LRSColinAssembler

Introduction

In yet another attempt to explain machine language to Colin, I’ve written a program that accepts a source program written in assembler language (very similar to that of IBM’s mainframe 360 series), produces an in-memory binary representation of it (rather than writing it out to a file) and then (if there are no errors) executes it. See <https://en.wikipedia.org/wiki/Assembly_language>

Note: Over the decades, there have been any number of different approaches to how CPUs work. Rather than getting too bogged down with interminable qualifications (e.g. “computers do it *this* way, except that the Motorola xxx does it *that* way, unlike the Intel yyy which does it *this other way*”), I’ll often make general statements that computers work in a specific way. And you should take these with a grain of salt. There will probably be exceptions to practically every general statement I make. But the statements will apply to many/most current CPU architectures, and are close enough for this introductory essay.

Micro-C – A Weird Little Language

I don’t know how well the following is going to work out, but I’m going to give it a shot.

The CPU I’ll be describing below has exactly 18 different basic operations it can perform, plus a handful of higher-order extensions such as printing a number or a string. The mnemonics for these extensions begin with “$” (e.g. $PSTRING to print a string). There is also a “DI” notation to specify a field with an integer in it, and a “DS” notation to specify a field that is a string.

These 18 operations can be mapped to a fictitious C-like language that I’ll describe below. Note that there are a *lot* of things that “mini-C”[[1]](#footnote-1) can’t do. For example, you can add one number to another (R5 += count), but you can’t add two numbers and put the result in a third place (a = b + c). And you certainly can’t do expressions (a = b \* (c + d / 5)).

So be prepared for something shockingly primitive.

* There are 16 special variable names, R0..R15. These are *registers* (see later). There’s also the separate Condition Code register called CC that holds a few bits of data encoding the arithmetic result of certain expressions, indicating whether the result was negative, zero or positive.
  + Note: While the CC is technically a register, you can’t work with it directly in machine language, so unless specified specifically below, referenced to registers below refer only to R0..R15.
* Assignments (with one exception) can be done *only* to registers (R5 = Count). You can assign to a field, but only from a register (Count = R5, but not Count = InitialValue).
* To keep things as simple as possible, all arithmetic is done using 16-bit words.

The statements allowed in the language are given in the following table: The abbreviations are:

* *Reg* means a register, R0..R15
* *Addr* is the address of a field in memory. Note the use of the “@” prefix to indicate the address of a field, not the contents of the field
* *Fld* is a field in memory. Think *variable*. Hopefully it should be clear from context whether this is the address of the field or its contents
* *Num* is a numeric constant, such as 5. Note that negative values are not allowed (-5), although you can accomplish this with setting a register to 0 and then subtracting a field with 5 in it from the register.
* *ArithOp* is one of “+=”, “-=”, “\*=” or “/=”
* *RelOp* is a relational operator, one of “<”, “<=”, “=”, “!=”, “>=”, “>”
* The notation *lob(reg)* refers to the low-order (least significant) byte of a register.
* All statements can be prefixed by a label. For example, “Loop: R5 = R6”
* All Addresses and Fields can have an optional subscript (which must be a register). An example is “R5 = Array[R6]”
* Comments begin with “//” and continue to the end of the line
* Keywords (e.g. Call, If, print, etc.) are case-insensitive (can be any mix of upper or lower case characters)

