SPU pipeline examination in the IBM Full-System Simulator for the Cell Broadband Engine processor

Cycle-accurate simulation made easy

David Murrell

April 17, 2006

Find out the exact cycle where the SPE stalls, or identify a poor choice of branch predictions, using pipeline tracing in the Cell Broadband Engine™ (Cell BE) simulator.

This article describes the SPU pipeline configuration, performance metric, and trace facilities that are available in the IBM® Full-System Simulator for the Cell BE platform (available for public download on the IBM alphaWorks Web site -- see Resources). You can get information to show the detailed internal state of an SPU at each cycle of execution. Cell BE software developers who are concerned with tuning the low-level performance characteristics of their applications can use these mechanisms to identify stalls, operand dependencies, and so on. The article also covers the procedures necessary to configure SPU characteristics and to enable the inspection features and gives some guidance in interpreting the resulting output syntax and semantics.

Enabling the SPU tracing facilities in the simulator

The IBM Full-System Simulator for the Cell Broadband Engine Processor platform models the Cell BE processor. This article assumes you have access to an installation of the simulator and some familiarity with its basic operation. If not, please refer to Peter Seebach's developerWorks "Get started with the Cell Broadband Engine Software Development Kit" series, Part 1 of which outlines the required procedures. Some familiarity with the SPU pipeline operation is also assumed; for more on this see also SPE Book IV.

You can configure each of the models of the eight SPE cores (SPUs) to simulate code execution either in a purely functional manner or at the cycle level by switching the mode of its operation between "instruction mode" and "pipeline mode." Toggling between modes for a particular SPU is accomplished from the simulator's TCL command line (CLI) or from the GUI. For instance, for the CLI command to switch SPU 0 to pipeline mode for the "mysim" simulation instance (assuming the simulator "mysim" and SPU 0), the command is:

```
mysim spu 0 set model pipeline
```
You can toggle from the GUI in a couple of different ways. Within the control panel's processor tree pane, double-click the model mode entry of the desired SPE. The model will toggle between pipeline and instruction modes of operation:

**Figure 1. SPE0, set to the pipeline model**

Alternatively, you can use the SPU Modes dialog to toggle conveniently between "pipeline" and "instruction" modes. Use the "SPU Modes" control panel button to launch the SPU Modes dialog, then toggle the mode for the SPUs as appropriate:

**Figure 2. The SPU Modes window**

**Configuring the SPU pipeline operation**

By default, the SPU pipeline model characteristics are configured to correspond to the Cell Broadband Engine hardware. This is the only configuration that has been validated against BE SPU logic, and is the configuration the SDK compilers target when generating SPU code. Because the Full-System Simulator for Cell BE is a research tool, SPU pipeline characteristics can be
modified to facilitate academic exploration of alternative configurations. Experimenters wishing to exploit the SPU model configuration commands should bear several important aspects in mind:

- SPU pipeline characteristics can be configured which are not feasible to implement in hardware.
- Alternative configurations are untested and unsupported. Many complex and subtle interactions are modeled which are not guaranteed to operate properly in alternative configurations. Unexpected behaviors which produce misleading data is typically the result.
- SDK compilers assume the default Cell BE SPU configuration when scheduling generated code. Compiled code might not execute optimally on alternative configurations.

SPU characteristics are defined by the simulator's "cell" configuration object. Alterations must be made to a mutable copy of the "cell" object, and applied when constructing the simulated machine. The experimenter will need to modify the simulator's TCL startup script (for example, .systemsim.tcl) accordingly:

```
# Create a mutable copy of the cell configuration object (for instance, "myconf")
define dup cell myconf
# Configure characteristics here. Refer to the table below for parameters and values
myconf configure parameter value
myconf configure parameter value
...  
# Construct the "mysim" simulated machine using the "myconf" configuration object
define machine myconf mysim
```

Table 1 lists SPU pipeline configuration parameters:

### Table 1. SPU pipeline model parameters

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>BE default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spu/pipe/class/depth</td>
<td>See Table 2</td>
<td>Execution latency for instructions of type <code>iclass</code> (in SPU clock cycles)</td>
</tr>
<tr>
<td>spu/pipe/class/stall</td>
<td>See Table 2</td>
<td>Subsequent instruction issue stall cycles introduced by instructions of type <code>iclass</code> (in SPU clock cycles)</td>
</tr>
<tr>
<td>spu/feature/sp-dual</td>
<td>TRUE</td>
<td>Allow instruction dual-issue with single precision floating point operations</td>
</tr>
<tr>
<td>spu/feature/dp-dual</td>
<td>FALSE</td>
<td>Allow instruction dual-issue with double precision floating point operations</td>
</tr>
<tr>
<td>spu/feature/ls-line-contend-load-only</td>
<td>FALSE</td>
<td>Model contention between instruction line fetch requests and register file loads only</td>
</tr>
<tr>
<td>spu/frequency</td>
<td>3200M</td>
<td>SPU clock frequency (in Hz)</td>
</tr>
</tbody>
</table>

