彎曲評論

科技 · 人物 · 潮流

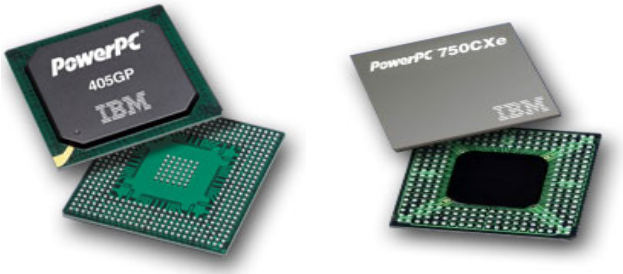
PowerPC and Linux Kernel Inside

陈怀临，首席科学家

《弯曲评论》

www.tektalk.cn

huailin@tektalk.cn



前言：

《PowerPC and Linux Kernel Inside》一书编著于 2002 年并发布在 www.xtrj.org 网站上。起笔的原因是当时整个工业界，出版界和 Linux 社区没有任何关于 Linux 在 PowerPC CPU 实现方面的分析，笔者基于 Linux 2.4.2 代码对 Linux Bootloader 和 Kernel 在 IBM PPC405 和 6xx/750 等 CPU 上的芯片相关实现部分进行了阅读，分析和加注。在编著本书的同时，笔者也将相关的 PowerPC 的规约部分综合整理于此，如 PowerPC 的 EABI 等等，目的是提供给 PowerPC 相关的程序员一个尽可能完整的关于 PowerPC 的视野。现整理发表于《弯曲评论》。

PowerPC 是一个 CPU 的规约，不同的厂商，如 IBM, FreeScale 等都有自己相应的实现和低，中和高端 CPU 产品系列。这些芯片，除了遵守 PowerPC 的基本规约外，都有一些厂商自己的扩展。因此，PowerPC 方面的操作系统程序员在工作中一定要细心的阅读来自厂商的芯片规约。如果阅读 Linux 底层代码，许多关于芯片的宏定义和代码逻辑跳转其实就是为了解决这些芯片之间细微差别的。

对于没有接触过 PowerPC 的读者，笔者建议不要直接进入 Linux/PPC 的实现细节研究，而是应该从理解 PowerPC 的基本规约，通用寄存器约定，控制寄存器约定，MMU 和缓存逻辑等方面有一个初步了解着手。

本书章节组织如下：

笔者在编著本书的时候，鉴于整个业界都没有这方便的数据，采用了英文的写作。

鉴于笔者在操作系统和 CPU 方面的知识和经验还很肤浅，本书一定会有错误和纰漏之处，希望读者见谅并指出。也非常希望读者能与笔者联系，技术交流。

1. Embedded PowerPC Family

The demand for high-performance embedded processors has opened up market opportunities for the IBM PowerPC[a] processor and other 32-bit RISC processors. IBM's PowerPC 400 family of processors sometimes referred to as embedded controllers, because of their high level of integrationÑwere developed to meet the needs of developers of embedded applications. However, embedded applications sometimes require processors with even higher performance than that currently provided by the PowerPC 400 family. In this case, many application designers are using PowerPC 600 family processors originally developed for desktop and workstation applications.

PowerPC processors are often chosen for embedded applications because they provide a wide range of price and performance selections andÑsince they are all based on the PowerPC architecture have a common development environment across the entire product range.

1.1 PowerPC 600 Family Processor Types

There are many processor types in the PowerPC 600 family, but only a subset are being recommended for embedded designs. The 601 processor is not a pure PowerPC processor; it is a hybrid that was used as a transition from POWER to PowerPC architecture and is no longer being made available to customers. The 603e, 603ev, 603v2, 740, and 750 processors are all being supported in embedded applications. The 603 processor was the first pure PowerPC processor. The others were follow-on designs with higher performance and/or larger caches, but they are similar to one another from a programming point of view. The 750 processor also has a built-in L2 cache controller, and the 740 and 750 processors have two integer execution units which enhance their performance beyond that of the 603 processors. The EM603e is an embedded version of the 603e, 603ev, and 603v2 processors, which are produced at reduced cost. The cost is reduced by not testing the floating-point unit, which increases yield, in some cases by

using plastic packaging, which is less expensive than ceramic. The 602 processor is the only 600 family processor that was designed specifically for embedded use. It is priced lower than the other 600 family processors, but lacks the level of integration provided by the 400 family processors. From a programming point of view, it differs from the rest of the 600 family in that it only supports single precision floating-point, and some of the PowerPC Book I instructions have been eliminated and must be emulated in software. It has a different bus than the other 600 family processors, which allows it to have a reduced pin count. The 604, 604e-v1, and 604e-v2 processors are being used in many applications, but are not being recommended for new embedded designs, because the other 600 family chips provide an overlapping range of price and performance and can be better supported if the 604 variants do not need to be supported.

1.2. Using PowerPC 600 Family Processors in Embedded Applications

Because the 600 family processors do not have an integrated DRAM controller like the 400 family processors, system designs require a companion chip set like the IBM27-82660 memory controller and PCI bridge chip to connect the processor to memory and PCI expansion slots. This chip set works with all of the 600 family processors except for the 602 processor.