|  |  |
| --- | --- |
| Statement | Description / Example |
| Reg = Reg or Fld or Num | R5 = R6 or R5 = Count or R5 = @Count; R5 = 10. Sets the Condition Code (CC) |
| Fld = Reg | Count = R5 |
| Reg ArithOp Fld/Reg | R5 += Increment; R5 \*= R6 |
| Call Reg,Addr | Transfer control to the specified address. The register is assigned the address of the instruction immediately after the CALL |
| Ret Reg | Branches to the address in the register (return from subroutine) |
| lob(Reg) = Addr | Sets the low-order byte of the register. The high-order byte is not changed |
| Fld = lob(Reg) | Sets the first byte of Fld to the low-order byte of Reg |
| Reg Relop Reg or Reg Relop Fld | R5 == 0 or R5 < Count. Sets the Condition Code (CC) |
| if CC is Relop go to Addr | If CC is <= go to Loop |
| Goto Label | Transfer control to the statement at that label |
| int Count = Num | Declares *Count* to be a field initialized to Num |
| string Msg = “Hello world” | Declares *Msg* to be a field initialized to the string “Hello World” |
| print Reg / Fld | Prints the contents of a register (print R5), the numeric contents of the field, or the string at address Fld |
| Trace On / Off | Starts / stops a special tracing mode |
| Stop | Stops executing the program |

And that’s it.

So let’s see what a program to calculate the numbers from 1 to 10 and their squares would look like.

print Msg

R3 = 1 // Starting value

R2 = 10 // Ending value

Loop: R3 > R2 // Sets Condition code

If CC is > go to End

R4 = R3

Temp = R4

R4 \*= Temp

Call R14,Show

R3 += One

Goto Loop

End: Stop

Show: print R3

Print Arrow

Print R4

Print Sep

Ret R14

Int Temp = 0;

Int One = 1;

String Msg = “Here are your numbers and their squares: “

String Arrow = “=>”

String Sep = “, “

As I said, primitive, but maybe not totally horrible.

So this the level of details that computers run at. Keep this in mind as we go through the sections below. You’ll see that things that are unfamiliar at first will actually map into mini-C.

How a CPU Works – Machine Language

Computers, of course, work exclusively on binary data, 1’s and 0’s. We usually group these into 8-bit bytes[[2]](#footnote-2) and that’s what we’ll use here.

Machine language is simply a series of bytes and each byte value has a defined operation associated with it[[3]](#footnote-3). So (in the system described in this document) when the CPU sees a byte with 0x5A in it, it knows that it’s to ADD two numbers together[[4]](#footnote-4). Similarly, 0x5B means SUBtract. 0x45 means to CALL a subroutine and 0x07 means to RETurn from a subroutine.

In a very real (but technical) sense, when computers are executing programs, they’re actually being an interpreter. This is necessary. You can imagine I suppose a *very* specialized CPU that is hardwired to do exactly one calculation. But in a general purpose computer, where the programs that can run change every day (or even every second), you need something that says “What should I do next? Oh, add. OK I’ll get the addition circuitry working on it. Once that’s done, it goes back to the what-should-I-do-next stage. And that’s an interpreter.

The CPU does have circuitry that knows how to fetch a byte from memory, how to place it into a *register* (see next section), how to pass two numbers to the arithmetic routines (Add, Subtract, etc.), how to send the result back to memory, and so on. All very low-level stuff. Which is good. The manufacturer (Intel, Motorola, IBM, etc.) can keep programs the same from the programmer’s point of view, while being able (thanks to an ever-increasing number of transistors on a chip) to improve the performance of the next generation of a CPU by changing the implementation details under the hood. My favorite example of this is the IBM 1620 computer. Model 1 of the computer didn’t have circuitry to truly add two numbers. It used a lookup table in memory at addresses 300-399. If it wanted to add, say, 2 + 5, the Add circuitry would fetch the digit in location 325 and find 7! The 1620 model 2 actually did have circuitry to really add numbers.

Enconding Instructions Numerically

A *register* is a one of several on-CPU-chip, high speed memory slots. See more below. Suppose we want to (for whatever reason) load data from memory address 32 (0x0020) into register 6, add the data from memory address 42 (0x002A), and store the result back in address 162 (0x00A2). This would take 3 machine instructions, a Load, an Add and a Store. The Load instruction has the numeric value 88 (0x58), Add is 90 (0x5A) and Store is 80 (0x50). In memory this would look like (with blanks added to improve readability and in hex): 58600020 5A60002A 506000A2. So the first instruction would have operation code (“opcode”) 0x58 (Load), refers to register 6 (0x60 – we’ll worry about the 0 in 60 later), and an address 0x002A. And similarly for the other two instructions