The "sp-dual" feature determines whether another instruction can be issued simultaneously with a single precision operation. "dp-dual" governs the same behavior for double precision operations. The "ls-line-contend-load-only" feature specifies whether instruction line fetch contends with register file load operations only (TRUE), or with both load and store operations (FALSE). In the SPU model, this contention point is checked whenever an instruction line prefetch request is ready to enter the load/store pipe and a load/store operation is valid in stage "n" of the odd execution pipe.
The depth parameter determines the latency needed to execute an instruction of the indicated class (from the time an instruction of that type enters the top of the execution pipe). The stall parameter specifies the minimum number of cycles the machine will delay before issuing any subsequent instruction to the execution pipe receiving an instruction of the indicated class. Table 2 gives pertinent depth and stall information for each "iclass":

**Table 2. SPU pipe depth and stall by instruction class**

<table>
<thead>
<tr>
<th>iclass name</th>
<th>Execution pipe</th>
<th>BE default</th>
<th>Instruction types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Depth</td>
<td>Stall</td>
</tr>
<tr>
<td>BR</td>
<td>odd (1)</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Branch</td>
</tr>
<tr>
<td>FP6</td>
<td>even (0)</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Single precision floating point</td>
</tr>
<tr>
<td>FP7</td>
<td>even (0)</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Integer multiply, integer/float conversion, interpolate</td>
</tr>
<tr>
<td>FPD</td>
<td>even (0)</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Double precision floating point</td>
</tr>
<tr>
<td>FX2</td>
<td>even (0)</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Load immediate, logical operations, integer add/subtract, sign extend, count leading zeros, select bits, carry/borrow generate</td>
</tr>
<tr>
<td>FX3</td>
<td>even (0)</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Element rotate/shift</td>
</tr>
<tr>
<td>FXB</td>
<td>even (0)</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Special byte operations</td>
</tr>
<tr>
<td>NOP</td>
<td>odd (1)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No-op</td>
</tr>
<tr>
<td>LS</td>
<td>odd (1)</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Loads/stores, branch hints</td>
</tr>
<tr>
<td>NOP</td>
<td>even (0)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No-op</td>
</tr>
<tr>
<td>SHUF</td>
<td>odd (1)</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Shuffle bytes, quadword rotate/shift, estimate, gather, form select mask, generate insertion control</td>
</tr>
<tr>
<td>SPR</td>
<td>odd (1)</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Channel operations, move to/from SPR</td>
</tr>
</tbody>
</table>

For instance, fully pipelined double-precision operations could be simulated by enabling the "dp-dual" feature (setting spu/feature/dp-dual to TRUE), and by removing issue stalls for instructions following DP operations (setting spu/pipe/FPD/stall to 0) in the mutable configuration object prior to constructing the machine.

Be aware that in the current version of the SPU model, modifying the load/store (LS) iclass depth and stall configuration parameters produces unpredictable behavior.

**Capturing simple SPU performance metrics**

The Cell BE Full-System Simulator provides a number of mechanisms to instrument applications and collect performance data. A thorough treatment of this subject is beyond the scope of this article, so I will briefly describe the most commonly used methods.
The SPU model provides several event counters that can be controlled through TCL commands, as well as by the SPU applications themselves. Counters for SPU n can be reset using the simulator's "mysim spu n stats reset" TCL command. The Cell BE SDK includes declarations for functions to start, stop, and clear the model's event counters. These functions resolve to specially encoded No-Ops (AND x,x,x) when the SPU application is compiled. As the application is executed in simulation, the No-Op instructions are intercepted by the model to control the event counters.

```
#include "profile.h"
...
prof_cp0(); /*!< Clear performance counters, inserts "AND 0,0,0", same as prof_clear() */
prof_cp30(); /*!< Commence event counting, inserts "AND 30,30,30", same as prof_start() */
    /*!< Code sequence to measure */
prof_cp31(); /*!< Cease event counting, inserts "AND 31,31,31", same as prof_stop() */
...
```

When each of these counter control operations is interpreted by the SPU model, a message is displayed at the simulator's output window:

```
SPUn: CPx, instruction count(non-NOP count), cycle count
```

n is the SPU number and x is the control function code (AND x,x,x). The total number of instructions executed (instruction count), non-NOP instruction count subset, and SPU clock cycles elapsed since the counters were last cleared, follows that information on each line.

More detailed event counter information can be displayed using the simulator's "mysim spu n stats print" TCL command.

## Enabling the SPU tracing facilities in the simulator

SPU tracing provides much greater visibility into the internal operations of the simulated machine. Pipeline tracing is enabled by issuing the following TCL CLI commands in the simulator:

```
simdebug set SPU_DISPLAY_ISSUE 1
simdebug set SPU_DISPLAY_EXEC 1
```

Alternatively, you can do this through the GUI. Bring up the simulator's debug control panel by pressing the "Debug Controls" button. Then, select the "SPU_DISPLAY_ISSUE" and "SPU_DISPLAY_EXEC" options from the debug controls dialog:
When an SPU model is configured to execute in pipeline mode and the debugging facilities are enabled as described, a text frame will be written to standard output (stdout) each cycle. A considerable amount of text information is displayed each cycle. Simulation performance will also degrade while tracing is enabled, due to the run time overhead required to collect and format trace data.

Interpreting the SPU trace output

Below is a sample SPU trace frame captured during the execution of an FFT algorithm written for the Cell BE platform (at cycle 229). Trace frames are dumped at the end of each cycle, after all updates, execution effects, and state changes have been made by operations in the current cycle, and before any pipelines are advanced for the next cycle. This is an important factor to keep in mind because the simulator's SPU model is clock-synchronous and is updated in reverse pipeline order. Updates occur to the execution pipe, then issue, then fetch. I will attempt to explain areas where this is an issue in interpreting the trace frames.
Figure 4. And this is just one cycle!

The SPU trace frame is composed of a number of textual regions, each of which describes an aspect of the SPU core pipeline state. The balance of this article provides detailed descriptions for each of these regions. See a legend for a typical SPU trace frame below. Table 3 provides a list of links you can use to navigate to any particular region of interest depicted in the annotated trace frame legend of Figure 5.

Table 3. Map

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>Cycle and Instruction Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Misspredict Status</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Hint Status</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Prefetch Unit Status</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Prefetch Status (Load/Store)</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Hint Target Buffer</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>Inline Prefetch Buffers</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>Predicted Path Buffer</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>Issue Pipe 0 (Even)</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>Issue Pipe 1 (Odd)</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>Execution Pipe 0 (Even)</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>Execution Pipe 1 (Odd)</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Pipe 0 Operand Dependencies</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Pipe 1 Operand Dependencies</td>
<td></td>
</tr>
</tbody>
</table>
Figure 5. The components of an SPU trace

![Image of SPU trace components](image)

**Cycle and instruction count**

Figure 6. Cycle and instruction count

This line indicates the current cycle number (in SPU processor clocks) and total instruction count (of executed instructions). Valid instructions are counted and interpreted in stage "o" (letter "oh", see section on execution pipes for details). The instruction count shown reflects having been incremented this cycle by the number of valid (non bubble) instructions present in stage "o." (Recall that the trace frame shows the state at the end of the cycle).

Return to legend.

**Mispredict status**

Figure 7. Mispredict status

Mispredict state is shown on this line. The number before the arrow (->) indicates the mispredict state (0 through 5) at the end of the current cycle. Zero indicates not under mispredict. The address after the arrow indicates the local store address of the next program counter (here, 0x003e0) -- since the trace frame shows state after execution has updated the SPU. The PC is updated by instructions executed in stage "o" (see section on execution pipes for details), by SPU interrupts, and by MMIO writes.

Return to legend.
### Hint status

**Figure 8. Hint status**

The hint state line address before the arrow (\(\rightarrow\)) is the local store address of the hinted instruction (usually a branch), in this case 0x0040c. The address after the arrow is the local store hint target address. If the hint is valid, it will be suffixed with an asterisk (*).

Here, we have a valid hint which was triggered when the branch at 0x0040c was loaded into the predicted path buffer. Upon triggering, instruction prefetch is re-directed to follow the last half-line address (64 bytes, or 16 instructions) of the hint target buffer. There is a pending request for line 0x00300 which was initiated for this reason (refer to the section on **prefetch unit state** for details).

Instructions from the 128-byte line at the hint target address have already been loaded into the hint target buffer, and will be fed into the predicted path buffer for subsequent issue as soon as the hinted branch is fetched into issue stage "g." See below for further description of the **hint target buffer** and hint instruction behavior.

Return to [legend](#).

### Prefetch unit status

**Figure 9. Prefetch unit status**

The prefetch unit state trace frame information has two parts. The first, labeled "**pre-fetch**" shows the pre-fetch request queue -- a four-deep queue of local store line-sized addresses (128-bytes, or 32 instructions each). The oldest request is furthest right in the queue. Here you see two pending requests for the 128-byte line at address 0x00300 (the oldest) and the adjacent line at address 0x00380. These requests were initiated by hint triggering and follow the address of the second hint target buffer half-line in **ilbh2**.

The second part of the prefetch unit trace frame information, labeled "**pre-fetch ls**" shows outstanding 128-byte line instruction prefetch requests being processed by the SPU load/store unit pipeline. The pipeline shown here is six stages long. Requests which reach the (right) end of the request queue are accepted into the load/store pipe from the request queue at a maximum rate of one every other cycle.
A new request will stall at the head of the request queue if:

1. A request is already in either of the first two stages (leftmost 2 positions) of the load/store pipe (every-other cycle entry rule).
2. A load/store instruction will execute at stage "o" of the execution pipe during the next cycle (local store arbitration rules dictate instruction fetch requests are lower in priority than load/store register file accesses). Thus, if stage "n" has a "hole" shown at the end of the current cycle (no valid load/store instruction), the prefetch request will have been allowed into the load/store pipe for the next cycle.

**Figure 10. A pending request**

Here you see a request (0x00300) arbitrating for the load/store pipe. The load/store pipe currently has no outstanding prefetch requests, but a load instruction (lqx at 0x003e0) is pending execution in stage "n" (will be at stage "o" next cycle). The request will not be allowed to enter the load/store pipe until a "hole" (non load/store instruction) opens in execution pipe stage "n." This will occur at the end of cycle 231, after the bubble currently at stage "l" (letter "el") has made its way to stage "n," two cycles from now. See below for more detail on interpreting the execution pipeline information.

Each prefetch request address is prefixed by either a number or a "?" (separated from the address by a dash). Numbers indicate the destination of the prefetch: 1 -> ilb1, 2 -> ilb2, and 3 -> ilbh.

Here, the request for line 0x00300 (designated 1-0x00300) is destined for ilb1 and for line 0x00380 (designated 2-0x00380) is destined for ilb2.

The "?" designation is given to "stale" inline prefetch requests. Recall that hint triggering causes inline prefetch to be redirected to obtain lines whose address follows ilbh2. Any prefetch requests which are already queued for older inline prefetches could be "stale," and would need to be flushed to allow the re-directed requests to proceed to arbitrate immediately for the load/store pipeline.

When the hint triggers, any prior outstanding inline prefetch requests are marked as stale. Stale requests are only candidates for disposal and will be processed normally if not discarded prior
to load/store pipe entry. After the predicted path buffer is loaded from ilbh1 (in other words, the hinted branch is fetched to issue stage "g"), any remaining stale prefetch requests are flushed from the request queue.

The mechanisms to queue and process outstanding prefetch requests differ somewhat between the SPU simulator and the SPU implementation. Modeling "stale" prefetch requests is an artifact of the SPU simulator approximation.

Return to legend.

**Hint target buffer**

**Figure 11. Hint target buffer**

The hint target buffer (ilbh) holds a maximum of one 128-byte line (32 instructions) beginning at the local store address of the hint target. The trace frame shows the hint buffer split into two half-lines (ilbh1 and ilbh2), each holding a maximum of 64 bytes (16 instructions). The half-line buffers are valid if marked with an equal sign (=), and invalid if marked with an X.

A prefetch request to load the hint buffer is initiated when a hint instruction (hbr-type without "P" bit set) is executed (reaches execution stage "o"). This request takes precedence over any pending inline prefetch requests. After the request is accepted by the load/store pipeline and subsequently reaches the end of the load/store pipe, 128 bytes (32 instructions) will be loaded into the hint buffers. Although a full 128-byte line is read, the actual number of valid instructions in the hint buffers depends upon whether the hint target is aligned on a 128-byte boundary. Here, the branch at 0x0040c targets address 0x00298, which is offset 0x18 (24 bytes, or six instructions) into the line beginning at 0x00280. Hence, only 10 of the 16 instructions held in ilbh1 will be valid.

Instructions held in the hint buffers will remain valid until:

1. The pipeline is flushed.
2. A new prefetch request initiated by an executed hint instruction reaches stage three or four of the load/store pipeline.

Valid instructions from the hint buffer will be fed into the predicted path buffer when the hinted instruction (here, the branch at 0x0040c) is loaded into stage "g" of the issue pipeline. Note that transferring instructions from the hint buffer to the predicted path buffer does not invalidate the contents of the source hint buffer.

Return to legend.
### Inline prefetch buffers

**Figure 12. Inline prefetch buffers**

The two inline prefetch buffers (ilb1 and ilb2) each hold 128-byte lines (a maximum of 32 valid instructions, depending upon alignment). Each buffer is split into 64-byte half-lines (a maximum of 16 valid instructions). The two halves of ilb1 are labeled "ilb11" and "ilb12." The two halves of ilb2 are labeled "ilb21" and "ilb22." Valid buffers are marked with an equal sign (=*), and invalid buffers are marked with an X.

A request to load an inline prefetch buffer is generated:

1. For the address following ilbh2 when a hint is triggered (the hinted branch is loaded into the predicted path buffer).
2. For the corrected address of a misprediction (in mispredict cycle/state three).
3. For the address following the second half-line of ilb1 or ilb2 when that half-line is loaded into the predicted path buffer.

Inline prefetch buffers are loaded with up to 32 instructions once the request reaches the end of the load/store pipeline.

Inline half-line prefetch buffers are invalidated:

1. When instruction content is transferred to the predicted path buffer
2. During pipeline flush
3. At mispredict cycle/state two
4. Eight cycles after a hint is triggered

Note that condition (1) above necessitates eventual inline buffer re-fetch, since the process of transfer to the predicted path buffer is "destructive" (as compared to the hint buffer which is not affected by predicted path transfer).

Return to legend.

### Predicted path buffer

**Figure 13. Predicted path buffer**
The predicted path buffer is a half-line (64 byte, 16 instructions) wide and directly feeds the issue pipelines. Upon hinted branch entry into issue stage "g," the hint buffers (ilbh1 and ilbh2) non-destructively transfer data into the predicted path buffer for subsequent issue. Otherwise, the inline prefetch buffers (ilb11, ilb21, ilb21, or ilb22) will in turn destructively transfer data to the predicted path buffer for issue.

The predicted path buffer is marked valid by an equal sign (\(=\)), or invalid by an \(X\). The next fetch offset within this buffer is denoted in parentheses following the local store address of the half-line in the buffer. Here, the predicted path buffer was filled with the half-line (0x00400) contents of ilb11 (which has been invalidated as a result). The first pair of instructions from the predicted path buffer have been sent to the top of the issue pipeline (stage "g") -- the shufb at 0x00400 in pipe 0, and the shufb at 0x00404 in pipe 1. Fetch will proceed to the pair of instructions at offset 0x8 within the predicted path buffer on the next cycle.

Prefetch miss occurs whenever the predicted path buffer cannot be filled with valid instructions from either the inline or hint target half-line buffers.

Return to legend.

Issue pipes

Figure 14. Issue pipes

The uppermost stages ("g" through "j") of the two SPU pipelines handle IN ORDER issue of instructions from the predicted path buffer. No shifting of instructions occurs between the two issue pipelines (to backfill vacancies opened from single issues).

The left-hand column of four lines corresponds to issue pipe 0 (even). The right-hand column corresponds to issue pipe 1 (odd). Instructions from the predicted path buffer at even effective addresses fill pipe 0, while instructions at odd effective addresses fill pipe 1. Each of the four lines corresponds to an issue pipeline stage, and shows the local store address and disassembly of the instruction resident in the given stage. Instructions which are valid are marked by an equal sign (\(=\)), while invalid slots (bubbles) are marked by \(X\). Valid instructions which reach stage "j" are issued to either execution pipe 0 or pipe 1, depending up on the instruction class. Note that the determination of issue pipe for a given instruction is made based on its address, while the determination of execution pipe is made based on the instruction's class. Refer to Table 4, provided in the "execution pipe" section below.

Two instructions at stage "j" of issue pipes 0 and 1 might be issued simultaneously (dual-issue) when:
1. There is no cross issue (or, the instruction at stage J of issue pipe 0 maps to execution pipe 0, and the instruction at stage J of issue pipe 1 maps to execution pipe 1).
2. There is no operand dependency. The following two possibilities can cause dependency stalls:
   a. A source register needed by the issue candidate instruction is the target of a valid unexecuted instruction in the execution pipe.
   b. A source register needed by the candidate in issue pipe 1 is the target of the instruction in issue pipe 0.
3. There are no structural issue stalls:
   a. Double-precision floating point instructions impose a six-cycle stall between consecutive issues.
   b. No instructions can be dual-issued with a double-precision instruction.

If no instructions can be issued for a given cycle, the issue pipelines will not advance (stall) and no instructions will be consumed out of the predicted path buffer. Bubbles (invalid stop instructions) will be inserted into the affected execution pipes each cycle until the issue stall conditions are resolved. Instructions in the issue pipes might be invalidated when:

1. There is a pipeline flush.
2. Mispredict cycle/state five is reached.

Return to legend.

Execution pipes

Figure 15. Execution pipes

Execution pipelines follow the same display conventions as the issue pipelines. You can think of pipe stage "jj" (marked with an asterisk (*) as the top of the execution pipelines. Execution pipelines do not stall.

Instructions are mapped by issue to either pipe 0 (even) or pipe 1 (odd) depending upon the class of instruction (or, the pipelines are asymmetric, having different functional units assigned to each pipeline). In the sample trace frame, all of the instructions in both issue pipes will be sent to execution pipe 1, since they are all either shuffles, loads, or stores. The latency required to
execute each instruction also varies by instruction type. The table below provides timing and pipeline assignment for each instruction class:

**Table 4. SPU instruction class and timings**

<table>
<thead>
<tr>
<th>Pipe</th>
<th>Instruction class</th>
<th>Execution timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Single precision floating point</td>
<td>6 cycles</td>
</tr>
<tr>
<td></td>
<td>Double precision floating point</td>
<td>7 cycles (6 cycle issue stall)</td>
</tr>
<tr>
<td></td>
<td>Integer multiply, integer/float conversion, interpolate</td>
<td>7 cycles</td>
</tr>
<tr>
<td></td>
<td>Load immediate, logical operations, integer add/subtract, sign extend, count leading zeros, select bits, carry/borrow generate</td>
<td>2 cycles</td>
</tr>
<tr>
<td></td>
<td>Element rotate/shift, special byte operations</td>
<td>4 cycles</td>
</tr>
<tr>
<td>1</td>
<td>Loads/stores, branch hints, channel operations, move to/from SPR</td>
<td>6 cycles</td>
</tr>
<tr>
<td></td>
<td>Shuffle bytes, quadword rotate/shift, estimate, gather, form select mask, generate insertion control, branch</td>
<td>4 cycles</td>
</tr>
</tbody>
</table>

Return to [legend](#).

**Operand dependency information**

**Figure 16. Operand dependency**

The system simulator's SPU model interprets all valid instructions at stage "o." The effects of differing execution unit pipeline lengths are modeled by "releasing" target operands from dependency calculations after the appropriate latency has expired. The two rightmost columns of the execution pipe section of the trace frame show the dependency information.

The "**P0**" column describes operand dependencies for instructions in execution pipe 0. The "**P1**" column describes dependencies for instructions in execution pipe 1. The number in parentheses is the target register number for the value to be produced by the corresponding instruction. For instance, the P1 column shows 8 for the lqx at 0x003e4 in stage "m" and 10 for the lqx at 0x003e0 in stage "n." Note that the shufb instruction ready at issue pipe 0 stage "j" (for execution pipe 1) is being stalled because of the register 10 source operand dependency. This is the cause behind the
insertion of bubbles (the three invalid stop instructions) at the top of execution pipe 1 until the lqx at 0x003e0 completes execution and writes its target register.

The value -1 in the dependency columns indicates the corresponding instruction has no targets (or is a bubble). The value 128 indicates the instruction’s target has been “written” (released) and will therefore no longer prevent instructions from issue due to source/target operand dependencies.

Where to go from here

The Cell BE simulator provides extremely detailed analysis on a cycle-for-cycle basis of the current state of a simulated SPE. The output is densely packed, but now that you've got an overview of the components, you can analyze performance carefully, identifying the code paths where performance is hurting. Once you know where a branch is being mispredicted, or an algorithm is stalling waiting for results, you may be able to reorganize code to reduce or eliminate stalls.

It might take a bit of practice to make effective use of this information; especially the first few times you have to track something down, budget extra time for figuring out what to look at, and for learning your way around the processor's architecture.