Programming Differences

When migrating an embedded system from a PowerPC 400 family processor to a 600 family processor, some of the software must be modified. If the software was designed for upgradability, the amount of change should not be significant. If you are using an operating system that supports PowerPC processors, the differences may be transparent to the application software.

Configuration Registers

The 600 family processors contain three configuration registers. The HID0 and HID1 registers are processor-specific. The Machine State Register (MSR) is similar to that of

the 400 family. One difference is that on exceptions for the 600 family only the low 16 bits of the MSR are saved. Refer to the processor user manual, Embedded Market Solutions, publication CD# SC09-
3032-05, for details of the HID0, HID1, and MSR, or visit Web site for online manuals at: http://www.chips.ibm.com/products/embedded/chips/sheets.html

Memory Configuration

The configuration requirements for the 600 family processors are more restrictive than those of the 400 family processors. The reset vector for the 600 family processors is at address 0xFFF00100, while for the 400 family processors, it is at the address 0xFFFFFFFC. In both cases, the user must provide
some type of nonvolatile memory, such as FLASH or ROM, at these addresses. The 600 family must also have DRAM at the lowest address range. The IBM27-82660 memory controller enforces these restrictions as well.

PowerPC 600 family processors that interface to FLASH or ROM through the IBM27-82660 memory controller will not access this memory correctly once the cache is enabled. The way around this restriction is to copy the ROM contents to DRAM during initialization, before caches are enabled. This increases the performance of the system, but also increases the cost.

Floating-Point Unit

The only difference between the 600 and 400 families in the area of the user instruction set, or PowerPC Book I instructions, is that floating-point instructions are supported by most of the 600 family processors and must be emulated for the 400 family processors. The 600 family processors contain 32 64-bit Floating-Point Registers and a Floating-Point Status and Control Register (FPSCR). There are three bits in the MSR that affect Floating-Point Unit (FPU) operations:

The FP (Floating-Point Available) is used to optimize context switches not involving FPU Registers.

The FE0 and FE1 (Floating-Point Exception Mode) bits control how floating-point exceptions are to be handled by the processor.

Many embedded applications do not require an FPU. In that case, the 602 processor or the EM603e may be better suited to the application.

Memory Manager Unit

Many embedded applications do not require a Memory Manager Unit (MMU), and some embedded processors do not even contain virtual memory capabilities. However, many of todayÕs embedded systems are being designed to take advantage of memory protection and other facilities made possible by a virtual mode MMU. The MMU design of the 600 family processors is completely different than that of the 400 family processors. The 600 family processors implement the MMU specifications provided by the PowerPC operating environment architecture (OEA) for PowerPC processors. The implementation allows for 4 GB of effective address space with a 4-KB page size and 256-MB segment size. The 600 family allows for page address translation using 16 Segment Registers (SRs) or block address translation using Block Address Translation (BAT) Registers.
The OEA specifies a hash table implementation for managing Translation Look-aside Buffers (TLBs). Some 600 family processors implement this entirely in hardware, and others require software assist. The software developer is not restricted to only this method.
The 600 family contains the following MMU-related Special Purpose Registers (SPRs) that are not in the 400 family:

* Block Address Translation Registers: The 600 family contains four instruction and four data BAT register pairs for virtual memory management.
*Segment Registers: The 600 family contains 16 SRs for virtual memory management.
*Software Table Search Registers: Some of the 600 family processors have built-in logic to search hash tables.

Others provide registers such as DMISS, DCMP, HASH1, HASH2, IMISS, ICMP, and RPA for software assist.
* SDR1: The SDR1 used to locate the hash table.

Cache Control

Defining regions of memory and memory-mapped I/O areas that are noncacheable is not as straightforward on the 600 family processors as it is on the 400 family processors. The recommended way of doing this is to use BAT Registers. This requires that the data address translation bit (DR) in the MSR be enabled.
The usual configuration has one BAT Register pair set cacheable for DRAM, and the other three pairs set noncacheable for memory-mapped I/O space.

Exceptions

The 600 family processors differ from the 400 family processors in the area of exceptions. Differences exist in the number of levels, the base address of the vectors, and the function of the exceptions.

The 600 family allows for two base addresses of exception vectors. The addresses of the exception vectors are controlled by the Exception Prefix bit (IP) in the MSR. Like the 400 family processors, the 602 processor has the upper 16 address bits of the exception vector base address as fully programmable.
The 400 family processors have a dual exception level design, while the 600 family has a single exception level design. The second pair of Save and Restore Registers SRR2 and SRR3 do not exist in the 600 family.

Byte Ordering

As in the 400 family, the 600 family processors default to run in big-endian mode. There are bits in the MSR register to run in PowerPC little-endian mode in normal and interrupt levels. When using the IBM27-82660 memory and PCI bridge controller, the system can be configured to run in true little-endian mode as well.

