

---

# Programming high-performance applications on the Cell BE processor, Part 2: Program the synergistic processing elements of the Sony PLAYSTATION 3

## An overview of the SPEs

Skill Level: Intermediate

[Jonathan Bartlett \(johnnyb@eskimo.com\)](mailto:johnnyb@eskimo.com)  
Director of Technology  
New Medio

07 Feb 2007

Take even greater advantage of the synergistic processing elements (SPEs) of the Sony® PLAYSTATION® 3 (PS3) in this installment of [Programming high-performance applications on the Cell BE processor](#). Part 1 showed how to install Linux® on the PS3 and explored a short example program. Part 2 looks in depth at the Cell Broadband Engine™ processor's SPEs and how they work at the lowest level.

The [previous article](#) in this series gave an overview of the Cell Broadband Engine (Cell BE) processor. (For other overviews, see [Resources](#) at the end of this article.) Part 2 begins an in-depth discussion of the Cell BE chip's SPEs. (For an in-depth discussion about programming the Power processing element (PPE), see the [Assembly language for Power Architecture series](#) on the developerWorks Linux zone.) Because the SPEs use such a different architecture, it's helpful to look at them in assembly language to get the full feel for what is happening. Later, I will show you how to program them in C, but assembly language gives a better view of the distinctiveness of the processor. Then, when moving to C, you will understand how different coding decisions might affect performance. This article focuses on the basic syntax and usage of SPE assembly language, and the ABI (the *application binary interface*, or the function calling conventions of the platform). The next two articles will explore communication between the SPE and the PPE and how to use the unique features of the SPE's assembly language to optimize your code.

As the [previous article](#) mentioned, the Cell BE chip consists of a PPE which has

several SPEs. The PPE is responsible for running the operating system, resource management, and input/output. The SPEs are responsible for data processing tasks. The SPEs do not have direct access to main memory, but only a small (256K on the PS3) *local store* (LS) which is in an independent, 32-bit address space. An address within the local store's address space is called a *local store address* (LSA), while an address within the controlling process on the PPE is called an *effective address* (EA). The SPEs include an attached *memory flow controller* (MFC). The SPEs use the MFC to transfer data between the local store, main memory, and other SPEs.

The synergistic processing unit (SPU) is the part of the SPE which actually runs your code. The SPU has 128 general-purpose registers, each 128 bits wide. However, the point of the SPU is not to do operations on 128-bit values. Instead, the processor is a *vector* processor. This means that each register is divided into multiple, smaller values, and instructions operate on all of the values simultaneously. Normally, the registers are treated as four distinct 32-bit values (32 bits is considered the word size on the SPUs), though they can also be treated as sixteen 8-bit values (bytes), eight 16-bit values (halfwords), two 64-bit values (doublewords), or as a single 128-bit value (quadword). The code in this article is actually non-vector (also known as *scalar*) code, meaning that it only works with one value at a time. It will use some vector operations, but we will only be concerned with one value within each register -- the others we will simply ignore. Later articles in [this series](#) will deal with vector operations.

This article does not require that you be experienced with assembly language, though it would be helpful. Some features of the SPE will be compared and contrasted with the features of the PPE, but knowledge of the PPE is also not required. For a discussion of the features of the PPE which are based on the Power Architecture, see the [Assembly language for Power Architecture](#) series.

The build commands in this article assume that you have Yellow Dog Linux installed according to the instructions in [Part 1](#). If you are using another distribution, some of the command names and flags may change. For example, if you are using the IBM System Simulator from the 1.2 SDK (IBM has released the 2.0 SDK, but the 1.2 SDK is what ships with YDL), then you should change all `gcc` references to `ppu-gcc` and all `embedspu` references to `ppu-embedspu`. Depending on where the libraries and header files are installed, additional flags might also need to be passed to find them.

## A simple example program

To begin looking at SPU assembly language, I will enter in a simple program for calculating the factorial of a 32-bit number using a recursive algorithm. The recursive nature of the algorithm will help to illustrate the standard ABI.

For reference, here is the C code which would perform the same function:

### Listing 1. C version of factorial program

```
int number = 4;
int main() {
```

```

        printf("The factorial of %d is %d\n", number, factorial(number);
    }

    int factorial(int num) {
        if(num == 0) {
            return 1;
        } else {
            return num * factorial(num - 1);
        }
    }
}

```

Now I'll enter in the assembly language version of this program and then discuss what each line means. Don't be taken aback by the size of the code -- it's mostly comments and declarations (the factorial function itself only has 16 instructions). Enter the following as `factorial.s`.

## Listing 2. First SPE program

```

###DATA SECTION###
.data

##GLOBAL VARIABLE##
#Alignment is _critical_ in SPU applications.
#This aligns to a 16-byte (128-bit) boundary
.align 4
#This is the number
number:
    .long 4

.align 4
output:
    .ascii "The factorial of %d is %d\n\0"

##STACK OFFSETS##
#Offset in the stack frame of the link register
.equ LR_OFFSET, 16
#Size of main's stack frame (back pointer + return address)
.equ MAIN_FRAME_SIZE, 32
#Size of factorial's stack frame (back pointer + return address + local variable)
.equ FACT_FRAME_SIZE, 48
#Offset in the factorial's stack frame of the local "num" variable
.equ LCL_NUM_VALUE, 32

###CODE SECTION###
.text

##MAIN ENTRY POINT
.global main
.type main,@function
main:
    #PROLOGUE#
    stqd $lr, LR_OFFSET($sp)
    stqd $sp, -MAIN_FRAME_SIZE($sp)
    ai $sp, $sp, -MAIN_FRAME_SIZE

    #FUNCTION BODY#
    #Load number as the first parameter (relative addressing)
    lqr $3, number

    #Call factorial
    brsl $lr, factorial

    #Display Factorial
    #Result is in register 3 - move it to register 5 (third parameter)
    lr $5, $3
    #Load output string into register 3 (first parameter)
    ila $3, output
    #Put original number in register 4 (second parameter)

```

```

    lqr $4, number
    #Call printf (this actually runs on the PPE)
    brsl $lr, printf

    #Load register 3 with a return value of 0
    il $3, 0

    #EPILOGUE#
    ai $sp, $sp, MAIN_FRAME_SIZE
    lqd $lr, LR_OFFSET($sp)
    bi $lr

##FACTORIAL FUNCTION
factorial:
    #PROLOGUE#
    #Before we set up our stack frame,
    #store link register in caller's frame
    stqd $lr, LR_OFFSET($sp)
    #Store back pointer before reserving the stack space
    stqd $sp, -FACT_FRAME_SIZE($sp)
    #Move stack pointer to reserve stack space
    ai $sp, $sp, -FACT_FRAME_SIZE
    #END PROLOGUE#

    #Save arg 1 in local variable space
    stqd $3, LCL_NUM_VALUE($sp)
    #Compare to 0, and store comparison in reg 4
    ceqi $4, $3, 0
    #Do we jump? (note that the "zero" we are comparing
    #to is the result of the above comparison)
    brnz $4, case_zero

case_not_zero:
    #remove 1, and use it as the function argument
    ai $3, $3, -1
    #call factorial function (return value in reg 3)
    brsl $lr, factorial
    #Load in the value of the current number
    lqd $5, LCL_NUM_VALUE($sp)
    #multiply the last factorial answer with the current number
    #store the answer in register 3 (the return value register)
    mpyu $3, $3, $5

    #EPILOGUE#
    #Restore previous stack frame
    ai $sp, $sp, FACT_FRAME_SIZE
    #Restore link register
    lqd $lr, LR_OFFSET($sp)
    #Return
    bi $lr

case_zero:
    #Put 1 in reg 3 for the return value
    il $3, 1
    ##EPILOGUE##
    #Restore previous stack frame
    ai $sp, $sp, FACT_FRAME_SIZE
    #Return
    bi $lr

```

To build the program, just use the C compiler:

```
spu-gcc -o factorial factorial.s
```

Now the Cell BE processor does not run SPE programs directly. It actually requires that the main code be written so that the PPE manages the resources. However, if Linux is given a program written for the SPE by itself and the `elfspe` package is properly installed, Linux will auto-create a minimal PPE process to manage the

resources for the SPE and act as a supervisor for the SPE process. Therefore, if you have the `elfspe` package installed, you can run the SPE program normally:

```
./factorial
```

If that doesn't work, be sure that the `elfspe` package is installed appropriately (see the [previous article](#) for instructions).

Now take a look at what each instruction and declaration does.

The program starts off with a typical `.data` declaration. In assembly language, the static data and global variables are separated out in memory from the code. You can switch back and forth between data and code sections, but when the program is assembled it will bring all of each section together into one unit. `.data` switches into the data section, while `.text` switches into the code section.

The data section holds the value we want to compute the factorial of in a space labeled `number`. Putting a string literal at the beginning of a line with a colon after it indicates that the address of the following declaration or instruction can be referred to throughout the program by that label. So, throughout the code, anywhere you see `number`, it will refer to the address of the next value. `.long` is a declaration which stores a value in a 32-bit space. In this case, we are storing the number 4.

Note, however, that before defining `number`, you align it using `.align 4`. The `.align` operation tells the assembler to align the next instruction or declaration at a certain boundary. `.align 4` aligns the next memory location to a 16-byte ( $2^4$ ) boundary. This is critical, as the SPU can only load exactly 16 bytes at a time, aligned to exactly a 16-byte boundary. If it is given an address to load from that is not at a 16-byte boundary, it simply zeroes out the last four bits of the address before loading it so that it will be an aligned load. Therefore, if your value is not properly aligned, it could be loaded *anywhere* within the register -- probably somewhere in the register that you don't expect. By aligning it to a 16-byte boundary, you know that it will load into the first four bytes of the register. After that is another alignment statement for the beginning of the string that gives your output. The `.ascii` declaration tells the assembler that what follows is an ASCII string, which is explicitly terminated with a `\0`.

After this, you define several constants for your stack frames. Remember that when a program makes a function call (especially for recursive ones), it has to store its return address and local variables on the stack. In C and other high-level languages, the language itself manages the run-time stack. In assembly language, this is handled explicitly by the programmer. The stack is set up for you by the operating system when the program starts. The stack starts at the high-numbered addresses of this region, and grows toward the low-numbered addresses as stack frames are added. You have to allocate space for each stack and move the appropriate values to that space. In this program you will have two stack frame sizes -- one for `main` and one for `factorial`. Each stack frame holds a pointer to the previous stack frame (called the *back chain pointer*), as well as a space for return addresses for when it calls other functions. While each of these is only a word size (4-byte) value,

they are each aligned to 16 bytes for easy loading and storing (remember, the SPUs only load from 16-byte-aligned addresses). The remaining space is used for saving registers and storing local variables. `main`'s stack will be the minimum of 32 bytes, while `factorial`'s will be 48, because `factorial` has a local variable to store. To name these quantities within the program and make the code more readable, we give these values symbols through the `.equ` operation. This tells the assembler to equate the given symbol with the given value. The stack frame sizes are assigned to the symbols `MAIN_FRAME_SIZE` and `FACT_FRAME_SIZE`, respectively. `LR_OFFSET` is the offset into the stack frame of the return address. `LCL_NUM_VALUE` is the stack offset of the local variable `num`. These will all be used to make access to stack frame offsets much clearer in the main body of code.

In the code section, you define a function's address the same way you defined addresses for global variables above -- just put their name followed by a colon. This indicates that the function's address will be the address of the next instruction. You use `.type` to tell the linker that this value should be used as a function, and you use `.global` to tell the linker that this symbol can be referenced outside of the current file when linking. `main` must be declared global, because it is used as the entry point to the program. Next, I'll go into the actual assembly instructions themselves.

I will discuss what the prologue does when we examine the `factorial` function. For right now, just know that it sets up the stack frame.

The first actual instruction you hit is `lqr $3, number`. This stands for "load quadword relative." The "quadword" part is a bit redundant, as only quadword loads and stores are allowed on the SPU. This loads the value in the address `number` (encoded as a relative address from the current instruction) into register 3. Unlike PPE assembly language, in SPE assembly language registers are always prefixed with a dollar sign. This makes it much easier to spot registers in the code. Since all registers on the SPU are 16-bytes long, this will load a full 16-byte quadword into the register, even though you are only really concerned with the first 4 bytes of it.

What you want to do with this value in register 3 is to calculate the factorial of it. Therefore, you need to pass it as the first (and only) parameter to the `factorial` function. THE SPU ABI, like the PPU ABI, uses registers to pass values to functions. Register 3 should hold the first parameter, register 4 should hold the second one, and so on. Therefore, the value you loaded into register 3 is already in the perfect spot for the function. Although registers can hold multiple values (in this case, four 32-bit values), when passing parameters to a function, each parameter value is passed in its own register.

This brings up the question, what are registers used for? If you've never programmed assembly language before, registers are the temporary storage that processors use for computing values. Since the SPU has 128 registers, it can keep a lot of temporary and intermediate values around without having to load and store back into memory like other architectures. This makes for both easier programming and faster execution. While the SPU makes no distinction in how registers are used, the ABI standard does. Here is a table of how the ABI uses each register within the SPU:

## Register usage in the SPU ABI

Register Range	Type	Purpose
0	Dedicated	Link Register
1	Dedicated	Stack Pointer
2	Volatile	Environment Pointer (for languages that need one)
3-79	Volatile	Function arguments, return values, and general usage.
80-127	Non-volatile	Used for local variables. Must be preserved across function calls.

I will get to the link register shortly, but basically it is used for temporary storage of return addresses. The stack pointer tells you where the end of your current stack frame is. The environment pointer is not used in most languages. All of the registers marked *volatile* can be freely changed within a function. However, that means that when a function makes a function call, it should expect that all of the values in volatile registers will be overwritten. All of the registers marked *non-volatile* must have their previous value saved before use and restored before returning from a function call. This allows you to have a set of registers which can be counted on to be preserved across function calls. However, they take more work to use, since your code must save and restore their previous values. The return value comes back in register 3.

Since you want the factorial of the number 4, it goes into register 3, the register used for the first parameter. You then branch to the function using `brsl $lr, factorial`. `brsl` stands for "branch relative and set link." This branches to the function entry point and sets the *link register* (LR) to the next instruction for the return address. Note that when you do `brsl`, you specify `$lr` for the register. This is an alias for `$0`. Notice also that you have to specify the link register explicitly. The SPU has no special registers. The link register is only special by convention -- the SPU assembly language allows you to set the link in *any* register you choose. However, for most purposes, this will be `$lr`.

After computing the factorial, you now want to print it out using `printf`. The first parameter to `printf` is the address of an output string. Therefore, you first need to move the result from register 3 to register 5 (register 4 will hold the original number). Then you need to move the address `output` into register 3. `ila` is a special load instruction that loads static addresses, in this case loading the address of the output string into register 3. It loads 18-bit unsigned values, which is the perfect size for local store addresses on the PS3. Finally, the original number is loaded into register 4. The `printf` function is called using `brsl $lr, printf`. Please note, however, that `printf` *is not executed on the SPE* because the SPE is incapable of input and output. This actually goes to a stub function which stops the SPE processor, signals the PPE, and the PPE actually performs the function call. After that, control is returned to the SPE.

The epilogue will be discussed in the analysis of the `factorial` code, but it basically just takes down the stack frame and returns to the previous function.

Before moving into a discussion of the `factorial` function, look at the layout of a stack frame more closely. Here is how the stack is supposed to be laid out according to the ABI:

Contains	Size	Beginning Stack Offset
Register Save Area	Varies (multiple of 16 bytes)	Varies
Local Variable Space	Varies (multiple of 16 bytes)	Varies
Parameter List	Varies (multiple of 16 bytes)	32(\$sp)
Link Register Save Area	16 bytes	16(\$sp)
Back Chain Pointer	16 bytes	0(\$sp)

The back chain pointer points to the back chain pointer of the previous stack frame. The link register save area holds the link register contents of the function being called, rather than for the current function. The parameter list is for parameters that this function sends to other function calls, not for its own parameters. However, unlike the PPE, this is only used if the number of parameters is greater than the number of registers available for parameters (not a very likely scenario). The local variable space is used as a general storage area for the function, and the register save area is used to save the values of non-volatile registers that the function uses.

So, in this function, we are using the back chain pointer, the link register save area, and one local variable. That gives a frame size of  $16 * 3 = 48$  bytes. As I mentioned earlier, `LR_OFFSET` is the offset from the end of the stack to the link register save area. `LCL_NUM_VALUE` is the offset from the end of the stack to the local variable `num`.

The *prologue* sets up the stack frame for the function. In the prologue, the first thing you do is save the link register. Since you have not yet defined your own stack frame, the offset is from the end of the calling function's stack frame. Remember that the link register is stored in the calling function's stack frame, not the function's own stack frame. Therefore, it makes most sense to save it before reserving the stack space. This is done using what is called a D-Form store (D-Form is an instruction format). Find an overview of common PPU instruction formats in [Assembly language for Power Architecture, Part 2](#) (the SPU formats follow the PPU formats fairly closely). The code for the store instruction is `stq d $lr, LR_OFFSET($sp)`. `stq d` stands for "store quadword D-Form." D-Form instructions take a register as the first operand, which is the register to be stored or loaded into, and a combination of a constant and a register as the second operand. The constant gets added to the register to compute the address to use for loading or storing. The other popular formats are the X-Form, which takes two registers which are added together, or the A-Form, which can hold a constant or a constant relative offset address. So in this instruction, `$sp` is the stack pointer (it's an alias for `$1`). The expression `LR_OFFSET($sp)` calculates the value of `LR_OFFSET` plus `$sp` and uses it as the destination address. So this instruction stores the link register (which holds the return address) into the proper location in the calling function's stack frame.

Next, the current stack frame pointer is stored as the back pointer for the next stack frame, even though you haven't established the stack frame yet (this is done through negative offsets). The SPU does not have an atomic store/update instruction like the PPU, so to make sure that the back pointers are always consistent, you must always store the back pointer *before* moving the stack pointer. Finally, the stack pointer is moved to reserve all the needed stack space using the instruction `ai $sp, $sp, -FRAME_SIZE`. `ai` stands for "add immediate," and it adds an immediate-mode value to a register and stores it back into a register. It adds together the register in the second operand with the constant in the third operand, and stores the result in the register specified in the first operand. Most instructions follow a similar format, with the register that holds the result specified in the first operand.

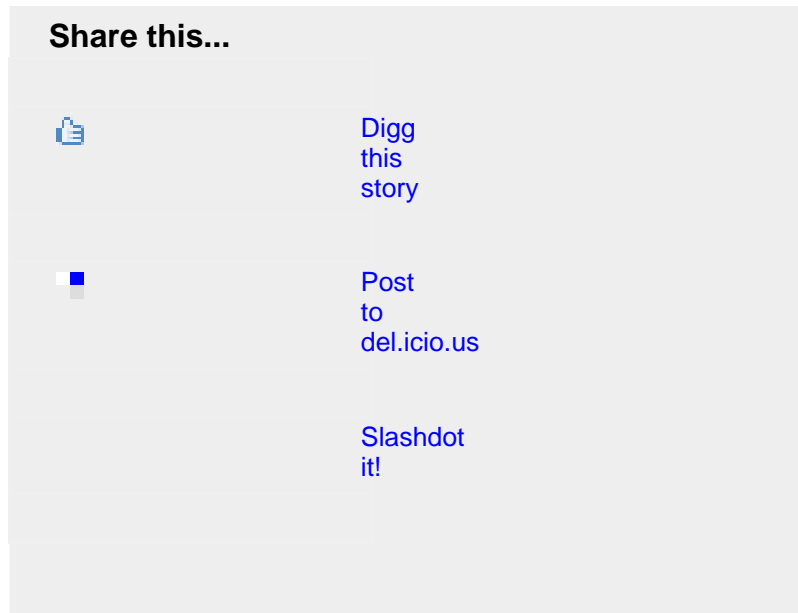
Note that the "add immediate" instruction is a vector operation. Remember that the SPU registers are 128 bits wide, but our value is only 32 bits long. The register is treated logically as multiple values, which are operated on all at once. The "add immediate" instruction actually treats the register as four separate 32-bit values, each of which have `-FRAME_SIZE` added to them, and then they are all stored back into the destination register. The preferred value size on the SPU is a 32-bit word, but others are supported, including bytes, halfwords, and doublewords. If the size of the operand is not specified in the instruction, that means either that the size doesn't matter (as in logical instructions, for instance) or that it is using a 32-bit value size. Bytes are indicated by including the letter `b` in the instruction, halfwords have an `h`, and doublewords have a `d`, though doublewords are usually only used in floating-point instructions (most often a `d` in an instruction refers to the D-Form of addressing, not a doubleword). But in this case we only care about the first word of the register. The other values simply do not matter in the ABI.

Next, you copy the first parameter to a local variable with `stqd $3, LCL_NUM_VALUE($sp)`. You need to do this because your parameter will get clobbered on the recursive function call, and you will need access to it afterwards.

Next, do an immediate-mode compare of register 3 with the number 0 and store the result in register 4 with `ceqi $4, $3, 0`. Note that with the PPU (and most processors for that matter), there is a special-purpose register to hold condition results. However, with the SPU, the results are stored in a general-purpose register -- register 4 in this case. Remember, this is a vector processor. So you are not actually comparing register 3 with the number 0. Instead, you are comparing each word of register 3 with the number 0. So you actually have four answers, even though you only care about one of them. The result is stored in the following way: if the condition for the word is true, then all of the bits on the destination word will be set; if the condition for the word is false, then all of the bits on the destination word will be unset. So for this instruction there will be four results, with each one being either all ones or all zeroes, depending on the results of the comparison.

The next instruction is `brnz $4, case_zero`. `brnz` stands for "branch relative if not zero." Remember, register 4 is the result of the previous comparison, so this is checking the previous compare result for zero or not-zero. The result register will be non-zero (or, true, all bits set to one) if the previous test for zero was true. Note that the previous two instructions could have been conflated into one instruction (`brz $3, case_zero`) since you were just testing for zero, but I separated them out into

two instructions so that you can better see how compares and branches work in the general case.



What happens if some of the comparisons have a result of true and others false? Since you are dealing with four 32-bit values rather than one 128-bit value, you could have different results for the different values. So if the results are different, do you branch or not? It turns out that several SPU instructions deal with only one of the register's values. In these cases, the value that is used is the one in the register's *preferred slot*. For 64-bit values it is the first half of the register; for 32-bit values the preferred slot is the first word of the register; for 16-bit values the preferred slot is the second halfword of the register; and for 8-bit values the preferred slot is the fourth byte of the register. Basically, the first word is the preferred word, and then the other alignments are on the least-significant byte or halfword of that word. When doing conditional branching, passing values to functions, returning a value from a function, and several other scenarios, the value in the preferred slot is the one that matters. In this case, you are to assume that the value passed in the function is in the register's preferred slot. And, if you look at the alignment of `number` in the `.data` section, you can see that this will be loaded into the preferred slot. Therefore, the branch will occur appropriately, as long as the value is in the preferred slot of the register.

Now assume that the number you are working with in register 3 is not zero. This means that you need to do a recursive step. The recursive C code is `return num * factorial(num - 1)`. The innermost computation requires decrementing `num` and passing it as a parameter to the next invocation of `factorial`. `num` is already in register 3, so you just need to decrement it. So, do an immediate-mode add like this: `ai $3, $3, -1`. Now, invoke the next `factorial`. To call a function according to the SPU ABI, all you need to do is put the parameters into registers, and then call `brsl $lr, function_name`. In this case, the first and only parameter is already loaded into register 3. So, you issue a `brsl $lr, factorial`. As I mentioned before, `brsl` stands for "branch relative set link." The

destination address is encoded as a relative address, the return address will be stored in the preferred slot of the specified register, and control will go to the destination address, which in this case is back to the beginning of the `factorial` function.

When control comes back to this point, the factorial result should be in register 3. Now you want to multiply this result by the current value under consideration. Therefore, you have to load it back in because it was clobbered in the function call. `lqd` stands for "load quadword D-Form." The first operand is the destination register and the second is the D-Form address to load. So `lqd $5, LCL_NUM_VALUE($sp)` will read the value that you saved on the stack earlier into register 5.

Now you need to multiply register 3 and register 5. This is done with the `mpyu` instruction (multiply unsigned). `mpyu $3, $3, $5` multiplies register 3 with register 5 and stores the result in the first register listed, register 3. Now, the integer multiply instructions on the SPU are somewhat problematic, especially signed multiplication (using the `mpy` instruction). The problem is that the result of a multiply instruction can be twice as long as its operands. The result of multiplying two 32-bit values is actually a 64-bit value! If it did this, then the destination register would have to be twice as large as the source register. To combat the problem, multiplication instructions only use the least-significant 16 bits of every 32-bit value so that the result will fit in the full 32-bit register. So, while the multiply treats the source registers as 32 bits wide, it only uses 16 bits of them. So, your value may be truncated if it is longer than 32 bits. And, if it is a signed multiply, the sign could even change on truncation! Therefore, to execute multiply instructions successfully, the source values need to be 16 bits wide, but stored in a 32-bit register (it doesn't matter for the multiplication if it is sign-extended to the rest of the 32 bits). This limits greatly the possible range of your factorial function. Note that floating-point multiplication doesn't have these issues.

So now you have the result, and it is in register 3, which is where it needs to be for the return value. All that is left to do is to restore the previous stack frame and return. So you simply need to move the stack pointer by adding the stack frame size to the stack pointer using `ai $sp, $sp, FRAME_SIZE`. Then restore the link register using `lqd $lr, LR_OFFSET($sp)`. Finally, `bi $lr` ("branch indirect") branches to the address specified in the link register (the return address), thus returning from the function.

The base case (what to do if the function's parameter is zero) is much easier. The result of `factorial(0)` is 1, so you simply load the number one into register 3 using `il $3, 1`. Then you restore the stack frame and return. However, since the base case doesn't call any other functions, you don't need to load the link register from the stack frame -- the value is still there.

And that's how the function works! Just note that writing deeply recursive functions on the SPE is problematic because there is no stack overflow protection on the SPE, and the local store is small to begin with.

## Conclusion

This article covered some of the main concepts of assembly language programming on the Cell BE processor of the PLAYSTATION 3 under Linux. Next time, I will examine the primary modes of communication between the SPE and the PPE.

# Resources

## Learn

- The details of every SPU instruction are available in the [SPU Instruction Set Architecture Reference](#) guide. However, most of the time you are better off looking at the short summaries in the [SPU Assembly Language](#) guide. In fact, to get a good overview of what the SPU can do, I suggest a read through the Assembly Language guide. It is both short and packed with information. If the instruction doesn't make sense, then look up the full definition in the full ISA document.
- For ABI details, see the [SPU ABI documentation](#) as well as the [Linux extensions to the ABI](#).
- The definitive source of information about the Cell BE processor itself is the [Cell BE Handbook](#).
- The IBM Semiconductor Solutions Technical Library [Cell Broadband Engine documentation](#) section lists specifications, user manuals, and more.
- Find all Cell BE-related articles, discussion forums, downloads, and more at the IBM developerWorks [Cell Broadband Engine resource center](#): your definitive resource for all things Cell BE.
- Keep abreast of all things Cell BE: subscribe to [IBM microNews](#).

## Get products and technologies

- Get Cell: [Contact IBM E&TS](#) for custom Cell BE-based or custom-processor based solutions.
- Get the [alphaWorks Cell Broadband Engine downloads](#) -- including the IBM Full System Simulator.
- Download the [SPE Runtime Management Library Version 1.2](#) from the Barcelona Supercomputing Center.

## Discuss

- Take part in the IBM developerWorks Power Architecture [Cell Broadband Engine discussion forum](#).
- Send a [letter to the editor](#).

## About the author

Jonathan Bartlett

Jonathan Bartlett is the author of the book [Programming from the Ground Up](#), an introduction to programming using Linux assembly language. He is the lead developer at New Media Worx, responsible for developing Web, video, kiosk, and desktop applications for clients.

## Trademarks

Sony and all Sony-based trademarks are trademarks of Sony Corporation of America.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc.