Assembly language for Power Architecture, Part 1: Programming concepts and beginning PowerPC instructions

Getting to the heart of the machine

Jonathan Bartlett

October 03, 2006

The POWER5™ processor is a 64-bit workhorse used in a variety of settings. Starting with this introduction to assembly language concepts and the PowerPC instruction set, this series of articles introduces assembly language in general and specifically assembly language programming for the POWER5.

View more content in this series

The POWER5 and other PowerPC processor families

The POWER5™ processor is the latest processor in a line of high-powered processors supporting the PowerPC® instruction set. The first 64-bit processor in this line of processors was the POWER3. The Macintosh G5 processor was an extension of a POWER4 processor with an additional vector processing unit. The POWER5 processor is the latest generation POWER processor, having both dual-core and symmetric multithreading capabilities. This allows a single chip to process four threads simultaneously! Not only that, each thread can execute a group of up to five instructions every clock cycle.

About this "Assembly language for Power Architecture" series

This four-part series of articles introduces assembly language in general and specifically assembly language programming for the POWER5:

- Part 1: Programming concepts and beginning PowerPC instructions
- Part 2: The art of loading and storing on PowerPC
- Part 3: Programming with the PowerPC branch processor
- Part 4: Function calls and the PowerPC 64-bit ABI

The PowerPC instruction set is used on a wide variety of chips from IBM and other vendors, not just the POWER line. It is used in server, workstation, and high-end embedded scenarios (think digital video recorders and routers, not cell phones). The Gekko chip is used in Nintendo's...
GameCube and the Xenon is used in the Microsoft Xbox 360. The Cell Broadband Engine is the up-and-coming architecture using the PowerPC instruction coupled with eight vector processors. The Sony PlayStation 3 will use the Cell, as well as numerous other vendors considering it for a wide variety of multimedia applications.

As you can see, the PowerPC instruction set is useful far beyond the POWER processor line. The instruction set itself can operate in either a 64-bit mode or a reduced 32-bit mode. The POWER5 processor supports both, and Linux distributions on POWER5 support both applications compiled for 32-bit and 64-bit PowerPC instruction sets.

**Getting access to a POWER5 processor**

All current IBM iSeries and pSeries servers use POWER5 processors and can run Linux. In addition, open source developers can request access to POWER5 machines for porting applications through IBM's OpenPower Project (see Related topics for a link). Running a PowerPC distribution on a G5 Power Macintosh will give you access to a slightly modified POWER4 processor, which is also 64-bit. G4s and earlier are only 32-bit.

Debian, Red Hat, SUSE, and Gentoo all have one or multiple distributions supporting the POWER5 processor, with Red Hat Enterprise Linux AS, SUSE Linux Enterprise Server, and OpenSUSE being the only ones supporting the IBM iSeries of servers (the rest supporting IBM pSeries of servers).

**High level versus low level programming**

Most programming languages are fairly processor-independent. While they may have specific features that rely on certain processor abilities, they are more likely to be operating-system-specific than processor-specific. These high-level programming languages are built for the express purpose of providing distance between the programmer and the hardware architecture. This is for several reasons. While portability is one of them, probably more important is the ability to provide a friendlier model that is geared more towards how programmers think as opposed to how the chip is wired.

In assembly language programming, however, you are working directly with the processor's instruction set. This means that you have essentially the same view of the system that the hardware does. This has the potential to make assembly language programming more difficult because the programming model is geared towards making the hardware work instead of closely mirroring the problem domain. The benefits are that you can do system-level work easier and perform optimizations that are very processor-specific. The drawbacks are that you actually have to think on that level, you are tied to a specific processor line, and you often have to do a lot of extra work to get the problem domain accurately modeled.