Timers

The timer facilities of the 600 family are not as versatile as those of the 400 family. The Decrementer Register (DEC) is a 32-bit register used for timing. Software can load values into the decrementer that will be decremented once every four bus clock cycles. When bit 0 (the most significant bit in the register) transitions from 0 to 1, a decrementer interrupt will occur unless exceptions are blocked via the EE bit in the MSR.

Alignment

The 600 family processors will handle most misaligned memory accesses. However, the following will cause alignment exceptions, which must be handled by software:

* Misaligned floating-point load or store
* The operand of lmw, stmw, lwarx, or stwcx not word aligned
* A little-endian access is misaligned
* A multiple or string access in little-endian mode
* The operand of a dcbz is in a page that is write-through or cache-inhibited

Time Base

The time base is a 64-bit register that is automatically incremented at regular intervals. The time base of the 600 and 400 families differs in both the rate at which it is incremented and how it is accessed by the software. The time base of the 600 family is incremented once every four bus clock cycles, while the 400 family processors time base is incremented on each processor clock cycle. The SPRs for accessing the 600 family time base are different numbers than those of the 400 family. Some of the 400 family processors support the use of 600 family time base SPRs in addition to 400 family SPRs.

Development Environment

When considering an embedded processor, the development environment is often overlooked. The development environment consists of software as well a built-in logic for debug and trace.

If you are looking for a high-performance 32-bit RISC processor or family of processors

that support a wide range of price and performance or, if you have designed a product using an IBM PowerPC 400 family processor and want to develop a follow-on product requiring higher performance, you should consider a
PowerPC 600 family processor. With a little forethought, you can use both 400 and 600 family processors to develop products that meet a wide range of price and performance targets while investing in a single set of development tools.

---

2. PPC Programming Model

2.1 PowerPC Architecture Overview

The original PowerPC Architecture defines a family of processors that span a range of price and performance. The architecture specification is defined in four books that define four levels of the architecture. IBM currently offers two families of PowerPC implementations, the 600/700 family which is targeted towards general purpose computer applications, and the 400 family which has specific features which optimize it's use in embedded control applications.

Book 1 defines the User Instruction Set Architecture, to which all Power PC processors conform. Thus, application level programs that only use Book 1 features are portable among implementations without modification. Book 1 also defines floating-point support as well as the option of implementing 64 or 32-bit sized general-purpose registers. Floating-point operations maybe implemented with hardware assistance or, to maintain binary compatibility, as software routines where floating-point instructions cause illegal instruction exceptions.

Book 2 defines the Virtual Environment Architecture, which defines features that permit application programs to share data among programs in a multiprocessing system; and

optimize the performance of storage accesses. This book has been modified by IBM for embedded application oriented processors of the 400 family and is entitled IBM PowerPC Embedded Virtual Environment. The 400 family processors support all storage control instructions and the Time Base functionality defined in the original Virtual Environment Architecture. Future implementations can have support for multi-processor memory coherence.

Book 3 specifies the Operating Environment Architecture, which defines features to permit operating systems to allocate and manage storage to handle exceptions, and to support I/O devices in support of multiprocessor operating systems. Again, changes have been made to enhance embedded usage capability. The IBM PowerPC Embedded Operating Environment is similar to the base architecture but many differences exist. Changes include the addition of a DCR address space, a dual-level interrupt structure with critical and non-critical interrupt sources, additional cache management instructions, extended timer facilities, an MMU better suited for embedded usage, and enhanced debug capabilities. Because these features are available only to "privileged" programs, application code is still compatible regardless of Book 3 variations.

Book 4, the PowerPC Implementation Features defines all implementation specific aspects of the architecture. Any code for this book will likely need to be revised or removed when porting across PowerPC families.

PowerPC Book E combines the architecture specification of the original four architecture books, plus changes and enhancements, to define a standard evolutionary path for future PowerPC implementations that will provide greater code portability. Note: Through out this document, a reference to the 600 family of PowerPC implementations generally also includes the 700 implementations. When a specific distinction between the 600 and 700 families is necessary, it will be clearly indicated.

2.2 Memory Management Unit

The primary function of the Memory Management Unit (MMU) in a PowerPC processor

is the translation of effective (logical, virtual) addresses to physical (real) addresses for instruction and data storage accesses. The secondary function of the MMU is to provide access protection and storage attribute control on a memory sub-region basis. These functions support demand-paged virtual memory. There are many aspects of memory management that are implementation-dependent. The 400 series of processors have MMUs that are very similar to each other, and which are distinct from the 6xx/7xx MMU design.

Operating system code used for processor initialization, exception handling, device drivers, and physical memory management will have significant differences. Application code that uses only the Book 1 architecture features will not be impacted.

2.2.1 600/700 Family MMU Overview

The 600/700 family MMU manages memory by dividing the address range of memory into blocks, segments and pages. Blocks are used for defining large regions of memory to have the same access protection and storage attributes. Memory blocks are created by using instruction or data Block Address Translation (BAT) registers to define memory blocks sized between 128KB and 256MB.