While working with numeric code was mandatory in the earliest computers, pretty quickly someone wrote a program called an “assembler” that could take a text representation of each instruction and generate the proper numeric values. So the programmer could then write **L 6,First**, feed it to the assembler program and (assuming that there was a field called **First** that was at location 32 (0x0020), would generate 0x58600020. So our 3-instruction program above would be (assuming a field called, say, **Increment** at location 42 (0x002A) and **Result** at 162 (0x00A2))

**L 6,First**

**ADD 6,Increment**

**ST 6,Result**

*Much* better!

The mini/360 Architecture

First of all, to make things a bit simpler, instead of working with 32-bit words as the 360 does, we work exclusively with 16-bit words. All numbers are signed giving us a range from -32,768 to +32,767. Floating point is not supported.

Memory is organized as a single-dimensional array of bytes. The amount of memory is fixed at 1000 bytes, which should be large enough for any of our toy programs.

16-bit words are encoded in Big-Endian format (<https://en.wikipedia.org/wiki/Endianness>) with the most significant byte first in memory. Consider the decimal value 4660 (0x1234) stored at, say, decimal address 768 (hex 0x0300). Then the byte at address 0x300 would be 0x12 and the byte at 0x0301 would be 0x34.

Note that for historical reasons all x86-style CPUs use Little-Endian format. So the data would look backwards in memory. 4660 in memory would have byte 0x0300 as 0x34 followed by 0x12. I’ve chosen to use Big-Endian so the data looks simpler.

There’s only so much space on a given chip, so you can’t have the billions of transistors for the CPU and even more billions of transistors for multi-gigabyte memory on the same chip. So most memory is not on the same chip as the CPU. This means that there’s a certain delay as the CPU and one (or more) of the memory chips synchronize themselves as data is copied to and from memory. Also, there is no known technology that economically allows memory chips to be both dense (gigabytes per chip) and fast. So memory is normally slower than the CPU.

A way to partially alleviate this speed mismatch is to put a certain limited amount of fast memory on the CPU chip. There are several techniques for accomplishing this, notably *cache* memory (<https://en.wikipedia.org/wiki/Cache_(computing)>), but that’s beyond the scope of this document.

Historically the first way of coping with the mismatch was through the use of *registers* (<https://en.wikipedia.org/wiki/Processor_register>). A register is a piece of memory, not a part of main memory, that sits on the same chip as the CPU and is implemented in fast memory technology. However the amount of memory a register can hold is very limited. On a /360, it contains 32 bits. On our mini/360 it’s only 16 bits. We have 16 registers, numbered 0 to 15. In our assembler program it will also accept a register number prefixed with “R” (e.g. R5). If we had a cross-reference feature in the assembler (we don’t), this would give us a convenient way to see where in the program a register (also, fields) were used, which can be useful in debugging.

In addition to using registers for numerical processing, they can also be used as *index registers*. These are mostly used for subscripts. Many instructions reference a field in memory by its address. But suppose you want to treat an address as the beginning of an array and process in turn array[0], array[1] and so on. Then (for many instructions) you can specify that the contents of a specified register is implicitly added to the specified address. Note: if the specified index register is register 0, then nothing is added to the target address. To make this more concrete. Suppose a Load (L) instruction loads data from field *Data* at address 0x300 into register 6 – L R6,Data. Fine. In memory this would be 0x58600300. The data would be fetched from address 0x300 since the index register specified is register 0 (the 4th nybble in the instruction) which, when used as an index register, is taken as having 0 in it, regardless of what is actually in register 0. But suppose register 7 had the value 386 (0x0182) in it and we wrote L R6,Data[R7]. In hex this would be 0x58670300. Then data would be fetched from address 0x482.

