EXHIBIT MM
(VOL 2)

Linux development on the PlayStation 3, Part 3: Slimming down X11 with tiny tools

Renders great, less filling

Peter Seebach, Freelance author, Plethora.net

Summary: The Sony PlayStation 3 (PS3) runs Linux®, but getting it to run well requires some tweaking. In the third and final article of this series, on PS3 Linux, Peter Seebach talks about ways to get X11 slimmed down to fit on a smaller memory budget.

Date: 08 Apr 2008
Level: Intermediate
PDF: A4 and Letter (35KB | 8 pages)Get Adobe® Reader®
Also available in: Russian Japanese Portuguese

In Parts 1 and 2, you saw how to use the runlevel system and related tools to dramatically reduce memory usage on a PS3 running Linux, leaving more memory available for compilation and similar services. To wrap this up, let's look at a few more things that are worth doing, including at least one fairly hefty task: getting X11's footprint trimmed.

The problem with X11 is that you may well need it. This is a problem because the lowest memory footprint I saw for the X server itself on my PS3 was around 40MB, which is way, way, too much of available memory to sacrifice just for some pretty pictures. (Yes, I did learn to use UNIX® on a plain text terminal. Why are you looking at me funny?) That said, it really is very useful sometimes, and there are programs that don't make sense to run without graphics.

Run X11 on another machine

So you need X11, but the PS3 doesn't have enough memory to run it comfortably. You have one sneaky option: run over the network. X clients could be configured to run on your PS3, with an X server on a remote system. This isn't really exceptionally fast—the hypervisor seems to slow down networking a bit from the nominal gigabit performance the hardware has—but if you don't need the highest possible speed, it's actually pretty livable. You have a few options when setting this up. First, you're going to need to run an X server on another machine.

About this series

This series of three articles looks at PS3 Linux as a prospective development environment.

This first article, Part 1, introduces the basic configuration knobs and widgets specific to the PS3, shows you how to use them effectively, and suggests the kind of trickery that might get improved performance or a more usable display.

Part 2 and this article delve into some of the performance and tuning issues that, while they might apply on any system, are particularly useful for turning your PS3 from a proof-of-concept demo into a working system.

There's a never-ending rumor that OS X will run on a PS3, because one of the Cell/B.E. technology demo movies was once played on a Mac at a trade show, and people keep posting a picture of this and insisting that it's proof. Well, time to make that even worse; I'll use the native X11 server on an OS X machine to demonstrate this. In fact, X is nicely portable, and the essential tricks are the same everywhere.

First, start your X server, and get yourself a prompt; run a terminal program, of whatever sort you like. I picked xterm. Now, there are two ways forward. One is to use ssh's X tunneling feature to let you forward X requests from the client (that's the PS3) to the server (that's the Mac, in this case). The other is to use X directly over the wire. They each have strengths and weaknesses; in the case of a local network, over the wire may be easier to explain and use. First, figure out what your X server's DISPLAY is. In your X terminal, echo the environment variable DISPLAY; it'll probably be ":0.0". This means that your X server is running a display named "0.0". The colon separates the hostname from the display name. If your laptop were named "laptop," the full name of the display would be "laptop:0.0". (When the name is omitted, X uses clever hacks to access a local display faster.)

Wait, which one's the server?

In most of computing, the "server" is the program that doesn't have a direct interface to users, and the "client" is the program with the graphical interface. However, in X, this is reversed—and if that confuses you, don't feel too bad. A lot of users have been confused by it at first.

To understand why the graphical program is the "server," and the program doing all the computation is the "client," ask yourself the question: What is it that is being provided? The X server is providing a graphical display. Thus, the server handles incoming requests from clients such as "draw me some pixels" or "give me a window," and the clients use these services as they wish.

Now, if you were to just pop over to the PS3, and set the environment variable DISPLAY to, for instance, "laptop:0.0", and run an xterm, you'd discover a tragic flaw in this scheme: Permission denied. By default, X will not allow arbitrary clients on a remote host to run on the local display. This is a security feature; obviously, you'll want to subvert it. The simplest way to do this is to allow clients on your PS3 to access your X server. On the local machine, run the command `xhost +<machine>`, where `<machine>` is the hostname or IP address of your PS3. I never bothered to set up DNS for my unroutable dynamic block, so I used `xhost +10.10.10.134`. With that done, running X commands on the PS3 works; they show up on the Mac, without the substantial memory overhead of an X server running on the PS3.

Forward X11 requests

Now, there's another option. The `ssh` command (you did leave sshd running, right?) can forward X11 requests. When you run `ssh` with access to an X server, passing it the `-X` option causes it to forward X requests; in fact, the `-Y` option may be preferable, as it enables "trusted" forwarding, which bypasses more security features. (You want to bypass them because otherwise you may not be able to, say, open windows.) Optionally, you might look at the `-C` option, which specifies compression of data, including X11 packets. This is useful on slow networks, but not so useful on fast networks. Try it both ways to see. The syntax for this one is easier in some ways; just `ssh -X <ps3>`, and on the PS3, start running commands that use X. This way, the PS3 doesn't have the memory overhead of the server,

only that of the clients. Time to start konquerer, right? But konqueror takes up about 20MB, plus 7.5MB for kded, 5.8MB for klauncher, 5MB for kio_file, 4.7MB for kdeinit...

This, of course, highlights a secondary problem: even without the server, X apps can chew up a lot of memory. Offloading the server does improve available memory, but it doesn't completely eliminate the problem.

Try to trim the server

Remote access just may not be fast enough, or you may not have a convenient X server to use. There is another option: run the server locally, but without KDE or Gnome. To do this, you'll probably have to omit the runlevel 5 X display manager and login window; instead, log in on a console and start X yourself. The classic way to do this is to just run the `xinit` program, which runs the commands in your .xinitrc file, found in your home directory. Note that these commands are run sequentially; if you want more than one program at once, the first few need to be backgrounded (place an & at the end of the line). For instance, here's a .xinitrc that will do you well:

Listing 1. X server, plain, hold the mustard

```
xterm &
exec twm
```

X terminates when the program it was running as its "session" terminates; in this case, that's the twm program that the xinitrc exec'd as its last command. Some people prefer to use an xterm as the session program, and run the window manager in the background. It's up to you.

If you use that .xinit file, and run `xinit`, you'll likely see a grey stippled background, and a mouse cursor with a sort of odd nine-paneled, uh, thing. That's an xterm, waiting for you to decide where to place it. The twm window manager's primary feature is that it's tiny; it doesn't do much for you, it mostly trusts you to know what you want. On the other hand, its total memory footprint of a tad over 2MB is noticeably smaller than that of something larger, like Gnome or KDE. On my system, the xterm program actually took up more space than twm, although the bulk of the space went to the X server, with a virtual size of 62MB, and a resident set of 34MB actually in RAM. That's a bit large. What can we do about it?

Not much. The first obvious thing to do would be to disable modules. If you've ever had to do this before, you're doubtless familiar with the process. Go into /etc/X11, edit xorg.conf (you need root privileges to do this), and comment out the modules you don't need.... Only there aren't any modules in xorg.conf. In modern X.org servers, the default behavior is to load everything and see what sticks. Conveniently, it's easy to override this; if you provide a Modules section, only the things you ask for will be loaded. On a system with no functioning graphical hardware, the various GL-related options are pretty much worthless; here's what I ended up using:

Listing 2. Fewer modules

```
Section "Module"
        Load "extmod"
        Load "type1"
        Load "freetype"
EndSection
```

That doesn't actually change things much, I'm afraid, It got my X server from 34MB down to, wait for it, 33MB. The real problem, of course, is the huge amount of memory used by the internal buffer that's being written onto the framebuffer device: 1280x768 or so, and the framebuffer only works in 32-bit mode, really. Reducing the size of the framebuffer actually can help quite a bit—at substantial cost to your screen real estate, of course. For instance, switching to a 480p, non-fullscreen mode (which is substantially less than 480 pixels tall), I found that X was using only about 10MB of real memory; even 720p, non-fullscreen, used less (25MB) than I was getting in WXGA mode. If you're close to the edge, it might be worth scaling back a bit. It's just a shame the PS3 doesn't have a broader range of supported video modes.

Other ways to trim space

There's a little more you can do to trim space. If you aren't using the multiple login consoles of your PS3, you can turn off the ones you aren't using. Unlike other runlevel-related features, these are actually encoded directly in /etc/inittab. The default configuration for Fedora is to ship with six consoles enabled; I think the screen's more useful. Here's what you'd change in /etc/inittab to turn off five of them:

Listing 3. One console is enough, thank you

```
# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:off:/sbin/mingetty tty2
3:2345:off:/sbin/mingetty tty3
4:2345:off:/sbin/mingetty tty4
5:2345:off:/sbin/mingetty tty5
6:2345:off:/sbin/mingetty tty6
```

Changes to the runlevel files are recognized when you change to a new runlevel. What about changes to inittab? To let init know that you've changed inittab, run `/sbin/init q`. The `q` argument, rather than indicating a runlevel, tells init to reload its configuration files. (I think maybe it stands for query, but it's not documented.)

Shell size

While bash has many strengths—a feature set that's too large to explain, compatibility with lots of things, multiple different run modes for compatibility with different standards, and more—it does not run in a small amount of memory.

One alternative, useful for running scripts (although not ideal as an interactive shell) is the stripped down dash, a variant of Kenneth Almquist's shell clone (called ash) that was developed by the Debian maintainers. The goal of dash is to have a small and fast shell that can run scripts. It doesn't have every extra feature of bash, but it runs nearly all portable shell scripts, and does so in a small amount of memory, quickly.

On my test system, a typical bash invocation was running around 1.7MB of storage by the time it printed the first prompt; dash ran around 560KB. This can be especially useful if you can arrange for scripts or makefiles to use dash instead of bash; unfortunately, some programs will experience compatibility difficulties if /bin/sh isn't really bash. Still, if you need extra space, that's a place to go looking.

If you don't need regular background jobs, you can shut down anacron and crond; if you don't mind hard-coding your network settings, you can shut down dhclient. A particularly promising avenue might be to replace bash with dash (see Shell size). Still, you're pretty much at the margins here; past this, the only way to reclaim memory is to change the underlying structure of the ps3 framebuffer, or the hypervisor, and that's not practical.

On the other hand, starting with a system that had already begun to swap before you got a prompt, and ending with a system that has over 100MB free storage with a couple of shells, a top process, and sshd running, is no small feat. While it's easy to look down on this from the lofty heights of laptops with 2GB or more of memory, the fact is that an awful lot of software development becomes quite feasible when your development platform has 100MB of free memory; that's enough that many builds will be able to stay entirely within the buffer cache, saving you a great deal of time, to say nothing of being able to mostly avoid swapping except under extreme circumstances.

Conclusion

With these tweaks, the PS3 becomes a viable, and even sort of roomy, development environment. (It's one of the most accessible environments for someone looking to dabble with the Cell Broadband Engine.) It stays responsive during compiles, and compiles run noticeably faster than they used to. While the memory provided isn't so great for a desktop system, with video players and Web browsers and e-mail clients all running at once, it's certainly adequate for a development effort, and indeed, with the overhead of GNOME and KDE out of the way, it can even manage some desktop work. It's quite possible that future updates to the PS3 firmware will improve things, as might new kernel versions or drivers. Just remember to stay focused on what you need out of the system, and go ahead and disable features you aren't using.

Resources

Learn

- See all parts in the *Linux development on the PlayStation 3* series.

- "Programming high-performance applications on the Cell BE processor, Part 1: An introduction to Linux on the PLAYSTATION 3" (developerWorks, January 2007) and "PS3 fab-to-lab, Part 1: Build Linux lab equipment from a Sony PLAYSTATION 3" (developerWorks, May 2007) are the first articles in other developerWorks series about running Linux on the PS3.
- Twm is your good friend if you're in a tight spot, memory-wise.
- Embedded systems often use BusyBox to trim space requirements. BusyBox is more useful in terms of disk space than memory, but it helps with everything.
- Learn more than you ever needed to know about cron and anacron, and bash and dash, and these other fine Unix utilities from Wikipedia. For dhclient, see man dhclient.
- Learn more about ssh.
- The Cell Broadband Engine Resource Center is the definitive resource for all things Cell/B.E.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and community topics in our new developerWorks spaces.

About the author

Peter Seebach has been collecting video game consoles for years, but has only been running Linux on them recently. He is still not sure whether this is a Linux machine that plays video games, or a game machine that runs Linux.

The Open Platform feature is not available on CECH-2000 series or later models of the PS3™ system.

On PS3™ system models sold earlier than the CECH-2000 series models, the Open Platform feature will not be available if the system software is updated to version 3.21 or later.

Overview of the Open Platform for the PLAYSTATION®3 system

There is more to the PLAYSTATION®3 (PS3™) computer entertainment system than you may have assumed. In addition to playing games, watching movies, listening to music, and viewing photos, you can use the PS3™ system to run the Linux operating system.

By installing the Linux operating system, you can use the PS3™ system not only as an entry-level personal computer with hundreds of familiar applications for home and office use, but also as a complete development environment for the Cell Broadband Engine™ (Cell/B.E.).

There are many flavors of Linux available, which are developed, managed, and distributed by the respective companies and development communities.

As Sony Computer Entertainment Inc. (SCE) does not develop or directly support a version of Linux for the PS3™ system, SCE is pleased to provide links for the following Linux distributions that support the PS3™ system:

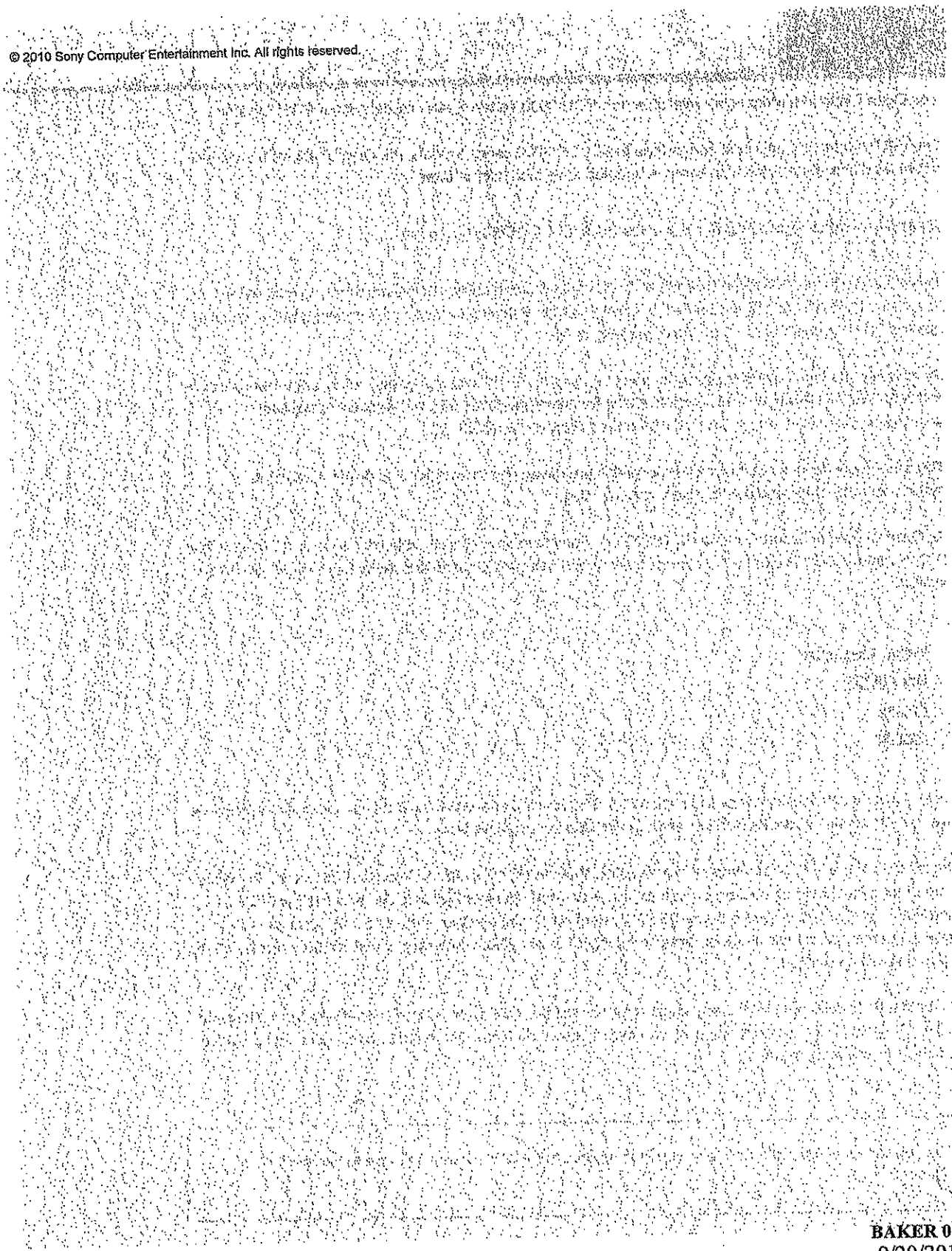
Yellow Dog Linux
OpenSUSE
Fedora
Ubuntu

The respective websites provide instructions for downloading or purchasing the Linux operating system, as well as information about installation and post-installation configuration.

Installation of the Linux operating system requires that the PS3™ system's internal hard disk be formatted. Important data should be backed up before proceeding with the installation. Even after the hard disk is formatted, the system software of the PS3™ system will not be deleted, and system features such as starting games will remain available after formatting the disk. (Note, however, that data that is saved on the hard disk will be deleted.)

Note that SCE does not provide any support for the installation and the use of Linux operating systems on a PS3™ system. For technical support, you must contact the Linux distributor or community that provided your Linux operating system.

To use the Linux operating system, you must update the PS3™ system software to version 1.60 or later.



The Open Platform feature is not available on CECH-2000 series or later models of the PS3™ system.

On PS3™ system models sold earlier than the CECH-2000 series models, the Open Platform feature will not be available if the system software is updated to version 3.21 or later.

Installing the boot loader and the Linux operating system

Installation of the Linux operating system on a PS3™ system varies depending on the Linux distribution. This page provides an example of how to install the boot loader and how to begin the Linux installation on the PS3™ system. For additional installation instructions, you must contact the Linux distributor or community that provided your Linux operating system.

Step 1: Obtain the boot loader

The Linux distribution you have selected may include a boot loader on a DVD. If not, you will need to obtain a boot loader from a Linux distribution website.

Currently there are two kinds of boot loaders: "kboot" and "petitboot". If the file name of the boot loader that you downloaded is "kboot-20080609.bld" or something similar to this name, you must rename it "otheros.bld" (all lowercase characters) so that it will be recognized by the PS3™ system.

Step 2: Prepare the storage or disc media that contains the boot loader

Use a PC to create a folder named "PS3" (all uppercase characters) on the storage or disc media. Next, create a folder named "otheros" (all lowercase characters) within the "PS3" folder.

Save the boot loader file as "otheros.bld" (all lowercase characters) in the "otheros" folder. The storage media can be either a USB flash drive or a CD/DVD. If you are using USB storage media, it must be formatted as a FAT file system. When the storage media is inserted in or connected to a PC that is running Windows, the file name for the folder in the media root directory should be "PS3\otheros\otheros.bld".

If you are using a PC that is running Mac OS X or Linux, the folder should be named "PS3otheros/otheros.bld". The boot loader will not be recognized if the folder names and file names are not exactly as specified above.

Step 3: Install the boot loader on the PS3™ system

Insert the USB storage media containing the boot loader in the USB connector on the PS3™ system. If the boot loader is saved on CD or DVD media, insert the disc in the PS3™ system's disc slot. Next, select [Settings] > [System Settings] > [Install Other OS] from the XMB™ menu.The system will search for the boot loader on the storage or disc media.

Follow the on-screen instructions to complete the installation. If the error message "No applicable installer was found" is displayed, check the folder names and file names in the storage or disc media.

Step 4: Set the PS3™ system to boot from the boot loader

Select [Other OS] in [Settings] > [System Settings] > [Default System] from the XMB™ menu, and then restart the PS3™ system. The system will start using the installed boot loader.

Step 5: Install the Linux operating system

Insert a bootable installation disc for the Linux operating system in the PS3™ system. The initial steps to install Linux may vary depending on the Linux distribution.

For example, for Fedora 10, you will need to type "linux noselinux video=720p" from the command line of the boot loader. Other Linux operating systems may only require pressing the Enter key. If both "linux32"and "linux64" appear on the installation menu, select "linux64" because the PS3™ system can execute only a 64-bit Linux kernel. For additional information on installing Linux, refer to the instructions provided with the distribution, or refer to the website for the associated community.

The Open Platform feature is not available on CECH-2000 series or later models of the PS3™ system.

On PS3™ system models sold earlier than the CECH-2000 series models, the Open Platform feature will not be available if the system software is updated to version 3.21 or later.

FAQ

Q. What do I need before I begin?

A. You must obtain a boot loader (otheros.bld) and a Linux installation disc. If you plan to download the files instead, you should prepare storage or disc media, such as a USB flash drive or a DVD-R/RW.

Q. Do I need a PC for the installation?

A. Yes. You will need a personal computer that is running Windows, Mac OS X, or Linux and a DVD burner to create the Linux disc from an ISO image (a bootable installation disc). Please note that at this time, the Web browser for the PS3™ system cannot be used to save downloaded content to the desired directory on the media.

Q. Do I need to install the boot loader (the otheros.bld file that prompts the PS3™ system to launch Linux) every time I start the [Other OS]?

A. No. Once the boot loader is correctly installed, the settings will remain even after you turn off the PS3™ system. Therefore, there is no need to install the boot loader every time. However, this is not the case when the boot loader file installed in the system's boot loader storage area has been damaged by an [Other OS], or by the boot loader itself. In that case, turn on the PS3™ system by holding down the power button for more than 10 seconds (until a beep is heard twice) to force the PS3™ system to start from the XMB™ screen. Then install the boot loader again.

Q. I can't find the [Default System] item on the XMB™ menu of the PS3™ system.

A. The [Default System] option will not be displayed until the boot loader is correctly installed in [Settings] > [System Settings] > [Install Other OS]. Follow the installation procedure provided on this website to complete the boot loader installation.

Q. I cannot go back to the XMB™ screen of the PS3™ system after using Linux.

A. The method for switching from Linux to the XMB™ screen of the PS3™ system depends on the version of the PS3™ Linux utility. The current method is to type "ps3-boot-game-os" from the command line. For additional instructions, contact the Linux distributor or associated community. Note that if you turn on the PS3™ system by pressing the power button for more than 10 seconds until a beep is heard twice, the XMB™ screen will be displayed by default from the next time you turn on the system.

Q. Can the entire internal hard disk of the PS3™ system be used by Linux?

A. No. Linux cannot use the entire hard disk because the [PS3™ System] partition must be maintained on the hard disk for use by the PS3™ system software.

Q. Can the boot loader be installed without preparing a partition for the [Other OS] on the PS3™ system's hard disk?

A. It depends on the type of boot loader you use. Typically, only "live Linux systems" (Linux that runs only in memory, from a DVD or from USB storage media) do not require use of the PS3™ system's hard disk. For details on the other boot loaders, contact the boot loader distributor.

Q. How do I uninstall Linux or its boot loader?

A. You can remove Linux by reformatting the PS3™ system's hard disk via [Settings] > [System Settings] > [Format Utility] > [Format Hard Disk] from the XMB™ menu. You cannot uninstall the boot loader, although it can be overwritten with another boot loader by following the installation instructions on this website. Also, selecting [PS3™] as the default system software in [Settings] > [System Settings] > [Default System] from the XMB™ menu will cause the boot loader to be ignored.

Programming high-performance applications on the Cell BE processor, Part 1: An introduction to Linux on the PLAYSTATION 3

Overview, installation, and first programming steps

Jonathan Bartlett (johnnyb@eskimo.com), Director of Technology, New Medio

Summary: The Sony® PLAYSTATION® 3 (PS3) is the easiest and cheapest way for programmers to get their hands on the new Cell Broadband Engine™ (Cell BE) processor and take it for a drive. Discover what the fuss is all about, how to install Linux® on the PS3, and how to get started developing for the Cell BE processor on the PS3.

Date: 03 Jan 2007
Level: Intermediate
Also available in: Korean Russian Japanese

The PLAYSTATION 3 is unusual for a gaming console for two reasons. First, it is incredibly more open than any previous console. While most consoles do everything possible to prevent unauthorized games from being playable on their system, the PS3 goes in the other direction, even providing direct support for installing and booting foreign operating systems. Of course, many of the game-related features such as video acceleration are locked out for the third-party operating systems, but this series focuses on more general-purpose and scientific applications anyway.

The real centerpiece for the PS3, however, is its processor -- the Cell Broadband Engine chip (often called the Cell BE chip). The Cell BE architecture is a radical departure from traditional processor designs. The Cell BE processor is a chip consisting of *nine processing elements* (note the PS3 has one of them disabled, and one of them reserved for system use, leaving seven processing units at your disposal). The main processing element is a fairly standard general-purpose processor. It is a dual-thread Power Architecture™ element, called the *Power Processing Element*, or PPE for short. The other eight processing elements, however, are a different story.

The other processing elements within the Cell BE are known as *Synergistic Processing Elements*, or SPEs. Each SPE consists of:

- A vector processor, called a Synergistic Processing Unit, or SPU
- A private memory area within the SPU called the *local store* (the size of this area on the PS3 is 256K)
- A set of communication channels for dealing with the outside world
- A set of 128 registers, each 128 bits wide (each register is normally treated as holding four 32-bit values simultaneously)

- A *Memory Flow Controller* (MFC) which manages DMA transfers between the SPU's local store and main memory

The SPEs, however, lack most of the general-purpose features that you normally expect in a processor. They are fundamentally incapable of performing normal operating system tasks. They have no virtual memory support, don't have direct access to the computer's RAM, and have extremely limited interrupt support. These processors are wholly concentrated on processing data as quickly as possible.

Therefore, the PPE acts as the resource manager, and the SPEs act as the data crunchers. Programs on the PPE divvy up tasks to the SPEs to accomplish, and then they feed data back and forth to each other.

Connecting together the SPEs, the PPE, and the main memory controller is a bus called the *Element Interconnect Bus*. This is the main passageway through which data travels.

The most surprising part of this design is that the SPE's 256K local store is not a cache -- it is actually the full amount of memory that an SPE has to work with at a time for both programs and data. This seems like a disadvantage, but it actually gives several advantages:

- Local store memory accesses are extremely fast compared to main memory accesses.
- Accesses to local store memory can be predicted down to the clock cycle.
- Moving data in and out of main memory can be requested asynchronously and predicted ahead of time.

Basically, it has all of the speed advantages of a cache. However, since programs use it directly and explicitly, they can be much smarter about how it is managed. It can request data to be loaded in before it is needed, and then go on to perform other tasks while waiting for the data to be loaded.

While the Cell BE processor has been out for a while in specialized hardware, the PS3 is the first Cell BE-based device that has been affordable and readily available. And, with Linux, anyone who wants to can program it.

It runs Linux? How do I get it on there?

It is unusual for gaming consoles to allow foreign operating systems to be installed on them. Since consoles are usually sold at a loss, they are usually locked down to prevent games from running on them without the publisher paying royalties to the console developer. Sony decided to open up the PS3 console a little bit, and allow third-party operating systems to be installed, with the caveat that they do not get accelerated graphics.

Because of this, you can now install Linux on the PS3. You have to jump through a few hoops, but it definitely works. Terra Soft Solutions has developed Yellow Dog Linux 5 in cooperation with Sony specifically for the PS3. It even offers, uniquely among distributions so far, support for those using it on PS3. Yellow Dog Linux (also known as YDL) has been an exclusively PowerPC-based distribution since its inception, so it was not surprising that Sony contracted it to develop the next version of YDL specifically for the PS3.

See below for instructions on installing the initial release of YDL 5 onto the PS3.

Preparing the PS3

To install Linux, you need several pieces of additional hardware:

- A display and appropriate cabling
- A USB keyboard
- A USB mouse
- A USB flash drive

On the display, there are a few gotchas to watch for. First of all, the 20GB PS3 only comes with an analog composite RCA plug for attaching to TV-like output devices. You can convert it to VGA through a special cable (see Resources for more information). Unfortunately, this operates only at 576x384. If you want better resolutions, you'll have to use the HDMI port. However, that can lead to additional problems. HDMI can be easily converted to DVI through a cable. So this should be able to be fed to a DVI-compatible monitor, right? Well, no. There is a content-protection protocol called HDCP. When outputting data over the HDMI port, the PS3 will not output any data to non-HDCP-compliant devices. Therefore, unless your monitor is HDCP-compliant, you cannot use it to get digital output from the PS3, and you're stuck with 576x384 (though some have reported higher resolutions using component video output rather than composite).

To prepare the PLAYSTATION 3, perform the following steps:

1. Connect the ethernet cable to the PS3. Be sure the network has a DHCP server on it.
2. If this is a fresh-from-the-factory PS3, go through the setup steps as it prompts you on your first bootup, including setting the language, time, and a username for the PS3 system.
3. Go to **Settings**, then **System Settings**, and choose **Format Utility**.
4. Select **Format Hard Disk**, and confirm your selection twice.
5. Select that you want a **Custom** partitioning scheme.
6. Select that you want to **Allot 10GB to the Other OS**. This will automatically reserve the remaining disk space for the PS3's game operating system. When finished, it will restart the system.
7. When the system restarts, go to **Settings** then **System Update**.
8. Choose **Update via Internet**.
9. Follow the screens for the system update to download and install the latest system updates. Some screens only have cancel buttons, with no instructions on how to move forward. In order to move forward on those screens, use the **X** button on your controller.
10. Once the PS3 restarts, it's ready to have Linux installed on it.

Preparing to install

Now you're ready to prepare the Linux side of things. Here are the steps you need to do on your own computer (not the PS3) to prepare for the installation:

1. Download and burn the YDL 5 DVD ISO. There is no CD install -- the PS3 only takes DVDs.
2. Download the PS3 OtherOS installer from Sony (see Resources) and save it as otheros.self. This is the file that runs on the PS3 game operating system to install foreign bootloaders. NOTE: otheros.self is no longer needed if you are using PS3 firmware 1.60 or higher.
3. Download the YDL bootloader from Terra Soft (again, see Resources) and save it as otheros.bld. This will be the bootloader that the Sony installer will install.
4. Insert a USB flash drive into your computer.
5. At the top level of your flash drive, create a directory called PS3. Immediately under the PS3 directory, create another directory called otheros.
6. Copy the last two files you downloaded, otheros.self and otheros.bld, into the PS3/otheros directory you just created on your flash drive.

Now it is time to install.

Performing the installation

Perform the following steps on the PS3 to install Linux onto it:

1. Remove the flash drive from your computer and insert it into the PS3.
2. Go to **Settings**, then **System Settings**, and then choose **Install Other OS**.
3. Confirm the location of the installer, and follow the screens for the installation process. Note that this *only installs the bootloader, not the operating system.*
4. When the installer finishes, go to **Settings**, then **System Settings**, and select **Default System**. Then choose **Other OS** and press the **X** button.
5. Insert the YDL 5 DVD.
6. Plug in your USB keyboard and mouse.
7. Now restart the system. You can either do this by holding down the **PS** button on the controller and then choosing **Turn off the system**, or by simply holding the power button down for five seconds. Then turn the system back on.
8. When it boots back up, it will look like it is booting Linux. That's because the bootloader is actually a really stripped down Linux kernel called kboot.
9. When it gets to the kboot: prompt, type `install` if your output is going through the HDMI port, or `installtext` if you are going analog. The remaining instructions assume you used the installtext option, but there is little difference.
10. After media verification it may give a Traceback error in the blue area of the screen. Just ignore this and proceed through the installation screens.
11. When it asks about partitioning, don't be concerned about it erasing the PS3 game operating system. The PS3's Other OS mode only allows the guest operating system to see its own portion of the drive. Even low-level utilities cannot see the other parts of the drive. So go ahead and let YDL erase all of the data on your drive, and then let it remove all of the partitions and create a default layout.
12. When it gets to the package installation, it takes approximately one hour to install the packages. However, it does not install the whole DVD.
13. When it reboots, if you are using analog output, you need to type in `yd1480i` at the kboot: boot prompt. Otherwise it will likely change the output to a resolution that the analog output isn't capable of.
14. When it boots, it will bring up a setup tool. There is nothing you really need to do here. If you don't do anything, it will time out and finish the bootup process.

And there you have it! YDL 5 is now on your PS3!

Post-install setup

Unfortunately, the installation program doesn't take care of all of the details, especially for analog displays. You still have several steps to do if you want to do things like automatically boot at the proper resolution, configure the X Window System on an analog device, and install the Cell BE SDK. For all of these, go ahead and make sure your YDL 5 DVD is in the drive, and mount it like this:

```
mount /dev/dvd /mnt
```

All of the instructions will assume the install DVD is mounted in this way, and that you are logged in as root.

To get an analog system to boot into its proper resolution at startup, edit the file /etc/kboot.conf, and change the line which reads `default=ydl` to `default=ydl480i` and save the file.

If you want to configure the X Window System for your analog device, you need to install and run the Xautoconfig package like this:

```
rpm -i /mnt/YellowDog/RPMS/Xautoconfig-*
Xautoconfig
```

Now you can start the X Window System by running startx, though on an analog device your screen is pretty tiny. Here's a quick hint to help you get around on such a tiny device: holding down **alt+left mouse button** will allow you to drag screens around on your desktop, even if you can't see the title bar.

If you want your system to have a graphical login at system boot, you need to edit the /etc/inittab file. Change the line `id:3:initdefault:` so that it says `id:5:initdefault:` and save the file. Now when you reboot the system, you will have a nice graphical login. Remember after you reboot to mount the DVD as shown above for the remaining steps. Note that Nautilus actually mounts it in a different location, so if you use Nautilus to mount your DVD, it will be mounted on /media/CDROM rather than /mnt.

Now to install the Cell BE SDK V2.0. To see if it is already installed by the installer, simply do `which spu-gcc`. If it is unable to find the program, then the SDK was not installed. To install it, you need to do the following:

```
cd /mnt/YellowDog/RPMS
rpm -i spu-binutils-* spu-gcc-* spu-gdb-* spu-utils* libspe-devel-*
```

However, one important set of packages did not get included on the DVD -- the 64-bit version of libspe. However, that is easily remedied. Get the SRPM of libspe either from the source DVD or from the Web site (it is called libspe-1.1.0-1.src.rpm). Then go to the directory you downloaded it into and perform the following steps:

```
rpm -i libspe-*.src.rpm
cd /usr/src/yellowdog/SPECS
rpmbuild -bb --target ppc64 libspe.spec
cd ../RPMS/ppc64
rpm -i elfspe-* libspe-*
```

Now you're all set to go. YDL 5 is installed, configured, and ready to go!

Some of you might be wondering how to get back into the game operating system, for the off chance that you might want to play a game or two on your PS3. To do this, run `boot-game-os` from either the

kboot: prompt or from the command line. If for some reason Linux is causing errors and won't load, you can load the game operating system by powering off the PS3, and then holding down the power button for five seconds (until you hear a beep) when powering it back on. Either of these methods will load the game operating system, but *it will also set the default system to be the game operating system as well.* So, to boot back into Linux, you'll have to go back into the settings and set it to boot the Other OS by default.

Okay, I've got Linux installed. Now what?

Now that you have Linux and the Cell BE SDK fully installed, the rest of this series will be about programming and using it. However, for a teaser, see a short introductory program in C which utilizes both the PPE and an SPE below.

Before looking at how this works, take a look at some of the common tools used to build Cell BE programs:

- **gcc**
  Our trusty compiler, built for generating PPC Linux binaries for the PPE. Use the -m64 switch to generate 64-bit executables.
- **spu-gcc**
  This is *also* our trusty compiler, but this one generates code for the SPEs.
- **embedspu**
  This is a special tool that converts SPE programs into an object file that can be linked into a PPE executable. It also creates a global variable that refers to the SPE program so that the PPE can load the program into the SPEs and run the program as needed. To embed into 64-bit PPC programs, use the -m64 flag.

Without the SPEs, the Cell BE processor is programmed essentially like any other PowerPC-based system. In fact, you can pretend that they don't exist and your code will work just fine. Doing this, however, will leave much of your computing power untapped. To take advantage of the SPEs, you will have to put in just a little more effort.

If you are new to Cell BE technology, remember that the PPE is the resource manager for the system. It handles operating system tasks, regulates access to memory, and controls the SPEs. The code for the PPE takes care of initializing the program, setting up one or more SPEs with tasks, and performing input and output. Of course, the PPE can also perform processing tasks as well, but generally the point is to offload all that is reasonable to the SPEs.

So, take a look at how a simple program is constructed to perform processing tasks on the SPE. The program will be very elementary -- it will calculate the distance travelled given a speed in miles-per-hour and a time in hours. Here is the code for the PPE (enter as `ppe_distance.c`):

Listing 1. Equation solver PPE code

```
#include <stdio.h>
#include <libspe.h>

//This global is for the SPE program code itself.  It will be created by
//the embedspu program.
extern spe_program_handle_t calculate_distance_handle;
```

```
//This struct is used for input/output with the SPE task
typedef struct {
        float speed;     //input parameter
        float num_hours; //input parameter
        float distance;  //output parameter
        float padding;   //pad the struct a multiple of 16 bytes
} program_data;

int main() {
        program_data pd __attribute__((aligned(16)));  //aligned for transfer

        //GATHER DATA TO SEND TO SPE
        printf("Enter the speed at which your car is travelling in miles/hr: ");
        scanf("%f", &pd.speed);
        printf("Enter the number of hours you have been driving at that speed: ");
        scanf("%f", &pd.num_hours);

        //USE THE SPE TO PROCESS THE DATA
        //Create SPE Task
        speid_t spe_id = spe_create_thread(0, &calculate_distance_handle, &pd, NULL
         -1, 0);
        //Check For Errors
        if(spe_id == 0) {
                fprintf(stderr, "Error creating SPE thread!\n");
                return 1;
        }
        //Wait For Completion
        spe_wait(spe_id, NULL, 0);

        //FORMAT THE RESULTS FOR DISPLAY
        printf("The distance travelled is %f miles.\n", pd.distance);
        return 0;
}
```

As mentioned before, the main job of the PPE in the Cell BE processor is to handle the input and output tasks. The only really interesting part is `spe_create_thread`. The first parameter is a thread group ID (zero indicates that it should create a new group for the thread), the second parameter is the handle to the SPE program, the third parameter is the pointer to the data you want to transfer, the fourth parameter is an optional environment pointer, the fifth parameter is a mask of which SPEs you are willing to run the program on (-1 indicates any available SPE), and the final parameter is a list of options you want to employ (in this case, you don't want any). The function returns the SPE task ID number, which you then use as a parameter to `spe_wait`. `spe_wait` returns when the SPE task terminates.

Here is the code for the SPE (enter as `spe_distance.c`):

Listing 2. SPE calculation example

```
//Pull in DMA commands
#include <spu_mfcio.h>

//Struct for communication with the PPE
typedef struct {
```

```
        float speed;      //input parameter
        float num_hours;  //input parameter
        float distance;   //output parameter
        float padding;    //pad the struct a multiple of 16 bytes
} program_data;

int main(unsigned long long spe_id, unsigned long long program_data_ea, unsigned
long long env) {
        program_data pd __attribute__((aligned(16)));
        int tag_id = 0;

        //READ DATA IN
        //Initiate copy
        mfc_get(&pd, program_data_ea, sizeof(pd), tag_id, 0, 0);
        //Wait for completion
        mfc_write_tag_mask(1<<tag_id);
        mfc_read_tag_status_any();

        //PROCESS DATA
        pd.distance = pd.speed * pd.num_hours;

        //WRITE RESULTS OUT
        //Initiate copy
        mfc_put(&pd, program_data_ea, sizeof(program_data), tag_id, 0, 0);
        //Wait for completion
        mfc_write_tag_mask(1<<tag_id);
        mfc_read_tag_status_any();
        return 0;
}
```

Future articles examine SPU programs more in depth, but here's a quick rundown of what is happening. The pointer passed as the third parameter to `spe_create_thread` comes in to this program as `program_data_ea`. EA stands for *effective address*, which is a main memory address as viewed from the main PPE program. Since the SPE does not have direct access to main memory, you cannot directly dereference this as a pointer. Instead, you must initiate a transfer request to copy data into your local store. Once it is in your local store, you can access the data through the *local store address*, sometimes abbreviated as LSA.

`mfc_get` initiates the transfer into the local store. Notice that in both the PPE and the SPE the struct was aligned to 16 bytes and padded to 16 bytes. We will deal with this more in a later article, but for the most part DMA transfers *must* be aligned to a 16-byte boundary and be sized as a multiple of 16 bytes. The `tag_id` allows for you to retrieve the status of the DMA operation. After the transfer, the next two functions cause the program to wait until the transfer is completed.

The main processing is trivially simple -- just multiplying the speed and the time. After the data is processed, `mfc_put` initiates a transfer back into main memory, and the next two functions cause you to wait for DMA completion. When all of that is done, the program exits.

Now you have to compile and run the program. That is actually pretty simple:

```
#Compile the SPE program
spu-gcc spe_distance.c -o spe_distance
#Embed the SPE program into an ELF object file, and expose it
#through the global variable: calculate_distance_handle
```

```
embedspu calculate_distance_handle spe_distance spe_distance_csf.o
#Compile the PPE program together with the SPE program
gcc ppe_distance.c spe_distance_csf.o -lspe -o distance
#Run the program
./distance
```

And there you have it! A fully working Cell BE program.

Conclusion

While you can't program the PS3 directly, its support for third-party operating systems allows you to install Linux on your PS3. Installing Linux on the PS3 takes a little bit of effort, but in the end you get a low-cost, fully working Cell Broadband Engine processor. Future articles in this series go in depth into programming the Cell BE and extracting every ounce of speed that you can from the SPEs.

Resources

- To purchase Yellow Dog Linux 5, go to Terra Soft Solution's Web site. (You can get it on public mirrors as well.)
- Get the bootloader from YDL's Web site. Be sure to name the file "otheros.bld".
- Get the Other OS Installer from Sony's Web site. Be sure to name the file "otheros.self".
- For those of you wanting to connect your PS3 to a monitor from the analog output, this cable purports to do just that at a low price. (I have not tested this cable myself, so I can't speak to its quality.)
- A short YDL installation manual is also available.
- The YDL Web site has a page about video support on the PS3.
- Fedora Core 5 can also go on the PS3, although there is no official support. To install Fedora, you'll need the Fedora Core 5 PPC Install DVD, the FC5 PS3 AddOn DVD, and the installation instructions. Also note that the Cell SDK that comes with the AddOn DVD has the gcc for the PPE named ppu-gcc and the embedspu utility is ppu-embedspu.
- Gamasutra has another introduction to programming the PS3 on Linux. However, it uses a function, copy_from_ls, which is not available in the SDK, but only in the code samples from IBM which it ships with its Cell BE simulator.
- The IBM Cell BE resource center includes links to a number of articles on the PPE, SPE, EIB, and more.
- IBM has also collected in one place a full set of Cell BE documentation.

About the author

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.

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

Jonathan Bartlett (johnnyb@eskimo.com), Director of Technology, New Medio

Summary: 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.

Date: 07 Feb 2007
Level: Intermediate
Also available in: Chinese Korean Russian

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 `stqd $lr, LR_OFFSET($sp)`. `stqd` 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 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.

Programming high-performance applications on the Cell BE processor, Part 3: Meet the synergistic processing unit

Programming the synergistic processing elements of the Sony PLAYSTATION 3

Jonathan Bartlett (johnnyb@eskimo.com), Director of Technology, New Medio

Summary: Continue looking in depth at the Cell Broadband Engine™ (Cell BE) processor's synergistic processor elements (SPEs) and how they work at the lowest level. This installment explores storage alignment issues and the communication facilities of the SPEs.

Date: 22 Feb 2007
Level: Intermediate
Also available in: Korean Russian Japanese

Non-aligned loads and stores

Because the synergistic processing unit (SPU) is focused on vector, not scalar, processing, it is only able to load and store 16 bytes at a time (the size of a register) from local store locations which are aligned on 16-byte boundaries. Therefore, you cannot just load a word from, say, memory location 12. To get that word, you would need to load a quadword from memory location 0, and then shift the bits so that the value you want is in the preferred slot. The original quadword must be loaded, the appropriate value inserted into the right location in the quadword, and then the result stored back. Because of these issues, it is usually advisable to store all data aligned to 16 bytes. To load a value which crosses a 16-byte boundary is even more difficult, as you would actually have to load it into two registers, shift them, and then mask and combine them. Storing such values is even more difficult, so it is best to never use values that cross 16-byte boundaries.

While it allows you to use data that is not aligned to 16-byte boundaries, the loading and storing technique I will discuss requires that the data be *naturally aligned* to prevent it from crossing the 16-byte boundary. That means that words will be 4-byte aligned, halfwords will be 2-byte aligned, and bytes don't have to be aligned at all.

Doing an unaligned load requires two or three instructions, depending on the size of the data. The reason for this is that if you are loading a single value, you probably want it in the preferred slot of the register. The first instruction does the load and the second instruction rotates the value so that the requested address is at the beginning of the register. Then, if the data is smaller than a word, a shift is needed to move it away from the beginning into the preferred slot (if it is a word or a doubleword, the beginning of the register *is* the preferred slot). Here is the code for a byte load, which takes an address in the preferred slot of register 3 and uses it to load a byte into the preferred slot of register 4:

Listing 3. Load from non-aligned memory

```
###Load byte unaligned address $3 into preferred slot of register $4###

#Loads from nearest quadword boundary
lqd $4, 0($3)
#Rotate value to the beginning of the register
rotqby $4, $4, $3
#Rotate value to the preferred slot (-3 for bytes, -2 for halfwords, and nothing fo
words or doublewords)
rotqbyi $4, $4, -3
```

Remember, the `lqd` instruction only loads from 16-byte boundaries. It will therefore ignore the four least significant bits during the load, and just load an aligned quadword from memory. Therefore, for arbitrary addresses, we have no idea where in the loaded quadword the value we wanted is. The `rotqby` instruction, "rotate (left) quadword by bytes," uses the address you loaded from to indicate how far to rotate the register. It only uses the least four significant bits of the address in the register (the ones ignored by the load) to determine how far to rotate. This will always be the number of bytes it needs to shift left to move the address specified to the beginning of the register. Finally, for bytes, the preferred slot is *not* at the beginning of the register, but three bytes to the right. So the instruction `rotqbyi` does a shift using an immediate-mode value to shift by. Word- and doubleword-sized transfers do not need this last instruction, because their preferred slot is at the beginning of the register anyway. At the end of this, register 4 has the final value, with the byte shifted into the preferred slot.

Storing is more difficult. Here is the code to store a byte that is in the preferred slot of register $4 into the address specified by register $3:

Listing 4. Store to non-aligned address

```
###Store preferred byte slot $4 into unaligned address $3

#Load the data into a temporary register
lqd $5, 0($3)
#Generate the controls for a byte insertion
cbd $6, 0($3)
#Shuffle the data in
shufb $7, $4, $5, $6
#Store it back
stqd $7, 0($3)
```

To understand this cryptic-looking sequence, again keep in mind that the SPU only does loads and stores a quadword at a time, on quadword-aligned addresses. Therefore, if you want to store only one byte, if you tried to do it directly on an unaligned address, it would both go into the wrong location and clobber the remaining bytes in the quadword. To avoid this, you need to first load the quadword from memory, insert the value into the appropriate byte in the quadword, and then store it back. The hard part is inserting it into the proper location based only on the address. Thankfully, two instructions help out, `cbd` ("generate control for byte insertion") and `shufb` ("shuffle bytes"). The `cbd` instruction takes an address and generates a control word that can be used by `shufb` to insert a byte at the proper location in the quadword for that address. `cbd $6, 0($3)` uses the address in register 3 to

generate the control quadword, and then stores it in register 6. The instruction `shufb $7, $4, $5, $6` uses the control quadword in register 6 to generate a new value into register 7 which consists of the original quadword that was in memory (now in register 5) and a byte from register 4 in the preferred slot, and stores the result in register 7. Once the byte is shuffled in, the value is stored back into memory.

To illustrate the technique, I'll write a function that takes the address of an ASCII character, loads it, converts it to uppercase, and stores it back. I'll put the function `convert_to_upper` in a separate file from the `main` function so that I can reuse it in another program later on. Here is the code for the `main` function (save it as `convert_main.s`):

Listing 5. Uppercase conversion program start

```
.data

string_start:
.ascii "We will convert the following letter, "
letter_to_convert:
.ascii "q"
remaining:
.ascii ", to uppercase\n\0"

.text
.global main
.type main, @function

main:
        .equ MAIN_FRAME_SIZE, 32
        .equ LR_OFFSET, 16
        #PROLOGUE
        stqd $lr, LR_OFFSET($sp)
        stqd $sp, -MAIN_FRAME_SIZE($sp)
        ai $sp, $sp, -MAIN_FRAME_SIZE

        #MAIN FUNCTION
        ila $3, letter_to_convert
        brsl $lr, convert_to_upper
        ila $3, string_start
        brsl $lr, printf

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

Now enter the function that actually does the uppercase conversion (enter as `convert_to_upper.s`):

Listing 6. Function to convert to uppercase

```
.text
.global convert_to_upper
.type convert_to_upper, @function
convert_to_upper:
```

```
        #Register usage
        # $3 - parameter 1 -- address of byte to be converted
        # $4 - byte value to be converted
        # $5 - $4 greater than 'a' - 1?
        # $6 - $4 greater than 'z'?
        # $7 - $4 less than or equal to 'z'?
        # $8 - $4 between 'a' and 'z' (inclusive)?
        # $9 through $12 - temporary storage for final store
        # $13 - conversion factor

        #address of letter stored in unaligned address in $3
        #UNALIGNED LOAD
        lqd $4, 0($3)
        rotqby $4, $4, $3
        rotqbyi $4, $4, -3

        #IS IN RANGE 'a'-'z'?
        cgtbi $5, $4, 'a' - 1
        cgtbi $6, $4, 'z'
        nand $7, $6, $6
        and $8, $5, $7
        #Mask out irrelevant bits
        andi $8, $8, 255
        #Skip uppercase conversion and store if $4 is not lowercase (based on $8)
        brz $8, end_convert

is_lowercase:
        #Perform Conversion
        il $13, 'a' - 'A'
        absdb $4, $4, $13

        #Unaligned Store
        lqd $9, 0($3)
        cbd $10, 0($3)
        shufb $11, $4, $9, $10
        stqd $11, 0($3)

end_convert:
        #no stack frame, no return value, just return
        bi $lr
```

To compile and run, perform the following commands:

```
spu-gcc convert_main.s convert_to_upper.s -o convert
./convert
```

The `main` function doesn't function too differently than before, so I won't discuss it here. Note, however, that it is passing the *address* of the letter to `convert_to_upper`, not the letter itself.

The `convert_to_upper` function takes the address of an arbitrary character, converts it to uppercase, and then stores it back and returns nothing. It never calls another function, so it doesn't need a stack frame.

The first thing the function does is an unaligned load as described previously into register 4. It then checks to see if the byte is in the range `a` through `z`. It does that by comparing if it is greater than `'a' - 1`, and then seeing if it is greater than `'z.'` I did not do a "less than" comparison, *because they aren't available on the SPU!* SPUs only have comparisons for "greater than" and "equal to." Therefore, if you want to do a "less than or equal to" comparison, you must do a "greater than" comparison and then do a "not" on it, which is performed using the `nand` instruction with both source arguments being the same register. You then combine the comparisons using the `and` instruction (note that you could have combined all the logical instructions into one with an `xor`, but the code would have been much less clear). Finally, because the branch instructions only operate on halfword or word values, you have to mask out the non-relevant portions of the register. (I didn't have to do that in the factorial example because I was dealing with a full word).

If the bits in the preferred slot of register 8 are all set to false, you skip to the end of the function. If they are true, you perform the conversion. The only byte-oriented arithmetic function on the SPU is `absdb`, "absolute difference of bytes," which gives the absolute value of the difference between two operands. You use that, combined with the difference between the lowercase and uppercase values, to perform the conversion. Finally, you perform an unaligned store. Since you did not call any functions or use any local storage, you did not need a stack frame at all, so you can now just exit through the link register.

Communication with the PPE

So far I have concentrated on SPE-only programs. Now I will look into PPE-controlled programs, and for that, I need to know how to get the PPE and the SPE to communicate.

Channels and the MFC

Remember that SPEs have a memory that is separate from the processor's main memory, called the *local store*. The SPE cannot read main memory directly, but instead must import and export data between the local store and main memory using DMA commands to a unit called the *memory flow controller*, or MFC. The local store address space is limited to 32 bits, but it is usually much smaller (in the Sony® PLAYSTATION® 3, for instance, it is only 18 bits). The reason for this is so that memory accesses by SPE code can be deterministic. Main memory can get swapped out, moved around, cached, uncached, or memory mapped. Therefore, the amount of time required for any particular memory access is completely unknown (if the memory is swapped out, who knows how long it will take). By separating out the SPE memory into a local store, the SPE can have a deterministic access time for any memory it accesses, and schedule the MFC to asynchronously move data in and out of main memory as needed. Addresses within an SPE's local store are called *local store addresses* (LSAs), while addresses within the main memory are called *effective addresses* (EAs). This will be important as you learn how to use the memory flow controller's DMA facilities.

SPEs communicate with the outside world by using *channels*. A channel is a 32-bit area which can be written to or read from (but not both -- they are unidirectional) using special instructions. A channel can also have a depth, or *channel count*. The channel count is the amount of data waiting to be read (for read channels), or the amount of data which can still be written (for write channels). Channels are used for all SPE input and output. They are used for issuing DMA commands to the memory flow controller, handling SPE events, and reading and writing messages to and from the PPE. The next program I'll show you utilizes the MFC and the channel interface to do character conversions on data specified by the PPE.

Creating and running SPE tasks

So far, the `main` function has not been using any parameters. However, when it is run from a PPE program, it actually receives three 64-bit parameters -- the SPE task identifier in register 3, a pointer to application parameters in register 4, and a pointer to runtime environment information in register 5. The contents of the areas pointed to by application and environment pointers are actually user-defined. However, remember that they point to memory *in the main storage of the application* (an effective address), not to the SPE's local store. Therefore, they cannot be accessed directly, but must be moved in through DMA.

SPE tasks are created with the function `speid_t spe_create_thread(spe_gid_t spe_gid, spe_program_handle_t *spe_program_handle, void *argp, void *envp, unsigned long mask, int flags)`. The parameters work as follows:

- **spe_gid**
  This is the SPE thread group to assign this task to. It can simply be set to zero.
- **spe_program_handle**
  This is a pointer to a structure which holds the data about the SPE program itself. This data is normally defined either automatically by embedding an SPU application within a PPU executable (this will be shown later), by using `dlopen()/dlsym()` on a library containing an SPU application, or by using `spe_open_image()` to directly load an SPU application.
- **argp**
  This is a pointer to application-specific data for program initialization. Set to null if it is not going to be used.
- **envp**
  This is a pointer to environment data for the program. Set to null if it is not going to be used.
- **mask**
  This is the processor affinity mask. Set it to -1 to assign the process to any available SPE. Otherwise, it contains a bitmask for each available processor. 1 means that the processor should be used, 0 means that it should not. Most applications set this to -1.
- **flags**
  This is a set of bit flags which modify how the SPE is set up. These are all outside the scope of this article.

A PPE/SPE program using DMA

As an example of DMA communication, I will write a program where the PPE takes a string, and invokes an SPE program which copies over the string, converts it to uppercase, and copies it back into main storage. All of the data transfers will use the MFC's DMA facilities, controlled through SPE channels.

The main SPE program will receive an effective address pointer to a struct containing the size and pointer of a string in main memory. It will then copy it into its buffer, perform the conversion, and copy it back. Here is the SPE code (enter as `convert_dma_main.s`):

Listing 7. SPU code to perform uppercase conversion for PPU program

```
.data

.align 4
conversion_info:
```

```
conversion_length:
        .octa 0
conversion_data:
        .octa 0
.equ CONVERSION_STRUCT_SIZE, 32

.section .bss #Uninitialized Data Section
.align 4
.lcomm conversion_buffer, 16384

.text
.global main
.type main, @function

#MFC Constants
.equ MFC_GET_CMD, 0x40
.equ MFC_PUT_CMD, 0x20

#Stack Frame Constants
.equ MAIN_FRAME_SIZE, 80
.equ MAIN_REG_SAVE_OFFSET, 32
.equ LR_OFFSET, 16

main:
        #Prologue
        stqd $lr, LR_OFFSET($sp)
        stqd $sp, -MAIN_FRAME_SIZE($sp)
        ai $sp, $sp, -MAIN_FRAME_SIZE

        #Save Registers
        #Save register $127 (will be used for current index)
        stqd $127, MAIN_REG_SAVE_OFFSET($sp)
        #Save register $126 (will be used for base pointer)
        stqd $126, MAIN_REG_SAVE_OFFSET+16($sp)
        #Save register $125 (will be used for final size)
        stqd $125, MAIN_REG_SAVE_OFFSET+24($sp)

        ##COPY IN CONVERSION INFORMATION##
        ila $3, conversion_info          #Local Store Address
        #register 4 already has address #64-bit Effective Address
        il $5, CONVERSION_STRUCT_SIZE    #Transfer size
        il $6, 0                         #DMA Tag
        il $7, MFC_GET_CMD               #DMA Command
        brsl $lr, perform_dma

        #Wait for DMA to complete
        il $3, 0
        brsl $lr, wait_for_dma_completion

        ##COPY STRING IN TO BUFFER##
        #Load buffer data pointer
        ila $3, conversion_buffer #Local Store
        lqr $4, conversion_data   #64-bit Effective Address
        lqr $5, conversion_length #SIZE
        il $6, 0                  #DMA Tag
        il $7, MFC_GET_CMD        #DMA Command
        brsl $lr, perform_dma

        #Wait for DMA to complete
        il $3, 0
```

```
        brsl $lr, wait_for_dma_completion

        #LOOP THROUGH BUFFER
        #Load buffer size
        lqr $125, conversion_length
        #Load buffer pointer
        ila $126, conversion_buffer
        #Load buffer index
        il $127, 0
loop:
        ceq $7, $125, $127
        brnz $7, loop_end

        #Compute address for function parameter
        a $3, $127, $126
        #Next index
        ai $127, $127, 1

        #Run function
        brsl $lr, convert_to_upper

        #Repeat loop
        br loop

loop_end:
        #Copy data back
        ila $3, conversion_buffer    #Local Store Address
        lqr $4, conversion_data      #64-bit effective address
        lqr $5, conversion_length    #Size
        il $6, 0                     #DMA Tag
        il $7, MFC_PUT_CMD           #DMA Command
        brsl $lr, perform_dma

        #Wait for DMA to complete
        il $3, 0
        brsl $lr, wait_for_dma_completion

        #Return Value
        il $3, 0

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

This code relies on some utility functions for handling DMA commands. Enter those functions as `dma_utils.s`:

Listing 8. DMA transferring utilities

```
##UTILITY FUNCTION TO PERFORM DMA OPS##
#Parameters - Local Store Address, 64-bit Effective Address, Transfer Size,
DMA Tag, DMA Command
.global perform_dma
.type perform_dma, @function
perform_dma:
```

```
        shlqbyi $9, $4, 4  #Get the low-order 32-bits of the address
        wrch $MFC_LSA, $3
        wrch $MFC_EAH, $4
        wrch $MFC_EAL, $9
        wrch $MFC_Size, $5
        wrch $MFC_TagID, $6
        wrch $MFC_Cmd, $7
        bi $lr

.global wait_for_dma_completion
.type wait_for_dma_completion, @function
wait_for_dma_completion:
        #We receive a tag in register 3 - convert to a tag mask
        il $4, 1
        shl $4, $4, $3
        wrch $MFC_WrTagMask, $4
        #Tell the DMA that we only want it to inform us on DMA completion
        il $5, 2
        wrch $MFC_WrTagUpdate, $5
        #Wait for DMA Completion, and store the result in the return value
        rdch $3, $MFC_RdTagStat
        #Return
        bi $lr
```

Now, not only do you need to compile this program, you need to prepare it for embedding in a PPE application. Assuming you still have the `convert_to_upper.s` from your last program in the current directory, here are the commands to compile the code and prepare it for embedding:

```
spu-gcc convert_dma_main.s dma_utils.s convert_to_upper.s -o spe_convert
embedspu -m64 convert_to_upper_handle spe_convert spe_convert_csf.o
```

This produces what is called a *CESOF Linkable*, which allows an object file for the SPE to be embedded in a PPE application and loaded as needed.

Here is the PPU code to make use of the SPU code (enter as `ppu_dma_main.c`):

Listing 9. PPU code to utilize SPU application

```
#include <stdio.h>
#include <libspe.h>
#include <errno.h>
#include <string.h>

/* embedspu actually defines this in the generated object file,
 we only need an extern reference here */
extern spe_program_handle_t convert_to_upper_handle;

/* This is the parameter structure that our SPE code expects */
/* Note the alignment on all of the data that will be passed to the SPE is 16-bytes
typedef struct {
        int length __attribute__((aligned(16)));
        unsigned long long data __attribute__((aligned(16)));
```

```
} conversion_structure;

int main() {
        int status = 0;
        /* Pad string to a quadword -- there are 12 spaces at the end. */
        char *tmp_str = "This is the string we want to convert to uppercase.
        /* Copy it to an aligned boundary */
        char *str = memalign(16, strlen(tmp_str) + 1);
        strcpy(str, tmp_str);
        /* Create conversion structure on an aligned boundary */
        conversion_structure conversion_info __attribute__((aligned(16)));

        /* Set the data elements in the parameter structure */
        conversion_info.length = strlen(str) + 1; /* add one for null byte */
        conversion_info.data = (unsigned long long)str;

        /* Create the thread and check for errors */
        speid_t spe_id = spe_create_thread(0, &convert_to_upper_handle,
        &conversion_info, NULL, -1, 0);
        if(spe_id == 0) {
                fprintf(stderr, "Unable to create SPE thread: errno=%d\n", errno);
                return 1;
        }

        /* Wait for SPE thread completion */
        spe_wait(spe_id, &status, 0);

        /* Print out result */
        printf("The converted string is: %s\n", str);

        return 0;
}
```

To build and execute the program, enter the following commands:

```
gcc -m64 spe_convert_csf.o ppu_dma_main.c -lspe -o dma_convert
./dma_convert
```

A lot of things are going on in this code, and my goal is to introduce all of the necessary foundational material so that we don't get bogged down in it when learning optimization secrets in the next article. (Stay with me, and you'll be on your way to expert SPU programming in no time!) Now, I'll explain what the code is doing. I'll start with the PPU code, since it's a little easier.

The first interesting part of the PPU code is the inclusion of the `libspe.h` header file, which contains all of the function declarations for running programs on the SPE. It then references a handle called `convert_to_upper_handle`. This is only an `extern` reference, not the declaration itself. This is because `convert_to_upper_handle` is defined in `spe_convert_csf.o`. The name of the variable was set on the command line of the `embedspu` command. That variable is the handle to the program code, which will be used to create your SPE tasks.

Next, you define the structure that will be used as the parameter to your SPE program. You need the length of the string and the pointer to the string itself. These all need to be quadword aligned, so that

you can copy it into your main program and use the values with DMA transfers. Note that the pointer you used is declared an `unsigned long long` rather than just a pointer. This is so that the address transfer is stored the same way whether it is compiled in 32-bit mode or 64-bit mode. With a pointer, if it were compiled in 32-bit mode, the pointer would be aligned differently within the structure. You also have to use the `memalign` function and `strcpy` to copy the data into an area of appropriate alignment. Here's a pointer from long nights of trial and error with this stuff: If you are continually receiving a "bus error," you are probably doing a DMA transfer that is either not 16-byte aligned or is not a multiple of 16 bytes.

In the main program, you declare your variables. Note that all of the declared variables which will be copied using DMA are aligned on quadword boundaries and are multiples of quadwords. That's because DMA transfers, with a few exceptions for small transfers, *must be quadword aligned in both the source and destination addresses* (the program will get even better performance if both source and destination are 128-*byte* aligned). Next, the SPE task is created with `spe_create_thread`, passing in your parameter structure. Now you can just wait for the SPE task to complete using `spe_wait`, and then print out the final value. As you may have guessed, most of the interesting parts of the program are taking place on the SPE, including all of the DMA transfers. DMA transfers are almost always done by the SPEs rather than by the PPE because they can handle much more data and many more active DMA operations than the PPE.

Before getting into the details of the main program, I'll explore the DMA utility functions. The first function is `perform_dma`, which, not surprisingly, performs DMA commands. The Cell BE Handbook defines the sequence of channel operations needed to perform a DMA transfer on pages 450-456 (see Resources). The first thing the function is doing is converting the 64-bit effective address in register 4 into two 32-bit components -- a high- and a low-order component (remember, the channels are only 32 bits wide). Because channels are written using a register's preferred word-sized slot, the 64-bit address already has the high-order bits in the preferred slot. Therefore, you just shift the contents to the left by four bytes into a new register to get the low-order bits in the preferred slot. You then write the local store address, the high-order bits of the effective address, the low-order bits of the effective address, the size of the transfer, the "tag" of the DMA command, and then the command itself to their appropriate channels using the `wrch` instruction. When the command is written, the DMA request is enqueued into the MFC provided it has available slots -- yours certainly does as you are not doing any other concurrent DMA requests. The "tag" is a number which can be assigned to one or many DMA commands. All DMA commands issued with the same tag are considered a single group, and status updates and sequencing operations apply to the group as a whole. In this application, you will only have one DMA command active at a time, so all of your operations will use 0 as the DMA tag. The DMA command should be either `MFC_GET_CMD` or `MFC_PUT_CMD`. There are others, but we aren't concerned with them here. MFC commands are all done from the perspective of the SPE, whether or not it is actually the SPE issuing the command. So `MFC_GET_CMD` moves data from main memory to the local store, and `MFC_PUT_CMD` goes the other way.

Because DMA commands are asynchronous, it is useful to be able to wait for one to complete. The function `wait_for_dma_completion` does precisely that. It takes a tag as its only parameter, converts it to a tag mask, requests a DMA status, and then reads the status. So how does this wait for the DMA operation to complete? When writing to the `$MFC_WrTagUpdate` channel with a value of 2, it causes the `$MFC_RdTagStat` to not have a value until the operation is completed. Thus, when you try to read the channel using `rdch`, it will block until the status is available, at which point the transfer will be complete.

Now, moving on to the actual program itself. The first thing our SPE program does is reserve space for the application's parameter data. This is also aligned to quadword boundaries (`.align 4` in

assembly language works the same as `__attribute__((aligned(16)))` in C because 2^4 = 16). `.octa` reserves quadword values (the mnemonic is a holdover from 16-bit days). You then define a constant `CONVERSION_STRUCT_SIZE` for the size of the whole structure.

After this, you go to the `.bss` section, which is like the `.data` section, except that the executable itself does not contain the values, it just notes how much space should be reserved for them. This section is for uninitialized data. `.lcomm conversion_buffer, 16384` reserves 16K of space, with the starting address defined in the symbol `conversion_buffer`. It is defined for holding 16K because that is the maximum size of an MFC DMA transfer. Therefore, if any string is longer than that, the PPE will have to invoke the program multiple times (a better program would simply break up the request into chunks on the SPE side).

The `main` function has the main meat of the program. It starts by setting up a stack frame. It then saves three non-volatile registers that will be used for the main control of the program. Next, it performs a DMA transfer to copy in the parameter structure from the PPE. Remember, the first parameter to the function is the 64-bit address that was passed in from the PPE. You then use a DMA command to fetch the full structure, and wait for the DMA to complete. After the transfer, you use the data in that structure to copy the string itself into your buffer in the local store using another DMA transfer, and wait for it to complete. Note that you used the `ila` instruction ("immediate load address") to load the address of the buffer. The `ila` instruction maxes out as 18 bits, which works for the PLAYSTATION 3. However, if a Cell BE processor has a larger local store size, you would load it instead with the following two instructions:

```
ilhu $3, conversion_buffer@h #load high-order 16 bits of conversion_buffer
iohu $3, conversion_buffer@l #"or" it with the low-order 16 bits of conversion_buff
```

Then the target effective address, the length of the string, the DMA tag, and a `MFC_GET_CMD` DMA command are all passed to `perform_dma`. The program then waits for the operation to complete.

At this point, all of the data is loaded in and you just need to convert it. You then use register 127 as your loop counter and register 126 as your base pointer, and perform `convert_to_upper` on each value until you get to the end of the buffer.

At `loop_end`, all of the data is converted, and you need only to copy it back. You use the same DMA parameters as for the last transfer, but this time it is an `MFC_PUT_CMD` command. Once the DMA is completed, your function is done. You load register 3 with the return value and perform the function epilogue to restore the stack frame and return.

SPE/PPE communication using mailboxes

While DMA transfers are an excellent way of moving bulk data between the SPE and the PPE, another simpler method for smaller transfers which I will briefly discuss is *mailboxes*. For the SPE, it is simply a set of channels (a read channel and a write channel) to write 32-bit values to the PPE.

To demonstrate the concept, I will write a very simple SPE server which waits for an unsigned integer number in the mailbox and then writes back the square of that number. Here is the code (enter as `square_server.s`):

Listing 10. SPU squaring server

```
.text
.global main
.type main, @function
main:
        #Read the value from the inbox (stalls if no value until one is available)
        rdch $3, $SPU_RdInMbox
        #Square the value
        mpyu $3, $3, $3
        #Write the value back
        wrch $SPU_WrOutMbox, $3
        #Go back and do it again
        br main
```

That's all! This will just sit around and wait for requests and process them. It simply quits when the parent program quits. And, if there is no value available in the inbox, the `rdch` instruction simply stalls until there is one.

The PPE side isn't much harder (enter as `square_client.c`):

Listing 11. PPE squaring client

```
#include <libspe.h>
#include <stdio.h>

extern spe_program_handle_t square_server_handle;

int main() {
        int status = 0;

        /* Create SPE thread */
        speid_t spe_id = spe_create_thread(0, &square_server_handle, NULL, NULL, -1
        if(spe_id == 0) {
                fprintf(stderr, "Unable to create SPE thread!\n");
                return 1;
        }

        /* Request a square */
        spe_write_in_mbox(spe_id, 4);
        /* Wait for result to be available */
        while(!spe_stat_out_mbox(spe_id)) {}
        /* Read and display result */
        printf("The square of 4 is %d\n", spe_read_out_mbox(spe_id));

        /* Do it again */
        spe_write_in_mbox(spe_id, 10);
        while(!spe_stat_out_mbox(spe_id)) {}
        printf("The square of 10 is %d\n", spe_read_out_mbox(spe_id));

        return 0;
}
```

To compile and run this program, issue the following commands:

```
spu-gcc square_server.s -o square_server
embedspu -m64 square_server_handle square_server square_server_csf.o
gcc -m64 square_client.c square_server_csf.o -lspe -o square
./square
```

The mailboxes, even for the PPE, are named according to the perspective of the SPE. So you write to the inbox and read from the outbox if you are the PPE. Unlike the SPE, the PPE does not stall and wait for a value when it reads or writes. Instead, the program must use `spe_stat_out_mbox` to wait for a value, and `spe_stat_in_mbox` to see if there are slots left for writing to the mailbox. You don't use the latter as you only have one value in play at a time.

The real power of mailboxes comes when a program combines the mailbox and the DMA approach. For example, an SPE task can be created which listens for buffer addresses on its mailbox, and then uses that address to pull in all of the data to be processed through DMA.

Conclusion

Thus far, this series has covered the main concepts of assembly language programming on the Cell BE processor of the PLAYSTATION 3 under Linux®. Topics covered include the basic architecture, the syntax of the SPU assembly language, and the primary modes of communication between the SPE and the PPE. The next article looks at how to pump every ounce of performance out of the Cell BE processor SPEs that you can. And later articles will apply this knowledge to SPE programming in C, to make your life just a little bit easier.

Resources

- See the other parts in the Programming high-performance applications on the Cell BE processor series.
- 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 instruction set architecture architecture reference.
- For ABI details, see the SPU ABI documentation as well as the Linux extensions to the ABI.
- An additional method of interprocess communication using special references called EAR references is this guide to CESOF linkables. However, the example given uses the function `copy_from_ls` which is not available in the open-source SDK, but is available in the IBM System Simulator for the Cell BE processor. `copy_from_ls` and `copy_to_ls` allow you to perform DMA transfers without regards to alignment, but they both take considerably longer to run.

- Check out this good tutorial on DMA transfers on the Cell BE processor using C.
- And here is a more extensive tutorial on using mailboxes (also in C).
- The documentation of the SPE management library describes in detail task creation and communication with SPEs from the PPE.
- The Definitive Source of Information about the Cell BE processor itself is the Cell BE Handbook.
- Keep abreast of all things Cell BE: subscribe to IBM microNews.

About the author

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.

Programming high-performance applications on the Cell BE processor, Part 4: Program the SPU for performance

Jonathan Bartlett (johnnyb@eskimo.com), Director of Technology, New Medio

Summary: Write optimal code for the Cell Broadband Engine™ (Cell BE) processor's synergistic processing unit (SPU) and have your programs running lightning fast. This installment of Programming high-performance applications on the Cell BE processor covers SIMD vector programming, branch elimination, loop unrolling, instruction scheduling, and branch hinting techniques. Previous installments have covered the basics of the Sony® PLAYSTATION® 3, the Cell BE architecture, and SPU programming.

Date: 06 Mar 2007
Level: Intermediate
Also available in: Chinese Russian

This article dives into the depths of instruction-cycle-counting, bit manipulation, and other nuances that assembly language has typically been notorious for. By the end of it you might be convinced never to program in assembly language ever again. However, the point of it all is not to program in assembly language at all times, but rather to understand what the compiler needs to do to optimize your code, and be able to supplement that with custom assembly language when required. Knowing how the SPU's assembly language works will also aid you in exploiting the processor in higher-level languages. Subsequent articles will use C and show how to use this optimization knowledge in real-world examples. The SPU has many C language extensions; knowing SPU assembly language will help you make sense of them, and knowing SPU optimization will help you use them well.

The starting program

The last article ended with a function called `convert_to_upper`, which operated one byte at a time to convert a string to uppercase. The functions in the programs in this article operate on whole buffers at a time. The SPU is built to deal with data in batches, so moving to a buffer-at-a-time model will make the enhancements easier. The first version will simply wrap a loop around the code developed in the previous article. Because it is based on code and concepts developed in the previous articles, I will not do a step-by-step explanation of the code.

Here is the unoptimized version of a buffer-at-a-time function for converting to uppercase (enter as `convert_buffer.s`):

Listing 1. First example program

```
.text

.global convert_buffer_to_upper
.type convert_buffer_to_upper, @function
convert_buffer_to_upper:
        ##REGISTER USAGE:
        #    3) buffer address / current address
        #    4) buffer size
        #    5) end_address
        #    6) current quadword
        #    7) current quadword with byte in first position
        #    8, 9, & 10) Determine if byte is in range
        #    11) byte insertion control
        #    12) current quadword with byte properly inserted
        #    13) true if we need to branch, false otherwise
        #    14) conversion factor

        #Calculate end address
        a $5, $4, $3

loop_start:
        #UNALIGNED LOAD
        lqd $6, 0($3)
        rotqby $7, $6, $3
        rotqbyi $7, $7, -3

        #IS IN RANGE 'a'-'z'?
        cgtbi $8, $7, 'a' - 1
        cgtbi $9, $7, 'z'
        xor $10, $8, $9
        andi $10, $10, 255

        #If no, exit
        brz $10, finish_loop

is_lowercase:
        #If yes, perform conversion
        il $14, 'a' - 'A'
        absdb $7, $7, $14

finish_loop:
        #Unaligned Store ($6 already has current word)
        cbd $11, 0($3)
        shufb $12, $7, $6, $11
        stqd $12, 0($3)

        #Increment pointer
        ai $3, $3, 1

        #Are we at the end?  If not then loop.
        cgt $13, $3, $5
        brz $13, loop_start

end_function:
        #Return
        bi $lr
```

As far as performance goes, the current code is terrible. The subsequent sections will improve on it a step at a time.

The function which drives the conversion function is now a little bit simpler since it only has to load the data, run the function, and copy it back. Here is the code (enter as `convert_driver.s`):

Listing 2. Uppercase conversion main function

```
.data

#This is the struct we will copy from the main PPE process
.align 4
conversion_info:
conversion_length:
        .octa 0
conversion_data:
        .octa 0
.equ CONVERSION_STRUCT_SIZE, 32

.section .bss #Uninitialized Data Section

#This is the buffer we will store the string in
.align 4
.lcomm conversion_buffer, 16384

.text

#MFC Constants
.equ MFC_GET_CMD, 0x40
.equ MFC_PUT_CMD, 0x20

.equ LR_OFFSET, 16

.global main
.type main, @function
.equ MAIN_FRAME_SIZE, 32
main:
        #Prologue
        stqd $lr, LR_OFFSET($sp)
        stqd $sp, -MAIN_FRAME_SIZE($sp)
        ai $sp, $sp, -MAIN_FRAME_SIZE

        ##COPY IN CONVERSION INFORMATION##
        ila $3, conversion_info          #Local Store Address
        #register 4 already has address  #64-bit Effective Address
        il $5, CONVERSION_STRUCT_SIZE    #Transfer size
        il $6, 0                         #DMA Tag
        il $7, MFC_GET_CMD               #DMA Command
        brsl $lr, perform_dma

        #Wait for DMA to complete
        il $3, 0
        brsl $lr, wait_for_dma_completion

        ##COPY STRING IN TO BUFFER##
        #Load buffer data pointer
        ila $3, conversion_buffer #Local Store
        lqr $4, conversion_data   #64-bit Effective Address
```

```
lqr $5, conversion_length #SIZE
il $6, 0                  #DMA Tag
il $7, MFC_GET_CMD        #DMA Command
brsl $lr, perform_dma

#Wait for DMA to complete
il $3, 0
brsl $lr, wait_for_dma_completion

##PERFORM CONVERSION##
ila $3, conversion_buffer
lqr $4, conversion_length
brsl $lr, convert_buffer_to_upper

##COPY DATA BACK##
ila $3, conversion_buffer   #Local Store Address
lqr $4, conversion_data     #64-bit effective address
lqr $5, conversion_length   #Size
il $6, 0                    #DMA Tag
il $7, MFC_PUT_CMD          #DMA Command
brsl $lr, perform_dma

#Wait for DMA to complete
il $3, 0
brsl $lr, wait_for_dma_completion

##EXIT PROGRAM##
#Return Value
il $3, 0

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

You will also need the `dma_utils.s` and the `ppu_dma_main.c` files from the previous article.

To build and run, perform these steps:

```
spu-gcc convert_buffer.s convert_driver.s dma_utils.s -o spe_convert
embedspu -m64 convert_to_upper_handle spe_convert spe_convert_csf.o
gcc -m64 spe_convert_csf.o ppu_dma_main.c -lspe -o dma_convert
./dma_convert
```

These same steps can be used to build all of the examples in this article.

Vectorizing the code

The most obvious optimization to make on a vector process is to vectorize the code. This is known as SIMD (single instruction, multiple data), or data parallelism. On the SPUs, most instructions can operate on registers as if they contained multiple, independent values (thus the single instruction acting on multiple data items). Each 128-bit register can be treated as 16 independent bytes, eight half

-words, four words, two doublewords, or as a single unit. The instruction set is primarily geared around dividing it into four 32-bit words, but there is enough support to handle any of these situations.

If you vectorize this code, since you are treating the values as bytes, that means that each instruction will operate on 16 values at once! However, the problem is that vector processing assumes that each and every instruction will be applied to all elements of the vector. However, in the main loop, you have a conditional branch. This means that vector elements which match the criteria go through a different set of instructions than those that do not. Therefore, at least the way the code presently stands, it cannot be vectorized.

What you need to do first is *eliminate the branch* so that the code uses the exact same instructions whether it matches your condition or not (as I'll show later, eliminating branches helps reduce branching stalls as well). So how is this done? The key is that the SPU has several conditional instructions, such as `selb`, `shufb`, and the bit operations, which allow conditional operations to occur without branching. What the program will end up doing is *calculating both answers*, and then using a conditional instruction to select the desired answer.

Here is the conversion code as it currently stands:

```
        #IS IN RANGE 'a'-'z'
        cgtbi $8, $7, 'a' - 1
        cgtbi $9, $7, 'z'
        xor $10, $8, $9
        andi $10, $10, 255

        brz $10, finish_loop

is_lowercase:
        #lowercase condition
        il $14, 'a' - 'A'
        absdb $7, $7, $14

finish_loop:
        #non-lowercase condition
        #all code winds up here
```

In this case, the two answers we are computing are:

- Uppercase-converted letter (if lowercase)
- Original input letter (if not lowercase)

The code starts with the original value in `$7`. The first thing you need to do is to move the code which calculates the converted value *before the condition*, and then store it in a different register (`$15` in this case). So the code will look like this:

```
        #$7 has our original value
        il $14, 'a' - 'A'
        absdb $15, $7, $14
        #$7 has the original, and $15 has the converted value
        #Choose between the value in $7 and $14 and put it in $7

        ##...rest of loop...
```

So now you need to figure out which value you want to use. The first thing you need to do is to use your original instructions to check the condition:

```
cgtbi $8, $7, 'a' - 1
cgtbi $9, $7, 'z'
xor $10, $8, $9
```

Note that the previous `andi` is no longer needed because it was used to mask out unwanted values for the conditional branch (conditional branches are based on true or false value of the *word* preferred slot value and you only care about the *byte* preferred slot value). Since you aren't branching you don't care! So now `$10` has all ones in the preferred slot if it is in range, and all zeroes if it is out of range. Now all you need is to choose `$7` or `$15` based on the value in `$10`. The instruction `selb` (select bits) is perfect for this. `selb` has four operands:

1. destination register
2. source value 1
3. source value 2
4. selector

`selb` operates by going through the selector bit-by-bit. For each bit position, if the bit is 0, the same bit position in the destination register uses the bit from source value 1. If the bit is 1, it uses the bit from source value 2. If you imagine each register as an array of bits, `selb` has the following meaning:

```
//imaginary representation of selb for those more familiar with C than assembly lan
for(i = 0; i < 128; i++) {
        destination[i] = selector[i] == 0 ? source_1[i] : source_2[i]
}
```

Now I hope you see why the condition statements set all of the corresponding bits in the destination register to 1 if the condition is true -- it makes it easier to use that value for `selb`. In this case, you can simply add the following line of code:

```
selb $7, $7, $15, $10
```

Now, all of your values, whether they are lowercase or not, will be processed through the following code sequence:

Listing 3. Branch-free conversion code

```
#Original value starts in $7

#Perform conversion and store in $15
il $14, 'a' - 'A'
```

```
absdb $15, $7, $14

#Is it lowercase ('a'-'z')?
cgtbi $8, $7, 'a'-1
cgtbi $9, $7, 'z'
xor $10, $8, $9
#$10 has all 1s for lowercase and all 0s for non-lowercase in the preferred

#Select appropriate value into $7 based on condition
selb $7, $7, $15, $10

#$7 now has the correct value
```

In this case, the choice was between the original value and a processed value, but the code would have been similar if the choice was between two processed values. In that case, you would have just had two sets of processing instructions, with each set using a different register for its result, and the `selb` instruction choosing between them. Likewise, if there were more than two possible directions for the code to go, multiple `selbs` could be used to choose between them. However, at that point, you probably need to look and see if the cost of calculating all of the different possibilities for every input is worth the benefit of eliminating branches.

Remember that the point of removing the branch was so that you can vectorize the code. The problem was that in order to vectorize the code, the code must follow the same set of instructions for each member of the vector. Now that you have eliminated the branches this is possible.

In fact, the core conversion code *is actually almost already vectorized.* All of the instructions operate on the whole register anyway. The problem before was threefold:

- The branch was causing either the whole register to be converted or not converted.
- The register holding the conversion factor was geared to a single byte usage rather than a whole register (`il` loads the given value into each *word* but you need it in each *byte*).
- The load/store instructions and the loop counter were geared for processing a single byte at a time.

Now that you've eliminated the branch, you need to load your conversion factor into every byte of the conversion register. The easiest way to do this is to put the conversion factor in the `.data` section manually and load it directly in. You should also move it outside of the loop since its value is invariant. So, in the `.data` section, add:

```
.equ CONVERSION_FACTOR, 'a' - 'A'
.align 4
conversion_bytes:
        .fill 16, 1, CONVERSION_FACTOR
```

And in the code before the loop, add:

```
lqr $14, conversion_bytes
```

With these additions, all values in register 7 will be processed appropriately. Here is the code again, with a possible starting value to demonstrate what is happening:

Listing 4. Following a set of values through the conversion

```
#$7 starts with 'Hello There!     '
#In hex, that's   0x48656c6c6f20546865726521202020 20
#$14 is the conversion factor in each byte
#In hex, that's   0x20202020202020202020202020202020

absdb $15, $7, $14
#  -> $15 now has 0x28454c4c4f00344845524501000000 00
cgtbi $8, $7, 'a'-1
#  -> $8 now has  0xffffffffff00ffffffffff0000000000
cgtbi $9, $7, 'z'
#  -> $9 now has  0xff0000000000ff000000000000000000
xor $10, $8, $9
#  -> $10 now has 0x00ffffffff0000ffffffff0000000000
selb $7, $7, $15, $10
#  -> $7 now has  0x48454c4c4f20544845524521202020 20
#     which is hex for 'HELLO THERE!     '
```

So now all you need to do is change the loop so that it will utilize this. It needs to load a full quadword (16 bytes) at once, and store it back at once, and increment the pointer by 16 instead of 1. This, interestingly, will require *fewer* instructions because you are no longer having to mess with the preferred slot. So, here is the complete function with the new loop skeleton:

Listing 5. Loop skeleton for vectorized code

```
##Store Conversion Factor##
.data
.equ CONVERSION_FACTOR, 'a' - 'A'
.align 4
conversion_bytes:
        .fill 16, 1, CONVERSION_FACTOR

.text
.global convert_buffer_to_upper
.type convert_buffer_to_upper, @function
convert_buffer_to_upper:
        #Calculate end address
        a $5, $4, $3

        #Load in conversion factors
        lqr $14, conversion_bytes

loop_start:
        #Aligned Load
        lqd $7, 0($3)

        ##CONVERSION##
        absdb $15, $7, $14
        cgtbi $8, $7, 'a'-1
        cgtbi $9, $7, 'z'
```

```
        xor $10, $8, $9
        selb $7, $7, $15, $10
        ##END CONVERSION##

        #Aligned Store
        stqd $7, 0($3)

        #Increment Pointer
        ai $3, $3, 16

        #Exit if needed ($5 has the ending address)
        cgt $13, $3, $5
        brz $13, loop_start

end_function:
        bi $lr
```

As you can see, the code is much simpler -- it has fewer branches and fewer instructions. However, this new code now assumes that the starting address is 16-byte aligned, and also that it has enough padding on the end that the next data element in memory is also 16-byte aligned. Otherwise, you might end up converting something besides letters! As you can see, for vector processing, *alignment and padding are both critically important*. It doesn't really matter if the data itself is large enough to fit in the buffer. Since it is converting as a vector, it doesn't cost anything to convert a few extra bytes of junk. If you wind up having to waste a few bytes in your buffer, it is trivial compared to the amount of time and code needed to special-case the beginning and end of unaligned data. By keeping the data aligned and padded to 16-byte boundaries, vector operations can be performed effortlessly.

Unrolling loops

Loop unrolling has been an optimization technique since the dawn of computer programming. I cover it here not only because it increases efficiency on its own by eliminating branches, but also because it if you do it right it will help later on in instruction scheduling.

Probably by this point you have already been having trouble keeping up with which register holds what value. After all, the register names are essentially arbitrary numbers, which are nearly impossible to make sense of. However, because the registers are only numbers, you can use `.equ` to give your registers descriptive names. For example, you can rewrite your conversion program as follows (note that the registers have been renumbered):

Listing 6. Uppercase conversion with named registers

```
.data
.equ CONVERSION_FACTOR, 'a' - 'A'
.align 4
conversion_bytes:
        .fill 16, 1, CONVERSION_FACTOR

.text
.global convert_buffer_to_upper
.type convert_buffer_to_upper, @function
        ##REGISTER DEFINITIONS##
        #Loop/function control registers
```

```
        .equ BUFFER_REG, 3              #Buffer address / current address
        .equ BUFFER_SZ_REG, 4           #Buffer size
        .equ BUFFER_END_REG, 5          #End address
        .equ CONVERSION_BYTES_REG, 6    #Conversion data
        .equ IS_FINISHED_REG, 7         #Finished conversion?

        #Conversion-oriented registers
        .equ CURRENT_VAL_REG, 8  #Current quadword
        .equ BOOL_TMP1_REG, 9    #used for computing IN_RANGE_REG
        .equ BOOL_TMP2_REG, 10   #used for computing IN_RANGE_REG
        .equ IN_RANGE_REG, 11    #Value in range?
        .equ PROCESSED_VAL_REG, 12   #Conversion bytes, properly masked

        #Information about registers
        .equ NUMREGS, 5          #Number of per-iteration registers
        .equ REGBYTES, 16        #Number of bytes in a register
convert_buffer_to_upper:
        #Calculate end address
        a $BUFFER_END_REG, $BUFFER_SZ_REG, $BUFFER_REG

        lqr $CONVERSION_BYTES_REG, conversion_bytes

loop_start:
        #Aligned Load
        lqd $CURRENT_VAL_REG, 0($BUFFER_REG)

        ##CONVERSION##
        absdb $PROCESSED_VALS_REG, $CURRENT_VAL_REG, $CONVERSION_BYTES_REG
        cgtbi $BOOL_TMP1_REG, $CURRENT_VAL_REG, 'a'-1
        cgtbi $BOOL_TMP2_REG, $CURRENT_VAL_REG, 'z'
        xor $IN_RANGE_REG, $BOOL_TMP1_REG, $BOOL_TMP2_REG
        selb $CURRENT_VAL_REG, $CURRENT_VAL_REG, $PROCESSED_VAL_REG, $IN_RANGE_REG
        ##END CONVERSION##

        #Aligned Store
        stqd $CURRENT_VAL_REG, 0($BUFFER_REG)

        #Increment Pointer
        ai $BUFFER_REG, $BUFFER_REG, REGBYTES

        #Exit if needed
        cgt $IS_FINISHED_REG, $BUFFER_REG, $BUFFER_END_REG
        brz $IS_FINISHED_REG, loop_start

end_function:
        bi $lr
```

It's a lot more verbose, but it also makes it easier to browse through the code. It also makes it much easier to do instruction scheduling for unrolled loops. I'll get to that in a minute. For the present, look at how you might unroll this loop four times, using different registers for each iteration (using different registers will help when you optimize instruction scheduling). I'll discuss why and how I rewrote the program in this way shortly:

Listing 7. Buffer conversion -- loop unrolled

```
loop_start:
        #ITERATION 0
        lqd $(CURRENT_VAL_REG+0*NUMREGS), 0*REGBYTES($BUFFER_REG)
        absdb $(PROCESSED_VAL_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS), $CONVER
        cgtbi $(BOOL_TMP1_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS), 'a'-1
        cgtbi $(BOOL_TMP2_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS), 'z'
        xor $(IN_RANGE_REG+0*NUMREGS), $(BOOL_TMP1_REG+0*NUMREGS), $(BOOL_TMP2_REG+
        selb $(CURRENT_VAL_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS),
        $(PROCESSED_VAL_REG+0*NUMREGS), $(IN_RANGE_REG+0*NUMREGS)
        stqd $(CURRENT_VAL_REG+0*NUMREGS), 0*REGBYTES($BUFFER_REG)

        #ITERATION 1
        lqd $(CURRENT_VAL_REG+1*NUMREGS), 1*REGBYTES($BUFFER_REG)
        absdb $(PROCESSED_VAL_REG+1*NUMREGS), $(CURRENT_VAL_REG+1*NUMREGS), $CONVER
        cgtbi $(BOOL_TMP1_REG+1*NUMREGS), $(CURRENT_VAL_REG+1*NUMREGS), 'a'-1
        cgtbi $(BOOL_TMP2_REG+1*NUMREGS), $(CURRENT_VAL_REG+1*NUMREGS), 'z'
        xor $(IN_RANGE_REG+1*NUMREGS), $(BOOL_TMP1_REG+1*NUMREGS), $(BOOL_TMP2_REG+
        selb $(CURRENT_VAL_REG+1*NUMREGS), $(CURRENT_VAL_REG+1*NUMREGS),
        $(PROCESSED_VAL_REG+1*NUMREGS), $(IN_RANGE_REG+1*NUMREGS)
        stqd $(CURRENT_VAL_REG+1*NUMREGS), 1*REGBYTES($BUFFER_REG)

        #ITERATION 2
        lqd $(CURRENT_VAL_REG+2*NUMREGS), 2*REGBYTES($BUFFER_REG)
        absdb $(PROCESSED_VAL_REG+2*NUMREGS), $(CURRENT_VAL_REG+2*NUMREGS), $CONVER
        cgtbi $(BOOL_TMP1_REG+2*NUMREGS), $(CURRENT_VAL_REG+2*NUMREGS), 'a'-1
        cgtbi $(BOOL_TMP2_REG+2*NUMREGS), $(CURRENT_VAL_REG+2*NUMREGS), 'z'
        xor $(IN_RANGE_REG+2*NUMREGS), $(BOOL_TMP1_REG+2*NUMREGS), $(BOOL_TMP2_REG+
        selb $(CURRENT_VAL_REG+2*NUMREGS), $(CURRENT_VAL_REG+2*NUMREGS),
        $(PROCESSED_VAL_REG+2*NUMREGS), $(IN_RANGE_REG+2*NUMREGS)
        stqd $(CURRENT_VAL_REG+2*NUMREGS), 2*REGBYTES($BUFFER_REG)

        #ITERATION 3
        lqd $(CURRENT_VAL_REG+3*NUMREGS), 3*REGBYTES($BUFFER_REG)
        absdb $(PROCESSED_VAL_REG+3*NUMREGS), $(CURRENT_VAL_REG+3*NUMREGS), $CONVER
        cgtbi $(BOOL_TMP1_REG+3*NUMREGS), $(CURRENT_VAL_REG+3*NUMREGS), 'a'-1
        cgtbi $(BOOL_TMP2_REG+3*NUMREGS), $(CURRENT_VAL_REG+3*NUMREGS), 'z'
        xor $(IN_RANGE_REG+3*NUMREGS), $(BOOL_TMP1_REG+3*NUMREGS), $(BOOL_TMP2_REG+
        selb $(CURRENT_VAL_REG+3*NUMREGS), $(CURRENT_VAL_REG+3*NUMREGS),
        $(PROCESSED_VAL_REG+3*NUMREGS), $(IN_RANGE_REG+3*NUMREGS)
        stqd $(CURRENT_VAL_REG+3*NUMREGS), 3*REGBYTES($BUFFER_REG)

        #Increment Pointer
        ai $BUFFER_REG, $BUFFER_REG, 4*REGBYTES

        #Exit if needed
        cgt $IS_FINISHED_REG, $BUFFER_REG, $BUFFER_END_REG
        brz $IS_FINISHED_REG, loop_start
```

What this program is doing is *calculating the registers being used.* You could have simply numbered the registers, but then writing the code and remembering which register does what would get even more tedious than before. However, since each iteration uses the same number of registers, you can simply calculate the register number at assembly time. For example, look at `$(BOOL_TMP1_REG+2*NUMREGS)`. This means that it is the `BOOL_TMP1_REG` for iteration 2. The actual register number, since `BOOL_TMP1_REG` is 9 and `NUMREGS` is 5, is 9+2*5, or 19. This way, if you need to add a register to your code later, the assembler will auto-recalculate the new register numbers and

you don't have to alter your register numbering convention. You would just assign the register its own symbolic name and increase the value of `NUMREGS`.

In addition, as will be apparent shortly, if you need to re-order your instructions for faster execution, this way of naming registers will make it much easier to see both iterations of the loop the register is dealing with as well as the register's purpose. It also makes it easier to modify the program when both of these are readily visible from the code itself.

Instruction scheduling

What is not usually apparent to new assembly language programmers is that *the order of instructions affects the speed of the program*. The reason for this is that instructions take more than one cycle to finish, but the processors are set up so that an instruction, depending on the ordering of the instructions, does not have to complete before it begins execution of the next instruction. This is known as *pipelining*. Setting up instructions so that they take full advantage of a processor's pipeline is called *instruction scheduling*. A few important terms related to pipelining and instruction scheduling are:

- **Latency** -- The number of clock cycles an instruction uses to produce a final value. This is the same as the size of the pipeline used to process the value.
- **Stall** -- A clock cycle where the processor does not begin a new instruction.
- **Dependency stall** -- This is a stall that occurs because one of the operands of the next instruction requires a value from a previous instruction that has not yet completed.

Most of the work of performance tuning on the SPU deals with avoiding register stalls. Therefore, take a look at the pipelining of different types of instructions on the SPU (information from the Cell BE Handbook, page 688):

SPU instruction latency

Instruction type	Latency	Pipeline	Additional notes
Double-precision floating-point operations	13	Even	The first six cycles are actually stalls in which no other instruction can be issued. Dual-issue (discussed later) is not allowed with these instructions either.
Integer multiplies, floating-point/integer conversion, interpolate	7	Even	
Single-precision floating-point	6	Even	
Byte operations	4	Even	
Element-based rotates and shifts	4	Even	
Immediate-mode loads	2	Even	
Simple integer and logical operations (including `selb`)	2	Even	

Load and Store Operations	6	Odd	Unlike other architectures, SPU loads and stores are deterministic because there is no cache. By reducing the memory so that it is all on-chip in the local store, the SPU can have much faster, much more reliable load and store times than other types of processors.
Branch hints	6	Odd	Special rules for branch hints will be discussed in a subsequent section.
Channel Operations	6	Odd	
Special-purpose register manipulation	6	Odd	
Branches	4	Odd	Properly hinted branches (discussed in a subsequent section) allow the next instruction to be issued in the very next cycle.
Shuffle bytes	4	Odd	
Quadword rotates and shifts	4	Odd	
Estimate	4	Odd	
Gather, mask, and generate insertion controls	4	Odd	

So, say that I have the following instructions:

```
a $5, $6, $7   #instruction 1
a $8, $5, $9   #instruction 2
a $10, $8, $7  #instruction 3
a $11, $8, $7  #instruction 4
```

In this program, it takes four clock cycles for instruction 1 to finish. Instruction 2 requires the result of instruction 1 (`$5`) to compute, so it has to wait the full four clock cycles. Instruction 3 requires the result of instruction 2 (`$8`), so it has to wait four clock cycles. Instruction 4 can be issued in the clock cycle *immediately after* instruction 3 because it does not require the result of instruction 3 to execute. You can visualize it like this:

```
a $5, $6, $7   #cycle 1
#Stall for $5  #cycle 2
#Stall for $5  #cycle 3
#Stall for $5  #cycle 4
a $8, $5, $9   #cycle 5
#Stall for $8  #cycle 6
#Stall for $8  #cycle 7
#Stall for $8  #cycle 8
a $10, $8, $7  #cycle 9
a $11, $8, $7  #cycle 10
```

As you can see, you will get a drastic increase in performance if you are able to arrange your instructions so that no instruction is waiting on any other instruction.

The SPU is not only able to process multiple values at once through its pipeline, it can also *dual-issue* instructions through different pipelines. The SPU has two pipelines, *even* (sometimes called *pipeline 0* or the *execute* pipeline) and *odd* (sometimes called *pipeline 1* or the *load* pipeline). In the table above, the different types of instructions were listed along with which pipeline they execute in. The SPU actually loads two instructions at a time from a doubleword-aligned boundary. This is called a *fetch group*. If the first instruction in this fetch group is an even pipeline instruction and the second one is an odd pipeline instruction, they can both be issued simultaneously. If these conditions are not all met, or if the second instruction needs to wait for dependencies before issuing, then they are issued in separate cycles. To help align instructions properly for enabling dual-issue, there are two no-operation instructions that can be used to properly pad the instructions -- `nop` (no-operation on the even pipeline) and `lnop` (no-operation on the odd pipeline). Also, you can use `.align 3` to force a given instruction to start in a new fetch group (it will be padded with appropriate no-ops to align it properly).

Look at one iteration in the loop and see how it performs in the SPU pipeline. I've added no-ops so you can see the pipeline issues better:

Listing 8. Loop iteration with stall information

```
.align 4 #force new fetch group
        #ITERATION 0
        nop
        lqd $(CURRENT_VAL_REG+0*NUMREGS), 0*REGBYTES($BUFFER_REG)
        #stall (waiting on CURRENT_VAL_REG)
        #stall
        #stall
        #stall
        #stall
        absdb $(PROCESSED_VAL_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS), $CONVER
        lnop
        cgtbi $(BOOL_TMP1_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS), 'a'-1
        lnop
        cgtbi $(BOOL_TMP2_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS), 'z'
        lnop
        #stall (waiting on BOOL_TMP2_REG)
        xor $(IN_RANGE_REG+0*NUMREGS), $(BOOL_TMP1_REG+0*NUMREGS), $(BOOL_TMP2_REG+
        lnop
        #stall (waiting on IN_RANGE_REG)
        selb $(CURRENT_VAL_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS),
        $(PROCESSED_VAL_REG+0*NUMREGS), $(IN_RANGE_REG+0*NUMREGS)
        lnop
        #stall (waiting on CURRENT_VAL_REG)
        nop
        stqd $(CURRENT_VAL_REG+0*NUMREGS), 0*REGBYTES($BUFFER_REG)
```

As you can see, this single iteration is wasting 8 cycles just waiting on registers to finish loading. In addition, it is wasting 7 opportunities for dual-issue. So even in this vectorized implementation, there is a lot of room for improvement!

You may be wondering why the program did not put `selb` and `stqd` in the same fetch group. You could have, but it would not have increased the speed of the program. Since `stqd` has to stall for the value of `CURRENT_VAL_REG` they would have had to be issued separately anyway, and you would not have gained any speed.

Sometimes instruction scheduling is a hassle. However, when used in conjunction with loop unrolling, it's not so bad. Since each iteration is using a different set of registers for computation, all you have to do is interleave computations from each iteration to fill up the time slots. So your new loop body looks like this:

Listing 9. Interleaved loop body minimizes dependency stalls

```
lqd $(CURRENT_VAL_REG+0*NUMREGS), 0*REGBYTES($BUFFER_REG)
lqd $(CURRENT_VAL_REG+1*NUMREGS), 1*REGBYTES($BUFFER_REG)
lqd $(CURRENT_VAL_REG+2*NUMREGS), 2*REGBYTES($BUFFER_REG)
lqd $(CURRENT_VAL_REG+3*NUMREGS), 3*REGBYTES($BUFFER_REG)
absdb $(PROCESSED_VAL_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS), $CONVER
absdb $(PROCESSED_VAL_REG+1*NUMREGS), $(CURRENT_VAL_REG+1*NUMREGS), $CONVER
absdb $(PROCESSED_VAL_REG+2*NUMREGS), $(CURRENT_VAL_REG+2*NUMREGS), $CONVER
absdb $(PROCESSED_VAL_REG+3*NUMREGS), $(CURRENT_VAL_REG+3*NUMREGS), $CONVER
cgtbi $(BOOL_TMP1_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS), 'a'-1
cgtbi $(BOOL_TMP1_REG+1*NUMREGS), $(CURRENT_VAL_REG+1*NUMREGS), 'a'-1
cgtbi $(BOOL_TMP1_REG+2*NUMREGS), $(CURRENT_VAL_REG+2*NUMREGS), 'a'-1
cgtbi $(BOOL_TMP1_REG+3*NUMREGS), $(CURRENT_VAL_REG+3*NUMREGS), 'a'-1
cgtbi $(BOOL_TMP2_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS), 'z'
cgtbi $(BOOL_TMP2_REG+1*NUMREGS), $(CURRENT_VAL_REG+1*NUMREGS), 'z'
cgtbi $(BOOL_TMP2_REG+2*NUMREGS), $(CURRENT_VAL_REG+2*NUMREGS), 'z'
cgtbi $(BOOL_TMP2_REG+3*NUMREGS), $(CURRENT_VAL_REG+3*NUMREGS), 'z'
xor $(IN_RANGE_REG+0*NUMREGS), $(BOOL_TMP1_REG+0*NUMREGS), $(BOOL_TMP2_REG+
xor $(IN_RANGE_REG+1*NUMREGS), $(BOOL_TMP1_REG+1*NUMREGS), $(BOOL_TMP2_REG+
xor $(IN_RANGE_REG+2*NUMREGS), $(BOOL_TMP1_REG+2*NUMREGS), $(BOOL_TMP2_REG+
xor $(IN_RANGE_REG+3*NUMREGS), $(BOOL_TMP1_REG+3*NUMREGS), $(BOOL_TMP2_REG+
selb $(CURRENT_VAL_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS),
$(PROCESSED_VAL_REG+0*NUMREGS), $(IN_RANGE_REG+0*NUMREGS)
selb $(CURRENT_VAL_REG+1*NUMREGS), $(CURRENT_VAL_REG+1*NUMREGS),
$(PROCESSED_VAL_REG+1*NUMREGS), $(IN_RANGE_REG+1*NUMREGS)
selb $(CURRENT_VAL_REG+2*NUMREGS), $(CURRENT_VAL_REG+2*NUMREGS),
$(PROCESSED_VAL_REG+2*NUMREGS), $(IN_RANGE_REG+2*NUMREGS)
selb $(CURRENT_VAL_REG+3*NUMREGS), $(CURRENT_VAL_REG+3*NUMREGS),
$(PROCESSED_VAL_REG+3*NUMREGS), $(IN_RANGE_REG+3*NUMREGS)
stqd $(CURRENT_VAL_REG+0*NUMREGS), 0*REGBYTES($BUFFER_REG)
stqd $(CURRENT_VAL_REG+1*NUMREGS), 1*REGBYTES($BUFFER_REG)
stqd $(CURRENT_VAL_REG+2*NUMREGS), 2*REGBYTES($BUFFER_REG)
stqd $(CURRENT_VAL_REG+3*NUMREGS), 3*REGBYTES($BUFFER_REG)

#Increment Pointer
ai $BUFFER_REG, $BUFFER_REG, 4*REGBYTES

#Exit if needed
cgt $IS_FINISHED_REG, $BUFFER_REG, $BUFFER_END_REG
brz $IS_FINISHED_REG, loop_start
```

This technique is called *software pipelining*, and this code only loses 2 cycles to stalls. However, it still does not make much use of dual-issue. In fact, there aren't many opportunities to do that at all in this code.

If you were to unroll the loop four more iterations, you could interleave each set of four so that one set of instructions was doing the loads while the other one was doing the executes, and that would save some clock cycles through dual-issue. However, for now, I will simply show how to save two clock cycles by adjusting the order of the `selb` and `stqd` instructions. Here is the new order:

Listing 10. Rescheduling instructions

```
        selb $(CURRENT_VAL_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS),
        $(PROCESSED_VAL_REG+0*NUMREGS), $(IN_RANGE_REG+0*NUMREGS)
        selb $(CURRENT_VAL_REG+1*NUMREGS), $(CURRENT_VAL_REG+1*NUMREGS),
        $(PROCESSED_VAL_REG+1*NUMREGS), $(IN_RANGE_REG+1*NUMREGS)
.align 3   ####Force to the start of a fetch-group
        #Next two issued concurrently
        selb $(CURRENT_VAL_REG+2*NUMREGS), $(CURRENT_VAL_REG+2*NUMREGS),
        $(PROCESSED_VAL_REG+2*NUMREGS), $(IN_RANGE_REG+2*NUMREGS)
        stqd $(CURRENT_VAL_REG+0*NUMREGS), 0*REGBYTES($BUFFER_REG)
        #Next two issued concurrently
        selb $(CURRENT_VAL_REG+3*NUMREGS), $(CURRENT_VAL_REG+3*NUMREGS),
        $(PROCESSED_VAL_REG+3*NUMREGS), $(IN_RANGE_REG+3*NUMREGS)
        stqd $(CURRENT_VAL_REG+1*NUMREGS), 1*REGBYTES($BUFFER_REG)
        stqd $(CURRENT_VAL_REG+2*NUMREGS), 2*REGBYTES($BUFFER_REG)
        stqd $(CURRENT_VAL_REG+3*NUMREGS), 3*REGBYTES($BUFFER_REG)
```

Simply by properly aligning a section of the program on what you want to be a fetch-group boundary and moving one instruction (the last `selb`) to a more opportune location, you can save two clock cycles. Note that without the `.align 3`, if the `selb` instructions were in the odd position while the `stqd` instructions were in the even position, you would not achieve dual-issue, because dual-issue *only occurs when the two instructions are both appropriately sequenced and aligned.*

Branch hinting

The SPU has no real *hardware* support for branch hinting. However, it makes up for this (and in some cases has the possibility of surpassing hardware support) by providing excellent *software* support for branch hinting.

Branch hinting is necessary on the SPU because mispredicted branches come at a high cost. It takes 18-19 cycles to recover from a mispredicted branch. In addition, by default, every branch encountered by the SPU is assumed to be not taken, including unconditional branches. What a branch hint does is specify to the processor that, for a specific branch instruction (also called a *hint-trigger address*), what address it is likely to branch to (called the *branch target address*). This allows the processor to prepare for the branch ahead of time (prefetching the instructions, for instance). Branch hints *never affect the logical outcome of a program*. They only affect the number of cycles required to run the program.

There are three branch-hinting instructions:

- `hbr hint_trigger, $register` -- This tells the processor that the branch instruction at the relative address `hint_trigger` is likely to branch to the address specified in register `$register`.
- `hbrr hint_trigger, branch_target` -- This tells the processor that the branch instruction at the relative address `hint_trigger` is likely to branch to the relative address `branch_target` (both are relative from the current instruction).
- `hbra hint_trigger, branch_target` -- The same as `hbrr`, except that `branch_target` is specified as an absolute address.

For a branch hint to be most effective (so that the branch does not stall at all), it must be set at least four instruction fetch-groups plus 11 cycles before the branch instruction. At minimum, the branch hint must be four instruction fetch groups before the branch instruction, or it will have no effect. It also may not be more than 255 instructions away (physically) from the branch that it hints for (the instruction itself only has space for 8 bits plus a sign bit for the relative offset of the hint trigger, which then has two zeroes concatenated at the end). For example, if it is four instruction fetch groups plus 3 cycles away from the branch instruction, the branch instruction will enter *hint stall* for 8 cycles, which, while not optimum, is still much better than the 18 cycles it would stall without the hint. Only one hint can be active at a time, and `sync` instructions, among other things, clear out any active hint.

The best place to use a hint in your code is before the loop. Since the loop will be more likely taken than not taken (at least for larger strings), you could give a symbolic name to your branch instruction, and hint before the loop that the branch is likely to be taken. The code change would look like this:

Listing 11. Hinted branches

```
        hbrr loop_branch_instruction, loop_start
loop_start:

        ##... conversions go here ... ##

        #Increment Pointer
        ai $BUFFER_REG, $BUFFER_REG, 4*REGBYTES

        #Exit if needed
        cgt $IS_FINISHED_REG, $BUFFER_REG, $BUFFER_END_REG
loop_branch_instruction:
        brz $IS_FINISHED_REG, loop_start
```

Because the hint is before the loop body, this code leaves your hint active for every iteration of the loop, but it only has to use one cycle.

Unfortunately, because the loop branch is so close to the return statement, you cannot predict both the loop branch and the return branch. However, if you thought that the branch loop was not likely to be taken (in this case, if the string is likely less than 64 characters) then you could hint the return address instead by changing the hint to:

```
#This assumes that $lr has the right address right now (true in our case)
hbr end_function, $lr
```

You can actually do some fairly advanced hinting behavior using register-based hint instructions. Just keep in mind the hint restrictions as well as the fact that hint instructions do take up a cycle of your program.

Conclusion

At the end, your optimized function should look like this:

Listing 12. Fully optimized conversion function

```
.data
.equ CONVERSION_FACTOR, 'a' - 'A'
.align 4
conversion_bytes:
        .fill 16, 1, CONVERSION_FACTOR

.text
.global convert_buffer_to_upper
.type convert_buffer_to_upper, @function
        .equ BUFFER_REG, 3
        .equ BUFFER_SZ_REG, 4
        .equ BUFFER_END_REG, 5
        .equ CONVERSION_BYTES_REG, 6
        .equ IS_FINISHED_REG, 7

        .equ CURRENT_VAL_REG, 8
        .equ BOOL_TMP1_REG, 9
        .equ BOOL_TMP2_REG, 10
        .equ IN_RANGE_REG, 11
        .equ PROCESSED_VAL_REG, 12

        .equ NUMREGS, 5
        .equ REGBYTES, 16
convert_buffer_to_upper:
        a $BUFFER_END_REG, $BUFFER_SZ_REG, $BUFFER_REG
        lqr $CONVERSION_BYTES_REG, conversion_bytes

        hbrr loop_branch_instruction, loop_start
loop_start:
        lqd $(CURRENT_VAL_REG+0*NUMREGS), 0*REGBYTES($BUFFER_REG)
        lqd $(CURRENT_VAL_REG+1*NUMREGS), 1*REGBYTES($BUFFER_REG)
        lqd $(CURRENT_VAL_REG+2*NUMREGS), 2*REGBYTES($BUFFER_REG)
        lqd $(CURRENT_VAL_REG+3*NUMREGS), 3*REGBYTES($BUFFER_REG)
        absdb $(PROCESSED_VAL_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS), $CONVER
        absdb $(PROCESSED_VAL_REG+1*NUMREGS), $(CURRENT_VAL_REG+1*NUMREGS), $CONVER
        absdb $(PROCESSED_VAL_REG+2*NUMREGS), $(CURRENT_VAL_REG+2*NUMREGS), $CONVER
        absdb $(PROCESSED_VAL_REG+3*NUMREGS), $(CURRENT_VAL_REG+3*NUMREGS), $CONVER
        cgtbi $(BOOL_TMP1_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS), 'a'-1
        cgtbi $(BOOL_TMP1_REG+1*NUMREGS), $(CURRENT_VAL_REG+1*NUMREGS), 'a'-1
        cgtbi $(BOOL_TMP1_REG+2*NUMREGS), $(CURRENT_VAL_REG+2*NUMREGS), 'a'-1
        cgtbi $(BOOL_TMP1_REG+3*NUMREGS), $(CURRENT_VAL_REG+3*NUMREGS), 'a'-1
        cgtbi $(BOOL_TMP2_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS), 'z'
        cgtbi $(BOOL_TMP2_REG+1*NUMREGS), $(CURRENT_VAL_REG+1*NUMREGS), 'z'
        cgtbi $(BOOL_TMP2_REG+2*NUMREGS), $(CURRENT_VAL_REG+2*NUMREGS), 'z'
        cgtbi $(BOOL_TMP2_REG+3*NUMREGS), $(CURRENT_VAL_REG+3*NUMREGS), 'z'
        xor $(IN_RANGE_REG+0*NUMREGS), $(BOOL_TMP1_REG+0*NUMREGS), $(BOOL_TMP2_REG+
```

```
        xor $(IN_RANGE_REG+1*NUMREGS), $(BOOL_TMP1_REG+1*NUMREGS), $(BOOL_TMP2_REG+
        xor $(IN_RANGE_REG+2*NUMREGS), $(BOOL_TMP1_REG+2*NUMREGS), $(BOOL_TMP2_REG+
        xor $(IN_RANGE_REG+3*NUMREGS), $(BOOL_TMP1_REG+3*NUMREGS), $(BOOL_TMP2_REG+
        selb $(CURRENT_VAL_REG+0*NUMREGS), $(CURRENT_VAL_REG+0*NUMREGS),
        $(PROCESSED_VAL_REG+0*NUMREGS), $(IN_RANGE_REG+0*NUMREGS)
        selb $(CURRENT_VAL_REG+1*NUMREGS), $(CURRENT_VAL_REG+1*NUMREGS),
        $(PROCESSED_VAL_REG+1*NUMREGS), $(IN_RANGE_REG+1*NUMREGS)
.align 3
        selb $(CURRENT_VAL_REG+2*NUMREGS), $(CURRENT_VAL_REG+2*NUMREGS),
        $(PROCESSED_VAL_REG+2*NUMREGS), $(IN_RANGE_REG+2*NUMREGS)
        stqd $(CURRENT_VAL_REG+0*NUMREGS), 0*REGBYTES($BUFFER_REG)
        selb $(CURRENT_VAL_REG+3*NUMREGS), $(CURRENT_VAL_REG+3*NUMREGS),
        $(PROCESSED_VAL_REG+3*NUMREGS), $(IN_RANGE_REG+3*NUMREGS)
        stqd $(CURRENT_VAL_REG+1*NUMREGS), 1*REGBYTES($BUFFER_REG)
        stqd $(CURRENT_VAL_REG+2*NUMREGS), 2*REGBYTES($BUFFER_REG)
        stqd $(CURRENT_VAL_REG+3*NUMREGS), 3*REGBYTES($BUFFER_REG)

        ai $BUFFER_REG, $BUFFER_REG, REGBYTES
        cgt $IS_FINISHED_REG, $BUFFER_REG, $BUFFER_END_REG
loop_branch_instruction:
        brz $IS_FINISHED_REG, loop_start

end_function:
        bi $lr
```

This code has been branch-eliminated, vectorized, loop-unrolled, instruction-scheduled, and branch-hinted. In other words, it's pretty darn fast. The next article switches to programming in C, but this information will still be useful for understanding what the compiler is trying (or at least should be trying) to do, and for analyzing the output of your compiler to understand where hand-coded assembly could give you better performance.

Stay tuned for the rest of this series: We'll explore coding the SPU in C, covering SPU extensions to C, and other higher-level optimization techniques.

Resources

- See the other parts in the Programming high-performance applications on the Cell BE processor series.
- Always keep your assembly language overview handy, as well as the instruction set architecture reference for more detailed information.
- ArsTechnica has an overview of SIMD architectures (before the Cell BE processor) as well as an introduction to the Cell BE's SIMD architecture.
- The Cell Broadband Engine Programming Handbook has a lot of interesting low-level details on the SPUs. Some interesting sections are:
  - Pages 75-76 and 687-688 discuss pipeline and latency issues.
  - Pages 768-772 describe why you need to use no-ops to take advantage of dual-issue rules, what the instruction pipeline looks like, and how instruction prefetch works. Page 772 describes the *hbrp* instruction which can be used to help out the prefetch schedule.

    - Pages 689-697 cover branch elimination and hinting.

- If you want to get really nuts with branch optimization, you should read the additional considerations the IBM compiler team uses for branch optimization.
- Additional optimization considerations and suggestions are available in the slide presentation Cell programming tips and techniques (PDF).
- Stay abreast of all things Cell BE: subscribe to IBM microNews.

About the author

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.