If a memory block is not defined for a given effective address, address translation is handled by the page address translation logic, which supports virtual paging. Virtual paging is done using a software constructed page table and hardware managed TLB. The 4K-byte pages are a fixed in size and define the granularity of memory management when blocks are not used. Effective page addresses are translated to interim virtual addresses using the segment registers, and a page table stored in RAM translates the virtual address to a physical address. Segments are 256MB effective address memory regions that are mapped to virtual segments by using the Segment Registers (SR0-SR15). The virtual segment address defined in the SR is used to access a Page Table Entry (PTE). SR registers are modified using the mtsr, mfsr, mtsrin and mfsrin instructions.

The PTE defines the physical address to be used. The MMU automatically scans the page table for a matching entry each time an execution unit presents an address. When a match is found, the entry is cached in the MMU managed TLB. The 603-processor software is assisted by table search functions, which are controlled by using the table search registers.

The tlbie, tlbia and tlbsync instructions are provided for software TLB management.

2.2.2 400 Family MMU Overview

The 400 family MMU divides the address range of memory into pages when address translation is enabled, and two bounded-regions when address translation is disabled. It has a unified 64-entry fully associative TLB array. Page entries for either instruction or data accesses can be placed anywhere in the TLB. Pages are individually sizeable from 1K byte to 16M bytes. Since each can be sized as needed, fewer TLB entries are needed and the amount of TLB entry swapping is eliminated, or greatly reduced compared to a 600 family implementation. TLB effective address matching is optionally additionally qualified by the TID field having to match the contents of the PID (Process ID) register.

When address translation is enabled (i.e., virtual-mode), the MMU divides the effective address range into pages. Pages are individually controllable regarding address translation and access protection. Eight page sizes are available between 1KB and 16MB. Page size and address translation is controlled by the entries in the TLB array. The TLB is completely managed by software, creating and deleting TLB entries, assisted by three table search instructions; tlbre, tlbwe, tlbsx. If a page table is used, there is no hardware assistance for lookup of page table entries or management of the Referenced and Changed bits. Virtual paging is done using a software constructed page table and software creation of entries in the TLB. Software can quickly check for the existence of, or find existing entries by using the table search instruction, tlbsx. Effective addresses without a matching TLB entry are automatically recorded in the SRR0 register for instructions or in the DEAR for data loads and stores, before the TLB miss exception handler is executed.

Effective page addresses are translated to interim virtual addresses by combining the 32-bit effective address with the 8-bit PID register value to form a 40-bit value which the fully associative array hardware of the TLB will attempt to match with the tag portion of a TLB entry. If a match is found in a TLB entry, the data portion of the TLB entry provides the physical address to be used.

The access protection of multiple pages can be controlled using the Zone Protection Register (ZPR). Each TLB entry can be assigned to one of 16 zones, which then control the protection for all such TLB entries. For the 403 family implementation, with address translation disabled (real-mode) read-only accesses can be defined for two regions of memory by using the PBL1-PBU1 and PBL2-PBU2 register pairs. The region can be either inclusive or exclusive of the address range defined by the register pairs. Protection against writes is enabled using the MSR[PE] (Protection Enable) bit. The PE bit is automatically negated upon the occurrence of any interrupt, allowing the interrupt handler write-access to all of memory. Control of speculative fetching is controlled using the SGR (Storage Guarded) register. This read-only access feature has been removed from the architecture and is not implemented on the 401 or 405 processors.

2.2.3 MMU Differences

The 400 family does not have the segmented memory concept, and therefore, no segment registers or tables as defined in the PowerPC architecture and as provide by the 600 family processors. It also doesn't have hardware assistance for TLB management by looking up page table entries in a page table. Instead, it uses a software managed TLB, therefore virtual-paging software for the 400 family will have to provide TLB entry management as well as the page table management functions. 600 family TLB management code being ported has to be modified to work with the 400 family processor’s unified TLB, and to use the tlbsx instruction.

When porting code from the 400 family, realize that the 600 family does not have the concept of zones, so page table entry creation software must be modified to eliminate any functions related to the TLB entries ZSEL field and the ZPR register. Guarding against speculative memory accesses is controlled by the SGR in the 400 family versus using the WIMG bits of either the BAT register or the page table entry in the 600 family processors.

2.3 Cache Differences

The PowerPC architecture does not impose a specific L1 cache organization. To ensure portability among PowerPC implementations, programmers should assume that the all implementations possess separate instruction and data caches. All IBM 600 and 400 family processor have separate (non-unified) instruction and data caches.

For the 401 and 403 implementations, the iccci instruction invalidates individual cache blocks. For the 405, iccci invalidates the entire instruction cache. Therefore, the portion of cache management code initializing the cache prior to first usage can run unchanged when ported from a 401/403 to a 405.

Although the additional iccci instructions would be redundant, they are harmless. The 600 family processors invalidate the entire instruction cache by asserting the HID0[ICFI] bit. When porting code from 600 family to 400 family, use of the iccci instruction is required as well as removal of references to HID0, which doesn't exist.

Another cache management instruction, icbt (instruction cache block touch), varies in execution privilege among the 400 family and does not exist at all in the 600 family. For 401 and 403 processors it is a privileged instruction that may only be executed when in supervisor mode. For the 405, user-mode software may also use it thereby making it available to applications for performance enhancement.

For the 401 and 440 processors, cache blocks are individually lockable, use of this feature will have to be discontinued if porting to other members of the 400 or 600 families. The 600 family can lock the entire instruction or data cache.

The 400 family provides a capability to read the cache. For the 401 and 403, the CDBCR register is used along with the icread, icdbdr and dcread instructions. The 405 uses the CCR0 register. The 600 family does not have any capability to read caches via code, however for debugging purposes, it can be read through the JTAG port.

2.4 Exception Differences

This section describes the differences in exception processing between the 400 and 600 family of PowerPC implementations. In some cases, the only difference is that the exception vector address varies for the same type of exception causing event. In others, exception vector address locations are used for different purposes, or the causing event may vary slightly with different detailed causal information.

Finally, some exceptions are completely unique to a specific family and/or implementation. The 400 family locates the interrupt vectors using the EVPR register that provides the high-order 16 bits of the exception handler routine addresses. The 600 family has only two possible vector locations, either 0x000n_nnnn or 0xFFFn_nnnn, selected by using the MSR[IP] bit. The restricted location options of the 600 family affects the system memory map, typically requiring location of RAM starting at address 0x0000_0000 and ROM covering the addresses starting at 0xFFF0_0000. This allows exception handlers resident in ROM during hardware initialization and then the subsequent use of RAM based exception handlers. When porting code to the 400 family the 600 family choices for exception vector locations are available via the flexibility of the EVPR.

2.4.1 Exception Vector Offsets With Different Usage

The exception vector offset of at 0x0100 is used for the critical interrupt exception type of the 400 family while it is used as the system reset exception type in a 600 family implementation. The critical interrupt is a 400 family implementation specific extension to the architecture that has no equivalent in the 600 family.

The exception vector located at 0x1000 is used for the programmable interval timer (PIT) function of the 400 family, and is used for instruction translation miss event in the 600 family. Additionally in the 600 family, the exception vector offsets of 0x1010 and 0x1020 are used for the fixed interval timer (FIT) and watchdog timer respectively. In the 400 family, this address space remains part of the 600 family instruction-translation-miss vector.

The exception vector located at 0x1100 is used for the data TLB miss interrupt of the 400 family. The 600 family has separate vectors for load vs. store and for data translation misses. Vector offset 0x1100 is used for load misses and 0x1200 is used for store misses.

The exception vector located at 0x1200 is used for the instruction TLB miss function of the 400 family, but is used for the data-store-translation miss event for the 600 family implementations. When porting code from the 600 family, be aware that the 400 family has a unique specific vector offset of 0x2000 that is used for debug events. This offset is not used at all for exception vectors in the 600 family, therefore this memory address range could have been used for other purposes. Any code or data located here would need to be moved so that at least a word starting at 0x2000 would be available for a branch instruction to the exception handler.

2.4.2 Exception addresses specific to the 400 family

0100 Critical Interrupt Pin Critical Interrupt pin
1000 Programmable Interval Timer Posting of an enabled Programmable Interval Timer interrupt in the Timer Status Register (TSR)
1010 Fixed Interval Timer Posting of an enabled FIT interrupt in the TSR
1020 Watchdog Timer Posting of an enabled first time-out of the watchdog timer in the TSR
1100 Data TLB Miss Valid matching entry for the EA and process ID of an attempted data access is not found in the TLB.
1200 Instruction TLB Miss Attempted execution of an instruction at an address and process ID for which a valid TLB entry does not exist.
2000 Debug Debug Events when in Internal Debug Mode

2.4.3 600 Family Specific Exceptions

The 600 family MSR[RI] bit indicates if the interrupt is recoverable, i.e. if execution of the exception causing instruction can be continued. Before any 600 family exception

handler attempts to return to the non-interrupt code, it should insure the RI bit indicates recoverability. The floating-point not enabled exception is 600 family specific since none of the 400 family have floating point units.

The Instruction Address Breakpoint at offset 0x1300 is analogous to the 400 family debug exception at 0x2000. The interrupt handler routine needs to be changed to work on the 400 family; in part to handle the additional debug events that are possible.

The 600 family System Management interrupt at 0x1400 does not have an equivalent exception in the 400 family.

0100 System Reset A system reset is caused by the assertion of SRESET or HRESET. HRESET causes branch to 0xFFF0_0100
0800 Floating-Point unavailable Attempt to execute a floating point instruction when MSR[FP]=0 (Also available on 405 and 440 with APU)
0900 Decrementer Most significant bit of the decrementer (DEC) register transitions from 0 to 1. MSR[EE] must be set.
0D00 Trace MSR[SE]=1 or when currently completing instruction is a branch and MSR[BE]=1
0E00 Floating Point Assist May be used by some PowerPC implementations for floating-point assist exceptions
1000 Instruction Translation Miss Effective address for an instruction fetch cannot be translated by the ITLB
1100 Data Load Translation miss Effective address for a data load operation cannot be translated by the DTLB
1200 Data Store Translation Miss Effective address for a data store operation cannot be translated by the DTLB or a DTLB hit occurs, and the change bit in the PTE must be set due to a data store operation
1300 Instruction Address Breakpoint Address bits 0-29 in the IABR matches the next instruction in the completion unit and the IABR enable bit (bit 30) is set to 1
1400 System Management Interrupt SMI input signal is asserted and MSR[EE] = 1

2.5 Time Base Differences

The PowerPC architecture defines a 64-bit incrementing register for use as a time base and all implementations have the prescribed 64-bits except the 403GA which uses 56 bits. The frequency of incrementing the time base register is implementation dependent. The 600 family processors time base is incremented at 1/4 the bus clock rate. The 400 family time base is incremented at either the execution unit clock rate or the user provided external clock rate. The 403GCX core rate can be twice that of the system bus clock rate and the time base is incremented at this core clock rate.

The instructions for writing to the time base are not implementation dependent, thus code written to set the time base on a 32-bit PowerPC implementation will work correctly on a 64-bit implementation whether running in 64 or 32-bit mode.

The time base read method is implementation dependent. The 600 family, 401x2 and 405 time base is read using different SPR numbers then those used for writing the 32-bit TBU and TBL registers. User privilege-level access is read-only using the mftb instruction to read TBL and mftbu to read TBU. Supervisor privilege-level programs can write the registers using the mttbl and mttbu to write the TBL and TBU registers, respectively.

Access to the time base registers in the 401M1 core and 403 use 4 time base registers. Instead of the 600 family's TBU and TBL registers, these processors have the functionally equivalent registers named TBHI and TBLO which can be read and written by supervisor privilege-level programs. User privilegelevel programs may only read the time base using the registers TBHU and TBLU. All time base register access is performed using the mtspr and mfspr instructions. The 401M1 core supports use of either the 403 family method or the 600 family method of access.

The Book E architecture does not have a mftb instruction. The 440 core is Book E compliant and so uses the 403 method of access.

2.6 Supported Endian Modes

The PowerPC architecture specifies big-endian as the normal memory access order. The 401 also supports true little-endian and PowerPC little-endian memory access ordering. The 403 and 600 family add only PowerPC little-endian. The 405 supports big-endian and true little-endian operation.

2.7 Floating-Point Support

The 600 family has a floating-point unit to perform floating-point operations with hardware assistance. The 400 family handles floating-point instructions by using the integer unit to provide emulation via program functions that are invoked when a floating-point instruction is encountered. The functions are invoked by having an illegal instruction exception handler that recognizes the opcode and invokes the corresponding floating-point routine. A higher performance approach is to directly call subroutines of a floating-point emulation run-time library, but binary program compatibility is sacrificed.

2.8 APU Support

The 401 and 405 cores have support for an Auxiliary Processing Unit (APU) which is tightly coupled to the processors execution unit. It can be used, for example, to add a floating-point operation unit to speed floating-point math operations. Or a Multiply with Accumulate (MAC) unit for DSP type instructions.

The 401 cores do not have the ability to load or store data to the APU. The 405 core provides load and store operations using specific instructions for moving opcodes and parameter to the APU. These are defined as extensions to the PowerPC architecture defined instruction set.

2.9 Debug Registers

Debug facilities are implementation dependent. In general, the 400 family implementations all provide the ability to cause a debug event if either a instruction or data address of interest is accessed. The 600 family provides only an instruction address breakpoint controlled by using the IABR. The 401 provides a single instruction address

and a single data address debug event generation ability. The are controlled using the DBCR and DBSR registers.

The 403 provides two instruction address and two data address debug event abilities, again controlled using the DBCR register and getting status via the DBSR register. However there is also a "first event" counter and event sequencing ability.

The 405 has four instruction address registers which can be used individually, or to define an inclusive or exclusive address range. It has two data address registers that can also provide discrete address matching or a range of data movement address matching capability. Additionally two registers are provided to further qualify the generation of a debug event based on the value of the data being loaded or stored. The debug functionality is controlled using the DBCR0 and DBCR1 registers. Status is indicated in the DBSR register.

2.10 Power Management

The 400 and 600 families implement somewhat similar power management functionality, but they use different control mechanisms.

The 400 family uses the MSR[WE] Wait-state Enable bit to control place the processor in the wait state. When an exception is taken, the Wait State is removed.

The 600 family uses a the MSR[POW] bit to control the enabling and disabling of power management. Exceptions automatically disable power management when the interrupt is taken. Its original state is saved in SRR1.

The 700 family provides for programmable power states; full power, doze, nap and sleep. Software selects these modes by setting one (and only one) of the three power saving mode bits in the HID0 register. A hardware interrupt causes the transfer of program flow to an interrupt handler that then invokes the appropriate power saving mode. There is also

the ability to use a decrementer register to enter the nap or doze mode for a predetermined amount of time and then return to full power mode using a decrementer interrupt.

2.11 Miscellaneous Processor Specific Registers

- PVR - The PVR value is unique for each processor implementation

2.11.1 600 Family Specific Registers

- SPRG0-SPRG3 - Only the 600 family has these general-purpose SPR registers, which the program can use as needed. Conventionally, for the 600 family processors SPRG0 is reserved as register containing the value for a stack pointer to be used by the first-level exception handler. SPRG1 is then used in first-level exception handlers to save a GPR, which is then used as a stack pointer for saving other registers by copying the content of SPRG0 to the saved GPR. The EABI specifies that GPR1 is used as the stack pointer and is always valid, so saving it using by means of an SPRG, or any other method, is not necessary when running EABI compliant programs.
- HID0 – Hardware Implementation Dependent register 0 is used for enabling and determining checkstop sources, bus parity control, cache control and configuration, and power management. Its functionality varies with each 600 family implementation.
- HID1 – Is used to configure the PLL and is accessed with mtspr and mfspr. Its functionality varies with each 600 family implementation.
- EAR – The External Access Register is an optional SPR in the PowerPC Operating Environment Architecture (OEA) that controls access to the external control facility. It identifies the target device for the external control facility, which communicates with special external devices. The 400 family has no equivalent facility.
- PIR – The Processor ID Register is optional register in the OEA that can be used by the operating system to assign an ID to a processor. It is also used when the processor communicates with I/O devices.

2.12 Summary

The PowerPC Architecture provides a high degree of code portability between different

implementations even when the implementations have significant differences in their intended applications. Porting code between implementations is a straightforward and manageable job that is primarily limited to the kerneland other supervisory mode code. Applications typically require almost no changes in order to be moved to a different target processor than that which they were originally developed for. Porting between processors can be greatly facilitated by the use of structured programming methods, which then minimizes the amount of code needing modification. For example, if all access to the Time Base is done using a single function which is used by the entire application and operating system, only that function needs to be modified when porting to another PowerPC processor.

3. PowerPC EABI

3.1. Overview

An Application Binary Interface (ABI) specifies an interface for compiled application programs to system software. The Embedded Application Binary Interface (EABI) is derived from the PowerPC ABI Supplement to the UNIX System V ABI. The PowerPC ABI Supplement was designed for workstation operating systems such as IBM's AIX and Apple's Mac OS. The EABI differs from the PowerPC ABI Supplement with the goal of reducing memory usage and optimizing execution speed, as these are prime requirements of embedded system software. The EABI describes conventions for register usage, parameter passing, stack organization, small data areas, object file, and exectable file formats. This application note covers the following EABI topics:

- Development tool file formats
- Data types and alignment
- Register usage conventions
- Stack frame creation and organization
- Function parameter passing

- Small data area usage and organization

3.2. File Formats

Object and executable files complying with the EABI are in the Extended Linking Format (ELF) and debug information is contained in Debug with Arbitrary Record Format (DWARF) records. The current revision of the DWARF standard is 1.1.0. There is a proposed revision of the standard currently known as DWARF 2.0.0 which primarily adds features to support C++ code debugging. Although not yet an official standard, many tool providers have implementations of, or close to, the current proposed DWARF 2.0.0 standard.

3.3. Data Types and Alignment

The PowerPC architecture defines scalar (integer) data type sizes as shown below table 1.

Table 1 - PowerPC Data type Size (bytes)

Data type	Size (bytes)
Byte	1
Halfword	2
Word	4
Doubleword	8
Quadword	16

All data types are aligned in memory, and in the stack frame, on addresses that are a multiple of their size. For example, a word, since its size is 4 bytes, is aligned on an address evenly divisible by 4. The address of a halfword is evenly divisible by 2. An exception to this rule is quad-words when they are not contained in a union or structure; they only require alignment to eight byte boundaries. Arrays are aligned on the boundary required by the size of the data type of the array elements. A structure (or union) is aligned based on the alignment requirements of the structures largest member. Thus, if the structure contains a doubleword, the doubleword member must begin on an address evenly divisible by 8. Padding of prior and subsequent members is done as needed to

maintain their individual alignment requirements. The size of a structure is always a multiple of the structure's alignment. If necessary a structure is padded after the last member to increase its size to be a multiple of its alignment. EABI compliant compilers and assemblers will automatically create correctly aligned data allocations but the padding required may cause problems for some applications. An example would be a networking- protocol data packet, which has specific alignment requirements. For these situations some compilers allow the alignment feature to be turned off, or overridden with different boundary values. For example, the IBM HighC/C++ compiler has a #pack pragma for this purpose. Since non-aligned data access requires multiple bus cycles for reads and writes, and perhaps even software assistance through an exception handler, performance will be decreased. Non-aligned data access should be avoided if at all possible.

Table 2 shows the ANSI C language data types and their sizes. For all types, NULL is defined as the value zero. Signed and unsigned integer types have the same size in all cases.

Table 2 - PowerPC ANSI C data types

ANSI C data type	PowerPC Data type	Size (bytes)
char	byte	1
short	halfword	2
int	word	4
long int	word	4
enum	word	4
pointer	word	4
float	word	4
double	doubleword	8
long double	quadword	16

3.4. Register Usage Conventions

The PowerPC architecture defines 32 general purpose registers (GPRs) and 32 floating-point registers (FPRs). The EABI classifies registers as volatile, nonvolatile, and dedicated. Nonvolatile registers must have their original values preserved, therefore, functions modifying nonvolatile registers must restore the original values before returning to the calling function. Volatile registers do not have to be preserved across function calls.

Three nonvolatile GPR's are dedicated for a specific usage, R1, R2, and R13. R1 is dedicated as the stack frame pointer (SP). R2 is dedicated for use as a base pointer (anchor) for the read-only small data area. R13 is dedicated for use as an anchor for addressing the read-write small data area. Dedicated registers should never be used for any other purpose, not even temporarily, because they may be
needed by an exception handler at any time. All the PowerPC registers and their usage are described in Table 3.

Table 3 - PowerPC EABI register usage

Register	Type	Used for:
R0	Volatile	Language Specific
R1	Dedicated	Stack Pointer (SP)
R2	Dedicated	Read-only small data area anchor
R3 - R4	Volatile	Parameter passing / return values
R5 - R10	Volatile	Parameter passing
R11 - R12	Volatile	
R13	Dedicated	Read-write small data area anchor
R14 - R31	Nonvolatile	
F0	Volatile	Language specific
F1	Volatile	Parameter passing / return values
F2 - F8	Volatile	Parameter passing
F9 - F13	Volatile	
F14 - F31	Nonvolatile	
Fields CR2 - CR4	Nonvolatile	
Other CR fields	Volatile	
Other registers	Volatile	

3.5. Stack Frame Conventions

The PowerPC architecture does not have a push/pop instruction for implementing a stack. The EABI conventions of stack frame creation and usage are defined to support parameter passing, nonvolatile register preservation, local variables, and code debugging. They do this by placing the various data into
the stack frame in a consistent manner. Each function which either calls another function or modifies a nonvolatile register must create a stack frame from memory set aside for use as the run-time stack. If a function is a leaf function (meaning it calls no other functions), and does not modify any nonvolatile
registers, it does not need to create a stack frame. The SP always points to the lowest addressed word of the currently executing functions stack frame.
Each new frame is created adjacent to the most recently allocated frame in the next available lower addressed memory. The stack frame is created by a function's prologue code and destroyed in its epilogue code. Stack frame creation is done by decrementing the SP just once, in the function prologue, by the total amount of space required by that function. To insure the SP update is an atomic operation that cannot be interrupted, a store-with-update (stwu) instruction is used. The prologue will also save any nonvolatile registers the function uses into the stack frame. Below is an example function prologue.

```
FuncX: mflr %r0 ; Get Link register
stwu %r1,-88(%r1) ; Save Back chain and move SP
stw %r0,+92(%r1) ; Save Link register
stmw %r28,+72(%r1) ; Save 4 non-volatiles r28-r31
```

The stack frame is removed in the function's epilogue by adding the current stack frame size to the SP before the function returns to the calling function. The epilogue code of a function restores all registers saved by the prologue, de-allocates the current stack frame by incrementing the SP, then returns to the
calling function. The following function epilogue example corresponds to the above prologue.

```
lwz %r0,+92(%r1) ; Get saved Link register
```

mtlr %r0 ; Restore Link register

lmw %r28,+72(%r1) ; Restore non-volatiles

addi %r1,%r1,88 ; Remove frame from stack

blr ; Return to calling function

Figure 1 illustrates the stack frame concept by using a 2-level deep, function calling example. At time 1, function A exists and calls function B. At 2, B's prologue code has executed and created B's frame. At 3, B has called C and C's prologue code has executed. At 4, function C has terminated and C's epilogue

code has destroyed C's frame by incrementing the value in the SP.

Figure 1 - Stack Frame creation and destruction

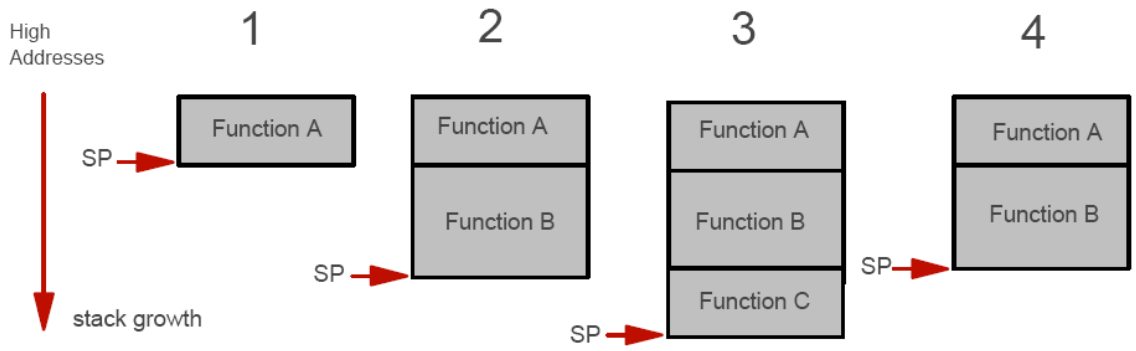


Figure 1 - