 0xF000
show: 0x3C00
show: 0xE400
show: 0x0E40
```

0.21.3 Description

The shift instructions, lsl, lsr, asr, are used to perform bitwise shifts. Very similar to the << and >> operations in C.

The program starts by putting 0x0088 in r6. Then it performs a logical shift left (lsl) of 1 bit.

We started with

```
0000000010001000
```

We discard the top bit, shift everything left one and shift in a zero as the new bit on the right

```
0000000010001000
 000000010001000
000000010001000
0000000100010000
```

Giving 0x0110.

Some processors let you only shift one bit left or one bit right per instruction. The lsa processor allows you to shift as many as 15 bits in one instruction. The next shift we perform is also a left shift, lsl, this time 3 bits.

r6 starts with

```
0000000100010000
```

Discard three off the top and shift in three zeros on the right and you get.

```
0000000100010000 started with this
   0000100010000 discard the top three bits
0000100010000    shift three bits left
0000100010000000 the new bits are zeros
```

So the result is 0x0880.

This three bit lsl is the same as this C code:

```
unsigned short rx;
rx = 0x0110;
rx <<= 3;
```

Next we put the value 0xFF00 in r6 and perform a 4 bit shift

```
1111111100000000
```

Discard the top 4 bits, shift left 4, shifting in zeros from the right gives

```
1111111100000000
    111100000000
111100000000
1111000000000000
```

or 0xF000.

Now we shift right, lsr, logical shift right. Same as lsl but in the other direction. We left off with r6 = 0xF000 and want to shift right 2 bits.

```
1111000000000000
```

discard the lower 2 bits, shift right 2, put zeros in on the left

```
1111000000000000
11110000000000
  11110000000000
0011110000000000
```

or 0x3C00

Now we get into the arithmetic shift right, asr. This is one you use when you have a signed number as it sign extends or otherwise preserves the sign bit. We start with r6 = 0x9000 and want to shift right arithmetically 2 bits. With a logical shift we arbitrarily shift zeros in to fill in the new bits, with an arithmetic shift we use bit 15 as the value we shift in.

So looking at the shifts one bit at a time, discard the lower bit, shift right one, instead of shifting in a zero shift in whatever r15 was before the shift.

```
1001000000000000 start here
100100000000000  discard lower bit
 100100000000000 shift right one
1100100000000000 take r15 from before (now r14) and shift that in
```

The second shift would then end up with

```
1110010000000000
```

giving 0xE400. If bit 15 had been a zero instead, then zeros would have shifted in from the top. This instruction is useful when you want to, as we did in the example, divide a signed number by 4. Using the C style logical shift right this would not have worked for a signed number. Shifting in a zero from the top would result in a positive number. If the signed number were positive you would have been okay.

The last example shows a logical shift right using a register as the source. The lower four bits of the source register determine how much to shift, in this case r6 = 0xE400 and r5 = 0x1234, so the lower four bits of r5 are 0x4 which means we want to shift r6 right 4 bits. This is a logical shift so zeros are shifted in from the top and the result is 0x0E40.

The shift instructions (lsl, lsr, asr) do not modify the flags.

0.22 Lesson 21: Rotates

0.22.1 Code

```
---- lesson21 ----
    b lesson21
    b lesson21
lesson21:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    llz r6,#0x88
    rol r6,#1
    stw [r7],r6
    ror r6,#5
    stw [r7],r6

    lhz r6,#0xFF
    rol r6,#4
    stw [r7],r6

    ror r6,#2
    stw [r7],r6

    lhz r6,#0x90
    llz r1,#0x00
    rlc r6,r5
    stw [r7],r6
    stw [r7],r1

    rrc r6,#1
    stw [r7],r6

    halt
---- lesson21 ----
```

0.22.2 Output

```
show: 0x1234
show: 0x0110
show: 0x8008
show: 0xF00F
show: 0xFC03
show: 0x0004
show: 0x0002
show: 0x8002
```

0.22.3 Description

Rotates are very similar to shifts, the difference is that with a shift the bits that are shifted off, are gone forever (off into the bit bucket or into the ether). With a rotate you preserve all of the bits, bits shifted off of one end of the register find their way into the other end of the register.

We start with r6 = 0x0088 and rotate left (rol) 1 bit. The rol and ror instructions take the bits off of one end and directly roll them into the other end. Start with

```
 0000000010001000
```

rol 1 bit so separate one bit off the top

```
0 000000010001000
```

and stick it on the bottom

```
000000010001000 0
```

becomes

```
0000000100010000
```

Or 0x0110.

Now we want to rotate right, ror that number 5 bits

```
0000000100010000
```

separate 5 bits on the right

```
00000001000 10000
```

And put them on the top

```
10000 00000001000
```

becomes

```
1000000000001000
```

Which is 0x8008.

Next start with 0xFF00 and rol 4 bits.

```
1111111100000000
```

take 4 off the left and put them on the right.

```
1111 111100000000
111100000000 1111
1111000000001111
```

Which is 0xF00F.

Now ror 2 bits.

```
1111000000001111
11110000000011 11
11 11110000000011
1111110000000011
```

Which is 0xFC03

Now there are two more flavors of rotate, rrc and rlc (rotate right through carry, or rotate left through carry). These instructions use the carry bit in the rotation. So for example with a ror, for each shift the bit that was in bit 0 of the register goes into bit 15 of the register. With a rrc, the bit in bit 0 of the register goes into the carry bit in r1, what was in the carry bit in r1 goes into bit 15 of the register.

As with the shifts, the rotates can use either an immediate value in the instruction or a register, and as with the shifts the lower 4 bits of that register are the shift amount, the upper 12 bits of the value in that register are ignored.

Starting with a 0x9000 rotate left through carry r5 bits, r5 contains 0x1234 so the rlc is 4 bits. Before we do this though there is a load of r1, looks a bit scary to be messing with r1, but what that is doing is making the carry bit a known state, otherwise we start to rotate and whatever was left in the carry bit gets dragged into our register, we probably don't want that. An and r6,r6 (or any register with itself) would have done the same thing as an and instruction clears the c and v bits no matter what the data is.

We can think of this two ways, think about shifting one bit at a time until we get the number of shifts, or think about chopping off the total amount of bits, one of them being left behind as the carry and another picking up the old carry bit.

Going to draw the carry bit out on the left by itself

```
0 1001000000000000
1 0010000000000000 first bit shift through carry
0 0100000000000001 shift 2
0 1000000000000010 shift 3
1 0000000000000100 shift 4
```

result is 0x0004 with a one in the carry bit which we see with a show of 0x0002.

The last thing is rrc 1 bit, remember for rrc bit 0 goes into the carry bit and the carry bit goes into bit 15.

```
1 0000000000000100 we were left with this
0 1000000000000010 shift 1
```

And the result is 0x8002.

The rol and ror instructions do not modify any flags. The rrc and rlc modify the c flag but leave the other flags untouched.

0.23 Lesson 22: The Stack

0.23.1 Code

```
---- lesson22 ----
    b lesson22
    b lesson22
lesson22:
    llz r5,#0x34
    lh  r5,#0x12
    lhz r7,#0xFF
    stw [r7],r5

    lhz r2,#0x10

    llz r6,#0x01
    push r6
    inc r6,r6
    push r6
    inc r6,r6
    stw [--r2],r6

    lsp r8,#0
    stw [r7],r8
    lsp r8,#1
    stw [r7],r8
    lsp r8,#2
    stw [r7],r8
    pop r8
    stw [r7],r8
    lsp r8,#0
    stw [r7],r8
    ldw r8,[r2++]
    stw [r7],r8
    ldw r8,[r2++]
    stw [r7],r8

    halt
---- lesson22 ----
```

0.23.2 Output

```
show: 0x1234
show: 0x0003
show: 0x0002
show: 0x0001
show: 0x0003
show: 0x0002
show: 0x0002
show: 0x0001
```

0.23.3 Description

The last of the three not so general purpose, general purpose registers, r2 is also known as the stack pointer. You may have heard of the term stack from prior programming experience. Putting things on the stack, pushing things on the stack, popping them off. Local variables live on the stack, that sort of thing.

The stack is just a name for some of the memory pointed to by the stack pointer. Most processors have special push and pop instructions that have the stack pointer hardcoded as part of the function of the instruction. The lsa processor does not need to have instructions like that as the normal instructions provide that functionality, the lsa processor instead has two special instructions (lsp, ssp) that have r2 hardcoded in the logic. The lsp and ssp instructions are similar to lpc and spc except that the stack pointer is used as the base address instead of the program counter.

Maybe you are not old enough to remember a time when almost everyone's bathroom had a dixie cup dispenser. Well imagine kid sized disposable paper cups (or plastic if it helps your visualization). And a plastic cylinder shaped holder that holds the cups leaving one cup visible sticking out the bottom. If you pull that bottom cup out you get one cup and unless that was the last one the holder has the next cup sticking out the bottom ready to be pulled out. Now imagine those cups are memory locations in your stack. If you want to save something temporarily, something that, for various reasons, does not need or want a permanent address. Let's say you have an operation that needs 7 registers but you have used all but 5 and need to temporarily save the contents of two registers so you can have 7 total available for this operation. Pushing two registers "on the stack" would be like writing the of the first register on a cup and pushing that cup up into the dixie cup holder. Then writing the value of the second register on another cup and pushing that cup into the dixie cup holder, now you can use the 7 registers for your operation, when you are finished and need to restore those two registers you borrowed, you can put those two registers back by popping the bottom cup (pulling it out of the holder) and looking at its contents and using that to restore the SECOND register, the last thing you put on the stack. Then pull (pop) what is now the bottom cup and put that in the FIRST register. You basically borrowed two memory locations temporarily and the key is that you didn't know, nor did you care how many cups were in the Dixie cup dispenser when you started, you pushed two cups up into the dispenser, then you removed two cups. You left it the way you found it. What you did care about is the number of items pushed and the order you pushed them. So that you can undo everything in reverse order.

Hopefully that was not too confusing of a visualization. To be more specific, a "push", on the lsa processor is a stw [--r2],rs, the stack pointer is decremented one word location, and the desired register is stored at that address. A "pop" on the lsa processor is an ldw rd,[r2++], read the value at the address pointed to by r2, then increment r2 one word location, save the value read in the destination register.

Assume r2 at the moment is at address X (other than initialization at boot time we don't normally care exactly what the stack pointer value is) and we do this:

```
stw [--r2],r6   ; push, store r6 at address X-1
stw [--r2],r7   ; push, store r7 at address X-2
llz r6,#0x55
llz r7,#0x66
ldw r7,[r2++]   ; pop, restore r7 from address X-2
ldw r6,[r2++]   ; pop, restore r6 from address X-1
```

After pushing r6 and r7 on the stack we can use them at will and not worry about whatever important information they were holding for us. When done using them we put them back the way we found them, same number of pops as pushes and in the reverse order. When done r2, r6, and r7 are back the way we found them.

The lsa-as assembler has a nice labor saving feature, instead of typing:

```
stw [--r2],r6
```

We can take a shortcut and type

```
push r6
```

Likewise instead of typing

```
ldw r6,[r2++]
```

We can type

```
pop r6
```

The lsa-as assembler will encode the proper instruction for you, saving some typing and making the code more readable. The example mixes both the complete syntax and the shortcut. If you examine the machine code created by lsa-as you will see that it did indeed create the same instructions.

Now let's look at the example program for this lesson. At some point before you start using the stack you do need to define where it starts in memory. The stack "grows" down, so we need to put it at a high enough memory address such that as it grows down it doesn't go so far as to run into our program or any of our data. For this example 0x1000 is plenty high enough. Also remember that a push decrements the stack pointer first then uses that memory location so 0x1000 is not actually used by the stack, the first location used is 0x0FFF.

Now we have a stack ready to use. R6 gets the value 0x0001 we push that on the stack. This means memory location 0x0FFF holds 0x0001. We increment r6 to the value 0x0002 and push that on the stack, memory location 0x0FFE holds a 0x0002. We increment r6 to 0x0003 and push that on the stack. 0x0FFD holds the value 0x0003.

And after all of this we get to our first new instruction, lsp, load sp relative. And it has only one form where you can have an 8 bit immediate. It is defined as:

```
    lsp rd,#imm
```

```
    rd = memory[sp+imm]
```

So lsp r8,#0 means read from memory wherever r2+0 is at at the moment and put that in r8. Well r2 is currently 0x0FFD and 0x0FFD+0 currently has the value 0x0003, so r8 gets a 0x0003 and we show that. Note that we have not moved r2, this was NOT a push or pop we are simply looking at the bottom few things on the stack without moving the stack.

lsp r8,#1 will look at the address r2+1. R2 is at 0x0FFD, reading 0xFFD+1 which is 0x0FFE we get 0x0002 and we show that.

lsp r8,#2 reads memory at address 0x0FFF and gets a 0x0001

Now we pop the first thing in the stack into r8. There is no rule that you have to pop to the same register you pushed, this is assembly language you can do whatever you want. So r8 gets the last thing put on the stack, a 0x0003, and we show that. This was a pop so it did move the stack pointer to 0x0FFE. The lsp r8,#1 is reading 0x0FFF, which is a 0x0002 and we show that.

We have had enough fun so we clean up the stack by popping the other two items we had pushed and show them as we go. Being assembly language, do you really have to pop everything off the stack? For examples like this, no, of course not, who is going to care if we leave stuff there, not going to get punished for breaking any rules. But for real applications you had better make an effort to put the stack back the way you found it. While cleaning up the stack if we didn't care about those two values for whatever reason instead of popping them into registers we could have added a 2 to the stack pointer to restore the stack pointer and essentially discard those items. Adding two takes two instructions so it comes out as a wash, but if we had 10 items to clean up that we didn't care to save adding 10 takes two instructions where 10 pops takes 10 instructions.

0.24 Lesson 23: Put it all together

0.24.1 Code

```
---- lesson23 ----
    b lesson23
    b lesson23
lesson23:
    lhz r2,#0x10

    lpc r3,main_add
    call r4,r3
    halt

myfun:
    llz r11,#3
    cmp r10,r11
    bc myfun_one
    push r4
    push r10
    inc r10,r10
    lpc r3,myfun_add
    call r4,r3
    pop r10
    pop r4
myfun_one:
    lhz r11,#0xFF
    stw [r11],r10
    b r4

main:
    push r4
    llz r10,#0
    lpc r3,myfun_add
    call r4,r3
    llz r10,#0
    pop r4
    b r4

main_add: .word =main
myfun_add: .word =myfun
---- lesson23 ----
```

0.24.2 Output

```
show: 0x0003
show: 0x0002
show: 0x0001
show: 0x0000
```

0.24.3 Description

Now we are going to learn the last instruction and put all of what we have learned into a real program. And not just any program, a program that uses recursion. You should have some programming experience before starting these lessons, so you can understand functions or subroutines. Recursion is where a subroutine calls itself (ideally with a point where it stops calling itself so you don't get stuck calling yourself forever).

This is the C program that we are going to write in assembly language

```
void myfun ( unsigned short ra )
{
    if(ra<3)
        myfun(ra+1);
    printf("0x%04X\n",ra);
}
unsigned short main ( void )
{
    myfun(0);
    return(0);
}
```

Before we start though we need to define our "calling convention". What is a calling convention? It is a set of rules we use to program the assembler such that we can call functions in a way that whoever writes the code for the function can follow the same programming model, and functions written by the same person or different persons will always work. The opposite would be that for each function out there you could/would have a unique calling convention, for any function that wants to call another function you would have to look up that calling convention up in some reference manual for that function and use it.

For this exercise: r3 is reserved for computing addresses to call r4 is the return address from a call, b r4 to return. r10 is the first parameter passed to a function r11 is the second, but we don't need more than one for this program Functions can destroy/use r10-r13 at will (observing of course the passed parameters arrive in those registers) but any other register used needs to be preserved (on the stack). The exception is r3 which is reserved in the sense that you can destroy it/use it any time you want to make a call, basically r3 is also a throwaway register.

Those are enough rules to implement this program.

First let's look at the assembler with the C program laid in as comments to see what is related to what.

```
    b lesson23
    b lesson23
lesson23:
    lhz r2,#0x10

    lpc r3,main_add
    call r4,r3
    halt

; void myfun ( unsigned short ra )
; {
myfun:

    ; if(ra<3)
    llz r11,#3
    cmp r10,r11
    bc myfun_one

    ;     myfun(ra+1)
    push r4
    push r10
    inc r10,r10
    lpc r3,myfun_add
    call r4,r3
    pop r10
    pop r4

myfun_one:
    ; printf("0x%04X\n",ra);
    lhz r11,#0xFF
    stw [r11],r10
    b r4
; }
; unsigned short main ( void )
; {
main:

    push r4
    ; myfun(0);
    llz r10,#0
    lpc r3,myfun_add
    call r4,r3
    pop r4

    ; return(0);
    llz r10,#0
    b r4

; }
; some global, internal variables
main_add: .word =main
myfun_add: .word =myfun
```

Now lets walk through this code to understand it.

So we start by booting up and we need to initialize the stack pointer. You have to initialize your stack pointer before you can use the stack. In this case we have set the stack pointer (r2) to 0x1000. Enough room for this example.

```
    b lesson23
    b lesson23
lesson23:
    lhz r2,#0x10
```

Normally your boot code would do other initialization stuff, eventually you need to call main(). This is our new instruction for this lesson, call.

What the call instruction does is take the pc at the time of execution (we know the pc points at the next instruction) and puts that address in the first register specified. In this case r4. The contents of the other register, r3 in this case, is then put into the pc, which is just like an unconditional branch using a register. The next instruction executed will be from whatever address we changed the pc to. It helps to think of the two registers rd, rs like you would with say an alu instruction, the one on the left, the destination gets modified (loaded with the pc), the one on the right gets read/used but is not modified (is loaded into the pc).

Remember that our calling convention says that r3 is reserved for computing addresses to call, and r4 holds the return address. call r4,r3 does just that.

Main in this case does not have any parameters so we don't have to mess with r10.

```
    lpc r3,main_add
    call r4,r3
```

If/when main returns it will return here (the instruction after the call) and we don't have anything else to do so we will halt.

```
    halt
```

Going out of order from the assembly code and going to explain main first then myfun

So we happen to use a label called main, matching the C function name to make life easier. Should have made that a calling convention rule, hmmm. This label is where code starts for this function, our entry point in this function.

```
; unsigned short main ( void )
; {
main:
```

The first thing we want to do in main() is call myfun. But we know that to make a call we will destroy r4. Of the registers we are using in main: r3 we can modify for calls, r10 thru r13 we can modify, but r4 we are going to need to return. The call will modify it so, per the calling convention we need to preserve r4 by putting it on the stack.

```
    push r4
```

Now let's call myfun. We need to pass a parameter to myfun, the value zero. The calling convention rules state that the first parameter goes in r10, it also says we do not have to preserve r10 so no need to push it on the stack.

```
    ; myfun(0);
    llz r10,#0
```

Then we can call myfun, we saw above how to use r3 for the address to call, and r4 will get filled with the address after the call so that myfun can return to finish executing this code.

```
    lpc r3,myfun_add
    call r4,r3
```

When myfun returns it returns here and, since we are done with the call to myfun we can restore whatever we had saved on the stack, in this case r4 was the only item.

```
    pop r4
```

Main is finished we want to return a zero, we know from the calling convention that the return value is stored in r10, and we know that you return from the function by branching to the address in r4.

```
    ; return(0);
    llz r10,#0
    b r4
; }
```

We have implemented our first real C function in assembly language!

Now let's get into myfun, which, due to the recursion may sound scary, but, just follow the calling convention, implement the C code in assembly language directly and you will see that it was really easy and not complicated or scary at all.

The label myfun is our entry point into this function

```
; void myfun ( unsigned short ra )
; {
myfun:
```

We know from the calling convention that the first parameter in the function (ra) is passed in register r10 and we know from C programming this is passed in by value not by address so r10 holds the value for the C variable ra.

We want to start by checking to see if ra<3, we want to call myfun if ra<3 and basically want to skip to the printf if ra>=3. If if ra>=3 skip to the printf. Remember that the carry bit can be used with a sub or cmp instruction to detect unsigned greater than or equal.

```
    ; if(ra<3)
    llz r11,#3
    cmp r10,r11
    bc myfun_one
```

If we did not branch that means ra<3 and we need to call myfun(ra+1). We are going to need to save r4 so that we can return from this function

```
    ;     myfun(ra+1)
    push r4
```

Now as a human C compiler we look ahead in the function and see that the variable ra is going to be used again after the function call we are about to make so we need to make sure it does not get lost or modified by the function we are calling. At the moment ra is in register r10 and we know from the calling convention that we have two problems, first functions can modify r10 - r13 at will so we have to get it out of r10 and keep it somewhere. Also the function we are calling needs a parameter and that parameter uses r10 so we need to get ra out of r10 and save it somewhere that will survive a function call. That is what the stack is for.

```
    push r10
```

Now ra is safe. The function call myfun(ra+1) needs to do some math on ra and then the calling convention says to put that value in r10. Well, a copy of ra is already in r10 and adding one to r10 is a simple increment.

```
    inc r10,r10
```

So our return value is safe, the ra that was passed to us is safe, the parameter to the function is ready, it is time to call the function.

```
    lpc r3,myfun_add
    call r4,r3
```

When this call to myfun returns it returns here, the first instruction after the call. We need to clean up after this function call and restore registers that we used to make the function call.

```
    pop r10
    pop r4
```

Now we are just about ready to implement printf. We could have reached this point in the code in one of two ways, one was when ra>=3 and we branched here, the other was after a call to myfun and we simply continued to execute to this point. We will put the label we need for the ra>=3 branch, and then implement printf.

```
myfun_one:
```

No we do not have a C library with a full implementation of printf, the example is cheating a bit knowing that we have a way using the simulator to show the contents of a register. That simulator feature is to store the register to address 0xFF00.

We chose to just leave ra in the register r10 since it was easy to do using the stack. The calling convention says that we can use r10 - r13 without having to preserve everything, we happen to be using r10 for ra so r11 is not a bad choice to use for the address 0xFF00.

```
    ; printf("0x%04X\n",ra);
    lhz r11,#0xFF
    stw [r11],r10
```

Being human C compilers we see that ra is no longer needed or used by the function, so we can discard it or free up any resources that we had used to save it. The only resource we allocated was the one that had already been allocated for it, r10, we don't have any rules about having to preserve or free up r10 so we have nothing to do here to clean up the C function.

End of the myfun function, time to return we know for this convention that is a b r4 instruction.

```
    b r4
```

The last bit of code is just a couple of internal variables. We know that lsa-as will compute the addresses for us if we use the =label_name format.

```
main_add: .word =main
myfun_add: .word =myfun
```

So we implemented the C program directly. To understand the output we focus on what the C program does and not have to focus on the assembly.

main calls myfun(0);

we enter myfun with ra = 0.

ra = 0 is less than 3 so call myfun(ra+1)

we enter myfun with ra = 1

ra = 1 is less than 3 so call myfun(ra+1)

we enter myfun with ra = 2

ra = 2 is less than 3 so call myfun(ra+1)

we enter myfun with ra = 3

ra = 3 is not less than 3 so we skip the myfun call and go to the printf

we printf 0x0003

the myfun call with ra = 3 is done we return

this returns to the myfun call with ra = 2 at the printf point.

we printf 0x0002

the myfun call with ra = 2 is done we return

this returns to the myfun call with ra = 1 at the printf point.

we printf 0x0001

the myfun call with ra = 1 is done we return

this returns to the myfun call with ra = 0 at the printf point.

we printf 0x0000

the myfun call with ra = 1 is done we return

this returns to main at the return(0) point.

main returns and the program is finished.

0.25 Lesson 24: Same C code but new calling convention

0.25.1 Code

```
---- lesson24 ----
    b lesson24
    b lesson24
lesson24:
    lhz r2,#0x10

    lpc r3,main_add
    llz r4,#2
    add r4,r0
    push r4 ; pc+0
    b r3    ; pc+1
            ; pc+2
    halt

myfun:
    lsp r10,#1
    llz r11,#3
    cmp r10,r11
    bc myfun_one

    inc r10,r10
    push r10
    lpc r3,myfun_add
    llz r4,#2
    add r4,r0
    push r4
    b r3
    inc r2,r2

myfun_one:
    lsp r10,#1
    lhz r11,#0xFF
    stw [r11],r10
    pop r3
    b r3

main:
    llz r10,#0
    push r10
    lpc r3,myfun_add
    llz r4,#2
    add r4,r0
    push r4
    b r3
    inc r2,r2

    llz r10,#0
    pop r3
    b r3
```

```
main_add: .word =main
myfun_add: .word =myfun

---- lesson24 ----
```

0.25.2 Output

```
show: 0x0003
show: 0x0002
show: 0x0001
show: 0x0000
```

0.25.3 Description

Using the same C code as before, we as the human C compilers are going to compile this code again. But we are going to use a different calling convention. The last time we used a register based calling convention. The lsa architecture is very much register based so it is a natural fit. But some other processors are heavily geared toward a stack based calling convention. And some processors it doesn't matter -- you can go either way.

The lsa processor has the call function which uses a register to store the return value, implying that you want to use a register for your return value in your calling convention. Some processors instead put the return value on the stack for you instead of a register implying for those architectures that you want to at least use a stack based return address scheme.

We are going to try to use the lsa processor but use a stack based calling convention:

Register r3, r4 are reserved for performing calls as you will see. r10-r13 can be used by the function without having to preserve the contents of the register before returning from the function. The last thing on the stack when you enter a function is the return value, that is stack pointer + 0 is the return value. Stack pointer + 1 is the first passed in parameter. Stack pointer + 2 is the second passed in parameter, and so on. Stack pointer + 1 is used for the return value. The calling function prepares the stack before calling the function. The function leaves the number of items on the stack with the exception that the function pops the return value (leaving the parameters on the stack). The calling function cleans up the stack to the state it was before the call.

So here is our C program again.

```
void myfun ( unsigned short ra )
{
    if(ra<3)
        myfun(ra+1);
    printf("0x%04X\n",ra);
}
unsigned short main ( void )
{
    myfun(0);
    return(0);
}
```

To really use the stack for calling functions takes a little bit of care (with this processor), here is what we are going to do, r3 will get the address of the function. We set up the stack according to the convention and an unconditional branch will serve to call the function. We have to figure out what the return address is so that we can push it onto the stack.

Here is a skeleton of the code used any time a function is called using this calling convention.

```
    ;setup last parameter and push it
    ;setup next to last parameter and push it
    ...
    ;setup second parameter and push it
    ;setup first parameter and push it
    lpc r3,main_add  ; r3 gets address of the function
    llz r4,#2         ; r4 is going to hold are return address temporarily
    add r4,r0         ; our return address is pc based
    push r4           ; pc+0
    b r3              ; pc+1
                      ; pc+2
```

From the time we add r4,r0 through to b r3 we cannot mess around. The add uses the program counter and those few lines of code are carefully crafted so that we can use the program counter in the add instruction to figure out what the return address is after the b r3 instruction.

We do that by visualizing what the instruction address is after the b r3 instruction relative to the add instruction. The comments on the side, pc+0, etc are relative to the add instruction and show us that if we add two to the pc in the add instruction that will give us the address to the instruction after the b r3, essentially after the call to the function. That is how we figured out that we needed to pre-load r4 with a 2 to make this all work.

That was the hard part. It gets really easy from here on out. For starters when someone calls your function, the parameters for that instance of the function are on the stack in a known place relative to the stack pointer.

```
sp+N    last parameter
sp+N-1   next to last parameter
...
sp+2    second parameter
sp+1    first parameter
sp+0    return address
```

In the case of myfun, ra will be at sp+1 each time we call it, and sp+0 is the return address.

We know how to reach sp+1 with an lsp rd,#1 instruction which we can do at anytime in this call to myfun (so long as we keep track of our stack usage within myfun). By having these items on the stack when we call another function (like the recursive call to myfun) we don't have to wrap that call to myfun with pushes and pops to save ra and the return address, they are already safe.

The call to myfun has a stack that looks like this

```
sp+1 0x0000, ra from the myfun(0) call
sp+0 return address to get back to main
```

When we call myfun(0+1) from within the myfun(0) call the stack now looks like this

```
sp+3 0x0000, ra from the myfun(0) call
sp+2 return address to get back to main
sp+1 0x0001, ra from the myfun(1) call
sp+0 return address to get back to myfun(0)
```

And when myfun(1) calls myfun(2)

```
sp+5 0x0000, ra from the myfun(0) call
sp+4 return address to get back to main
sp+3 0x0001, ra from the myfun(1) call
sp+2 return address to get back to myfun(0)
sp+1 0x0002, ra from the myfun(2) call
sp+0 return address to get back to myfun(1)
```

and so on.

Notice the inc r2,r2 after all of the calls to myfun()? Remember the calling convention says the callee will remove the return address but the caller has to remove/clean up the rest of the stack (the passed in parameters) so before we have completed a call to myfun we have to remove the passed in parameter from the stack. Since we no longer need that value (and pass by value in C also means the C function is allowed to mess up that copy of the passed value, so that stack location does not necessarily have what we wrote before the call) we can use an inc to remove it from the stack by putting r2 back where it was before we started preparing for this call.

Another great benefit to using a stack based calling convention is that we can have a lot of parameters. With register based we are limited by the number of registers that we can spare. The prior example didn't say it directly, but it implied we were limited to only four parameters for a function. What often happens is that register based is faster because registers are faster than memory (generally), but you have a limited number. Depending on the total number of registers in the processor you may have a rule that says the first X number of parameters are passed in on these registers and parameters X+1 on are passed in on the stack. And that way you can balance performance and still meet the minimum number of passed parameter rules for the programming language if any.

You are now experienced enough that with the explanation of how to make stack based calls, you can read through this example program and understand how it implements the C code directly.

Normally you would not want to use the lsa processor this way, this was for educational purposes. What you might want to do though is take advantage of the call instruction giving you a register based return address, but use the stack for parameters.

0.26 Final Comments

0.26.1

That is all...I declare you an assembly language programmer. Hopefully you didn't just blaze through this without taking time to play around and understand each lesson. Now would be a good time to either go back and do that or take what you know and move forward and write useful and interesting programs.

Take some time if you can and see what else this lsa processor package has to offer, it offers much more than just teaching assembly language.

Experiencing assembly for the first time (if this was your first time) with a somewhat straight forward and simple architecture like this you can take that knowledge and add to it by learning assembly language for other processors. The concepts are the same, loading, storing, some sort of program counter, usually a stack, etc. But the syntax is usually different.

Other processors likely will take the things you used here in multiple instructions or steps and combine them into one instruction. For example an alu instruction with immediate values

```
    add r7,#0x12      ; r7 = r7 + 0x12
```

Or three register/parameter instructions

```
    add r7,r8,r9      ; r7 = r8 + r9
    add r7,r8,#0x12   ; r7 = r8 + 0x12
```

Lots of cool things but lots more to remember and learn how to use.

A few recommended architectures:

-The msp430 from Texas Instruments

-The thumb instruction set which is the 16 bit subset of the ARM instruction set supported by some/many ARM processors. Note there is also a thumb2 instruction set, which as of this writing is on a very limited number of ARM cores. Not talking about thumb2 just thumb, thumb2 takes thumb to a whole new, complicated, level. -The ARM instruction set, the grandfather of the ARM instruction sets has three register stuff, etc. Very nice and clean and feature rich. -The MIPS family. The dlx derivative is also a teaching instruction set created by the mips designers.

-For fun but not necessarily practical the 6502 instruction set and the pdp11 instruction set.

The last instruction set you want to learn is x86. Most of the world may revolve around it but that is through momentum and not necessarily, a great design, or history. It is one of the harder instruction sets to learn and master and because of the long history and tweaks, microcoded cores, multiple x86 chip vendors, etc, no two x86 cores perform the same way. They all execute x86 instructions in some fashion (one generation building on the prior). If you want to use asm to hand tune for performance you will find that nicely tuned asm on one particular x86 core runs really slow on another and vice versa. You want your x86 asm code to be somewhat generic and not overly optimized for one core so that it runs good enough on all of the cores. Or at least be aware that tuning for a core takes a learning curve to understand that specific core. You can make your code faster for that core, but your tuned code is limited to performing on that core/family. That may be fine for the hardware you are developing this code for.

So long as our laptops and desktops and servers continue to be dominated by x86 processors it is worth learning enough to be able to read x86 assembler so that you can inspect what your C compiler is really doing when it compiles and optimizes your code. Be able to debug your code at that level to find if the C compiler is doing something that you didn't realize the C language permitted or perhaps you forgot to declare a variable volatile or something like that and the C compiler re-arranged your code.

If you are like me diving into x86 was a failure, backing up and learning a cleaner, digestible, instruction set first will allow you to understand and use the x86 instruction set (if you feel you really need to). After failing on my own to learn x86 I took a pdp11 assembly language class (no I am not THAT old, the class used computers that were antiques at the time), and then was able to easily learn x86 assembler. This portion of the lsa project is specifically to pass it on. You are likely not going to find a DEC Pro 350 laying around in working order with working floppy disks with working software you can use to write programs. You can get an msp430, which is a good teaching instruction set for less than $5, but you are likely afraid to try to figure out how to use it and probably didn't know it would be better to learn on something other than your desktop/laptop processor. In a simulation environment like this one you can, ideally, see everything going on and have no fear of mistakes, you are not going to trash your hard drive, blow up your video card or monitor. Etc.