Finally, certain instructions (e.g. Compare (CMP)) sets the *Condition Code*, a separate small register than encodes whether one field was less than, equal to, or greater than the other. Note that when a Compare is done, it’s implemented internally as Subtraction but not modifying the registers involved. So if the subtraction result is zero, then the two numbers are equal. Hence the Branch Equal and Branch Zero opcodes (see below) are the same.

Note: When I say things like “Adds the contents of *Count*” below, I mean “Adds the contents of the field at address *Count*”. And when I say “Copies R4 to R5” I mean that it copies the value in register 4 to register 5.

Instructions – Load and Store

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Example** | **Opcode** | **Description** |
| Load (L) | L R5,Count | 0x58 | Loads contents of *Count* into R5 |
| Load Register (LR) | LR R5,R4 | 0x18 | Copies R4 to R5 |
| Load Address (LA) | LA R5,Count | 0x41 | Loads the address of field *Count* into R5 |
| Load and Test Register (LTR) | LTR R5,R4 | 0x12 | Copies R4 to R5 and compares R5 to 0, setting the condition code. |
| Store (ST) | ST R5,Count | 0x50 | The opposite of Load. Stores R5 into *Count* |
| Insert Character (IC) | IC R5,Msg | 0x43 | Inserts the byte at Msg into R5. Only the low-order 8 bits of the register are modified. |
| Store Character (STC) | STC R5,Msg | 0x42 | Stores the low-order 8 bits of R5 into the first byte of field *Msg* |

Instructions – Arithmetic (All set the Condition Code)

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Example** | **Opcode** | **Description** |
| Add (ADD) | ADD R5,Count | 0x5A | Adds the contents of *Count* to R5 |
| Add Register (AR) | AR R5,R4 | 0x1A | Adds R4 to R5 |
| Subtract (SUB) | SUB R5,Count | 0x5B | Subtracts the contents of *Count* from R5 |
| Subtract Register (SR) | SR R5,R4 | 0x1B | Subtracts R4 from R5 |
| Multiply (MUL) | MUL R5,Count | 0x5C | Multiplies R5 by the contents of *Count* |
| Divide (DIV) | DIV R5,Count | 0x5D | Divides R5 by the contents of *Count*, giving the truncated integer result. For example, 19 / 7 => 2 |

Instructions – Subroutines

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Example** | **Opcode** | **Description** |
| Call (CALL) | CALL R14,Sub | 0x45 | Transfers control to the routine at *Sub* and sets R14 to the address of the instruction after the Call |
| Return (RET) | RET R14 | 0x07 | Transfers control to the code at the address in R14 |

Instructions – Compare

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Example** | **Opcode** | **Description** |
| Compare (CMP) | CMP R5,Count | 0x59 | Compares the contents of R5 with the contents of the field at address *Count*. Sets the condition code to Low if R5 < Count, to Equal/Zero if they are the same and to High if R5 > Count |
| Compare Register (CR) | CR R5,R4 | 0x19 | Compares the contents of R5 to R4 and sets the condition code |

Instructions – Branches

There are basically only two branch[[5]](#footnote-5) instructions, Branch on Condition and Branch on Condition Register.

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Example** | **Opcode** | **Description** |
| Branch (see mnemonics below) | B\* Address | 0x45 | Branches to the specified Address, depending on the Condition Code. |
| Branch Register (see mnemonics below) | B\*R R5 | 0x05 | Branches to the Address in the specified register, depending on the Condition Code. |

The Branch mnemonics allow us to, for example, branch only if (say) a previous Compare (CMP) opcode found one value lower than (less than) another. In that case, we’d use the BL mnemonic.

|  |  |
| --- | --- |
| **Instruction** | **Description** |
| Branch Low (BL) | Branches if the Condition Code is Low/Less than |
| Branch Low or Equal (BLE) | Branches if the Condition Code is Low/Less than or Equal/Zero |
| Branch Equal (BE or BZ) | Branches if the Condition Code is Equal/Zero |
| Branch Not Equal (BNE or BNZ) | Branches if the Condition Code is NOT Equal/Zero |
| Branch High or Equal (BHE) | Branches if the Condition Code is High or Equal/Zero |
| Branch High (BH) | Branches if the Condition Code is High |
| Branch (B) | Branches unconditionally |

There are Branch Register versions of all these (e.g. BLR, BNZR, BR, etc.)

Instructions – Pseudo-Instructions

Rather than get bogged down in I/O, I’ve implemented pseudo instructions. They are…

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Example** | **Opcode** | **Description** |
| $TRACEON | $TRACEON | 0xFA | No arguments. Starts tracing mode that shows the results of each opcode executed |
| $TRACEOFF | $TRACEOFF | 0xFB | No arguments. Turns off tracing mode |
| $PREG | $PREG R5 | 0xFC | Prints (displays) the contents of R5 in decimal |
| $PNUM | $PNUM Address | 0xFD | Prints (displays) the contents of the field at the given address |
| $PSTRING | $PSTRING Address | 0xFE | Prints (displays) the field at the given address until a byte with 0x00 is found |
| $STOP | $STOP | 0xFF | No arguments. Terminates the program |

Pseudo-Ops

These “opcodes” don’t actually generate machine instructions; they define data, either numeric or string. They normally have a label.

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Example** | **Opcode** | **Description** |
| DI | DI 42 | N/A | A 16-bit word with the value 42 (0x2A) in it |
| DS | DS Hello world | N/A | Defines the string “Hello world”. Appends a zero byte (0x00) to the end so that $PSTRING knows when to stop. Adjacent blanks are collapsed into a single blank. Commas are ignored. Special characters: \b for a blank, \c for a comma, \n for a newline, \s for semicolon, \t for a tab character |

Comments

Any text after and including a semicolon (;) are ignored. To print a semicolon, use \s in a DS pseudo-op.

Tips and Techniques

* Opcode mnemonics (L, CMP, $PNUM, etc.) are case-insensitive. All labels are case sensitive.
* A Label is defined as a symbol that starts at the first position of a line. Opcodes must start after the first blank
* To load a positive integer into a register, use Load Address (e.g. LA R5,42).
* To load zero into a register, subtract it from itself (e.g. SR R5,R5. Or LA R5,0)
* To create a negative number, subtract it from zero. Note that this restriction is because I kept the parsing of the program as simple as I could. So, for example, the DI pseudo-op doesn’t support a leading “-“.
  + - SR R5,R5
    - SUB FortyTwo
  + …
  + FortyTwo DI 42
* The program automatically copies all output from the Assembler to the clipboard. This makes it easy when the program is running with tracing. You can open a text editor in another window and paste in the Assembler output, with source code, with hex address and machine code, and now use it to help guide you through the tracing output in the original window.

Sample Code

Let’s print out the numbers from 1 to 10.

; $TRACEON ; Uncomment to trace

$PSTRING Msg

LA R2,1 ; Starting value

LA R3,1 ; Increment

Loop CMP R2,Ten ; Stopping value

BLE Print ; We’re good to go!

$PSTRING NL ; Terminate line

$STOP ; Buh-bye

Print $PSTRING Space ; Make output look pretty

$PREG R2 ; Print our value

AR R2,R3 ; Add increment

B Loop ; Back to loop end test

Ten DI 10

Msg DS The numbers from 1 to 10 are:\b

Space DS \b

NL DS \n

1. I should probably call it “micro-C” or even “nano-C”. [↑](#footnote-ref-1)
2. Some earlier computers worked with decimal digits (e.g. the IBM 1620), but I’m not aware of any modern CPU architecture doing so. [↑](#footnote-ref-2)
3. Some of which might involve (in higher-level programming language terms) throwing an invalid instruction exception. [↑](#footnote-ref-3)
4. The subsequent bytes indicate which numbers are to be added. [↑](#footnote-ref-4)
5. Which some CPU architectures call a Jump. [↑](#footnote-ref-5)