One nice thing about assembly language that most people don't think about is that it is very concrete. In high-level languages, there is a lot going on with every expression. You sometimes have to wonder just what is occurring under the hood. In assembly language programming, you can have a full grasp of exactly what the hardware is doing. You can step through the hardware-level changes every step of the way.
Fundamentals of assembly language

Before getting into the instruction set itself, the two keys to understanding assembly language are understanding the memory model and understanding the fetch-execute cycle.

The memory model is very simple. Memory stores only one thing -- numbers with a limited range called a byte (on most computers, this is a number between 0 and 255). Each memory location is located using a sequential address. Think of a giant roomful of post-office boxes. Each box is numbered, and each box is the same size. This is the only thing that computers can store. Therefore, everything must ultimately be structured in terms of fixed-range numbers. Thankfully, most processors have the ability to combine multiple bytes together as one unit to handle larger numbers, and also numbers with different ranges (such as floating-point numbers). However, how specific instructions treat a region of memory is irrelevant to the fact that every memory location is stored in the exact same manner. In addition to the memory lying in sequential addresses, processors also maintain a set of registers, which are temporary locations for holding data being manipulated or configuration switches.

The fundamental process that controls processors is the fetch-execute cycle. Processors have a register known as the program counter, which holds the address of the next instruction to execute. The fetch-execute works in the following way:

- The program counter is read, and the instruction is read from the address listed there
- The program counter is updated to point to the next instruction
- The instruction is decoded
- All memory items needed to process the instruction are loaded
- The computation is processed
- The results are stored

The reality of how this occurs is actually much more complicated, especially since the POWER5 processor can execute up to five instructions simultaneously. However, this suffices for a mental model.

The PowerPC architecture is characterized as a load/store architecture. This means that all calculations are performed on registers, not main memory. Memory access is simply for loading data into registers and storing data from registers into memory. This is different from, say, the x86 architecture, in which nearly every instruction can operate on memory, registers, or both. Load/store architectures typically have many general-purpose registers. The PowerPC has 32 general-purpose registers and 32 floating-point registers, which are each numbered (as opposed to the x86, where the registers are named rather than numbered). The operating system's ABI (application binary interface) will likely make special use of the first. It also has a few special-purpose registers for holding status information and return addresses. There are other special-purpose registers available to supervisor-level applications, but these are beyond the scope of this article. The general-purpose registers are 32 bits on 32-bit architectures and 64 bits on 64-bit architectures. This article series focuses on the 64-bit architectures.

Instructions in assembly language are very low-level -- they can only perform one (or sometimes a few) operations at a time. For example, while in C I can write $d = a + b + c - d$
In assembly language, each addition, subtraction, and function call would have to be its own instruction, and in fact the function call might be several. This may seem tedious at times. However, there are three important benefits. First, simply knowing assembly language will help you write better high-level code, since you will understand what is going on at the lower levels. Second, the fact that you have access to all of the minutiae of assembly language means that you can optimize speed-critical loops beyond what your compiler can do. Compilers are fairly good at code optimization. However, knowing assembly language can help you understand both the optimizations that the compiler is making (using the -S switch in gcc has the compiler spitting out assembly rather than object code), and help you spot the ones that it is missing. Third, you will have access to the full power of the PowerPC chip, much of which can actually make your code more compact than possible in higher-level languages.

So, without further ado, let's dive into the PowerPC instruction set. Here are some PowerPC instructions that are useful to beginners:

- **li REG, VALUE**
  - loads register REG with the number VALUE

- **add REGA, REGB, REGC**
  - adds REGB with REGC and stores the result in REGA

- **addi REGA, REGB, VALUE**
  - adds the number VALUE to REGB and stores the result in REGA

- **mr REGA, REGB**
  - copies the value in REGB into REGA

- **or REGA, REGB, REGC**
  - performs a logical "or" between REGB and REGC, and stores the result in REGA

- **ori REGA, REGB, VALUE**
  - performs a logical "or" between REGB and VALUE, and stores the result in REGA

- **and, andi, xor, xori, nand, nand, and nor**
  - all of these follow the same pattern as "or" and "ori" for the other logical operations

- **ld REGA, 0(REGB)**
  - use the contents of REGB as the memory address of the value to load into REGA

- **lbz, lhz, and lwz**
  - all of these follow the same format, but operate on bytes, halfwords, and words, respectively (the "z" indicates that they also zero-out the rest of the register)

- **b ADDRESS**
  - jump (or branch) to the instruction at address ADDRESS
bl ADDRESS
    subroutine call to address ADDRESS

cmpd REGA, REGB
    compare the contents of REGA and REGB, and set the bits of the status register appropriately

beq ADDRESS
    branch to ADDRESS if the previously compared register contents were equal

bne, blt, bgt, ble, and bge
    all of these follow the same form, but check for inequality, less than, greater than, less than or equal to, and greater than or equal to, respectively.

std REGA, 0(REGB)
    use the contents of REGB as the memory address to save the value of REGA into

stb, sth, and stw
    all of these follow the same format, but operate on bytes, halfwords, and words, respectively

sc
    makes a system call to the kernel

Notice that all instructions that compute a value use the first operand as the destination register. In all of these instructions the registers are specified only by their number. For example, the instruction for loading the number 12 into register 5 is li 5, 12. We know that 5 refers to a register and 12 refers to the number 12 because of the instruction format -- there is no other indicator.

Each PowerPC instruction is 32 bits long. The first six bits determine the instruction and the remaining portions have different functions depending on the instruction. The fact that they are fixed-length allows the processor to process them more efficiently. However, the limitation to 32 bits can cause a few headaches, which we will encounter. The solutions to most of these headaches will be discussed in part 2.

Many of the instructions above make use of the PowerPC extended mnemonics. This means that they are actually specializations of a more general instruction. For example, all of the conditional branches mentioned above are actually specializations of the bc (branch conditional) instruction. The form of the bc instruction is bc MODE, CBIT, ADDRESS. The CBIT is the bit of the condition register to test. The MODE has lots of interesting uses, but for simple uses you set it to 12 if you want to branch if the condition bit is set, and 4 if you want to branch if the condition bit is not set. Some of the important condition register bits are 8 for less than, 9 for greater than, and 10 for equal. Therefore, the instruction beq ADDRESS is really bc 12, 10 ADDRESS. Likewise li is a specialized form of addi and mr is a specialized form of or. These extended mnemonics help make PowerPC assembly language programs more readable and writable for simpler programs, while not removing the power available for more advanced programs and programmers.
Your first POWER5 program

Now let's get into some actual code. The first program we write won't do anything at all except load two values, add them together and exit with the result as a status code. Type the following into a file named sum.s:

Listing 1. Your first POWER5 program

```assembly
#Data sections holds writable memory declarations
.data
.align 3 #align to 8-byte boundary

#This is where we will load our first value from first_value:
    #"quad" actually emits 8-byte entities
    .quad 1
second_value:
    .quad 2

#Write the "official procedure descriptor" in its own section
.section ".opd","aw"
.align 3 #align to 8-byte boundary

#procedure description for .start
.global _start
#Note that the description is named .start,
# and the beginning of the code is labeled .start
_start:
    .quad .start, .TOC.@tocbase, 0

#Switch to ".text" section for program code
.text
_start:
    #Use register 7 to load in an address
    #64-bit addresses must be loaded in 16-bit pieces
    #Load in the high-order pieces of the address
    lis 7, first_value@highest
    ori  7, 7, first_value@higher
    #Shift these up to the high-order bits
    rldicr 7, 7, 32, 31
    #Load in the low-order pieces of the address
    oris 7, 7, first_value@h
    ori  7, 7, first_value@l
    #Load in first value to register 4, from the address we just loaded
    ld 4, 0(7)
    #Load in the address of the second value
    lis 7, second_value@highest
    ori  7, 7, second_value@higher
    #Shift these up to the high-order bits
    rldicr 7, 7, 32, 31
    #Load in the low-order pieces of the address
    oris 7, 7, second_value@h
    ori  7, 7, second_value@l
    #Load in second value to register 5, from the address we just loaded
    ld 5, 0(7)
    #Calculate the value and store into register 6
    add 6, 4, 5
    #Exit with the status
    li 0, 1    #system call is in register 0
    mr 3, 6    #Move result into register 3 for the system call
```

Before discussing the program itself, let's build it and run it. The first step in building this program is to **assemble** it:

```
as -a64 sum.s -o sum.o
```

This produces a file called `sum.o` containing the **object code**, which is the machine-language version of your assembly code, plus additional information for the linker. The "-m64" switch tells the assembler that you are using the 64-bit ABI as well as 64-bit instructions. The generated object code is the machine-language form of the code, however, it cannot be run directly as-is. It needs to be **linked** it so that it is ready for the operating system to load it and run it. To link, do the following:

```
ld -melf64ppc sum.o -o sum
```

This will produce the executable `sum`. To run the program, do:

```
./sum
echo $?;
```

This will print out "3", which is the final result. Now let's look at how the code actually works.

Since assembly language code works very close to the level of the operating system itself, it is organized very closely to the object and executable files which it will produce. So, to understand the code, we first need to understand object files.

Object and executable files are divided up into "sections". Each section is loaded into a different place in the address space when the program is executed. They have different protections and purposes. The main sections we will concern ourselves with are:

- **.data**: contains the pre-initialized data used for the program
- **.text**: contains the actual code (historically known as the program text)
- **.opd**: contains "official procedure declarations" which are used to assist in linking functions and specifying the entry point to a program (the entry point is the first instruction in the code to be executed)

The first thing our program does is switch to the `.data` section, and set alignment to an 8-byte boundary (.align 3 advances the assembler's internal address counter until it is a multiple of $2^3$).

The line that says `first_value:` is a symbol declaration. This creates a symbol called `first_value` which is synonymous with the address of the next declaration or instruction listed in the assembler. Note that `first_value` itself is a **constant**, not a variable, though the memory address it refers to may be updateable. `first_value` is simply an easy way to refer to a specific address in memory.
The next directive, `.quad 1`, creates an 8-byte data value, holding the value 1.

After this, we have a similar set of directives defining the address `second_value` holding an 8-byte data item with a value of 2.

The `.section "opd", "aw"` creates an "opd" section for our procedure descriptors. We force the section to align at an 8-byte boundary. We then declare the symbol `_start` to be global, which means that it will not be discarded after linking. After that, the `_start` symbol itself is declared (the .globl assembler does not define _start, it only marks it to be global once it is defined). The next three data items generated are the procedure descriptor, which will be discussed in a later article.

Now we can switch to actual program code. The `.text` directive tells the assembler that we are switching to the "text" section. After this is where `_start` is defined.

The first set of instructions loads in the address of the first value (not the value itself). Because PowerPC instructions are only 32-bits long, there are only 16 bits available within the instruction for loading constant values (remember, the address of `first_value` is constant). Therefore, since the address can be up to 64 bits, we have to load it a piece at a time (part 2 of this series will show how to avoid this). @-signs within the assembler instruct the assembler to give a specially-processed form of a symbol value. The following are used here:

- @highest refers to bits 48-63 of a constant
- @higher refers to bits 32-47 of a constant
- @h refers to bits 16-31 of a constant
- @l refers to bits 0-15 of a constant

The first instruction used stands for "load immediate shifted". This loads the value on the far right side (bits 48-63 of `first_value`), shifts the number to the left 16 bits, and then stores the result into register 7. Bits 16-31 of register 7 now contain bits 48-63 of the address. Next we use the "or immediate" instruction to perform a logical or operation with register 7 and the value on the right side (bits 32-47 of `first_value`), and store the result in register 7. Now bits 32-47 of the address are in bits 0-15 of register 7. Register 7 is now shifted left 32 bits, with bits 0-31 cleared out, and the result is stored in register 7. Now bits 32-63 of register 7 contain bits 32-63 of the address we are loading. The next two instructions use "or immediate" and "or immediate shifted" instructions to load bits 0-31 in a similar manner.

That's quite a lot of work just to load a single 64-bit value. That's why most operations on PowerPC chips operate through registers instead of immediate values -- register operations can use all 64-
bits at once, rather than being limited by the instruction length. In the next article we will cover the addressing modes that make this easier.

Now, remember, this only loads in the address of the value we want to load. Now we want to load the value itself into a register. To do this, we will use register 7 to tell the processor what address we want to load the value from. This will be indicated by putting "7" in parentheses. The instruction \texttt{ld 4, 0\textregistered7} loads the value at the address in register 7 into register 4 (the zero means to add zero to that address). Now register 4 has the first value.

A similar process is used to load the second value into register 5.

After the registers are loaded, we can now add our numbers. The instruction \texttt{add 6, 4, 5} adds the contents of register 4 to register 5, and stores the result in register 6 (registers 4 and 5 are unaffected).

Now that we have computed the value we want, the next thing we want to do is use this value as the return/exit value for the program. The way that you exit a program in assembly language is by issuing a system call to do so (exiting is done using the exit system call). Each system call has an associated number. This number is stored in register zero before making the call. The rest of the arguments start in register three, and continue on for however many arguments the system call needs. Then the \texttt{sc} instruction causes the kernel to take over and respond to the request. The system call number for \texttt{exit} is 1. Therefore, the first thing we need to do is to move the number 1 into register 0.

On PowerPC machines, this is done by adding. The \texttt{addi} instruction adds together a register and a number and stores the result in a register. In some instructions (including addi), if the specified register is register 0, it doesn't add a register at all, and uses the number 0 instead. This seems confusing, but the reason for it is to allow PowerPCs to use the same instruction for adding as for loading.

The exit system call takes one parameter -- the exit value. This is stored in register 3. Therefore, we need to move our answer from register 6 to register 3. The "register move" instruction \texttt{rm 3, 6} performs the needed move. Now we are ready to tell the operating system we are ready for it to do its trick.

The instruction to call the operating system is simply \texttt{sc} for "system call". This will invoke the operating system, which will read what we have in register 0 and register 3, and then exit with the contents of register 3 as our return value. On the command line, we can retrieve this value using the command \texttt{echo $?}.

Just to point out, a lot of these instructions are redundant, but used for teaching purposes. For example, since \texttt{first\_value} and \texttt{second\_value} are essentially constant, there is no reason we can't just load them directly and skip the data section altogether. Likewise, we could have stored the results in register 3 to begin with (instead of register 6), and saved a register move. In fact, we could have used register 3 for both a source and a destination register. So, if we were just trying to be concise, we could have written it like this:
Listing 2. A short version of the first program

```assembly
.section ".opd", "aw"
.align 3
.global _start
_start:
.quad _start, .TOC.@tocbase, 0
.text
li 3, 1   #load "1" into register 3
li 4, 2   #load "2" into register 4
add 3, 3, 4    #add register 3 to register 4 and store the result in register 3
li 0, 1   #load "1" into register 0 for the system call
sc
```

Finding a maximum value

For our next program, we will have a program that is a little more functional -- we will find a maximum of a set of values and exit with the result.

Here is the program; enter it as max.s:

Listing 3. Finding a maximum value

```assembly
###PROGRAM DATA###
.data
.align 3
#value_list is the address of the beginning of the list
value_list:
    .quad 23, 50, 95, 96, 37, 85
#value_list_end is the address immediately after the list
value_list_end:
###STANDARD ENTRY POINT DECLARATION###
.section ".opd", "aw"
.global _start
.align 3
_start:
    .quad _start, .TOC.@tocbase, 0
###ACTUAL CODE###
.text
_start:
    #REGISTER USE DOCUMENTATION
    #register 3 -- current maximum
    #register 4 -- current value address
    #register 5 -- stop value address
    #register 6 -- current value
    #load the address of value_list into register 4
    lis 4, value_list@highest
    ori 4, 4, value_list@higher
    rldicr 4, 4, 32, 31
    oris 4, 4, value_list@h
    ori 4, 4, value_list@l
    #load the address of value_list_end into register 5
    lis 5, value_list_end@highest
    ori 5, 5, value_list_end@higher
    rldicr 5, 5, 32, 31
    oris 5, 5, value_list_end@h
    ori 5, 5, value_list_end@l
    #initialize register 3 to 0
    li 3, 0
```
# MAIN LOOP

loop:
    # compare register 4 to 5
    cmpd 4, 5
    # if equal branch to end
    beq end

    # load the next value
    ld 6, 0(4)

    # compare register 6 (current value) to register 3 (current maximum)
    cmpd 6, 3
    # if reg. 6 is not greater than reg. 3 then branch to loop_end
    ble loop_end

    # otherwise, move register 6 (current) to register 3 (current max)
    mr 3, 6

loop_end:
    # advance pointer to next value (advances by 8-bytes)
    addi 4, 4, 8
    # go back to beginning of loop
    b loop

dering 4, 5

end:
    # set the system call number
    li 0, 1
    # register 3 already has the value to exit with
    # signal the system call
    sc

To assemble, link, and run the program, do:

```
as -a64 max.s -o max.o
ld -melf64ppc max.o -o max
./max
echo $?```

Hopefully now that you have experience with one PowerPC program and know a few instructions, you can follow the code a little bit. The data section is the same as before, except that we have several values after our `value_list` declaration. Note that this does not change `value_list` at all -- it is still a constant referring to the address of the first data item that immediately follows it. The data after it is using 64-bits (signified by `.quad`) per value. The entry point declaration is the same as previous.

In the program itself, one thing to notice is that we documented what we were using each register for. This practice will help you immensely to keep track of your code. Register 3 is the one we are storing the current maximum value in, which we initially set to 0. Register 4 contains the address of the next value to load. It starts out at `value_list` and advances forward by 8 each iteration. Register 5 contains the address immediately following the data in `value_list`. This allows a simple comparison between register 4 and register 5 to know when we are at the end of the list, and need to branch to `end`. Register 6 contains the current value loaded from the location pointed to by register 4. In every iteration it is compared with register 3 (current maximum), and register 3 is replaced if register 6 is larger.
Note that we marked each branch point with its own symbolic label, which allowed us to use those labels as the targets for branch instructions. For example, `beq end` branches to the code immediately following the `end` symbol definition later on in the code.

Another instruction to note is `ld 6, 0(4)`. This uses the contents of register 4 as a memory address from which to retrieve a value, which it then stores into register 6.

**Conclusion**

Hopefully, you now have a basic feel for assembly language programming on the PowerPC. The instructions may look a little weird at first, but they will become second-nature with practice. In the next article, I'll cover the various addressing modes of both the PowerPC processor, and how they can be used to do 64-bit programming more effectively. In the third article, we will cover the ABI more fully, discussing what registers are used for what purpose, how to call and return from functions, and other interesting aspects of the ABI.
Related topics

- Read all articles in this series.
- Read all of Jon Bartlett's articles on developerWorks.
- To learn all the details of the instructions available on POWER5 processors, check out the PowerPC Architecture Books.
- For the nitty gritty on how the PowerPC interfaces with the operating system and the C programming language, read the 64-bit PowerPC ELF Application Binary Interface Supplement 1.9.
- For an introduction to 32-bit PowerPC programming on Linux, see this guide to assembly language programming on Mac Minis.
- To learn more about IBM's Cell Processor, see the Cell Broadband Engine resource center.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

© Copyright IBM Corporation 2006
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)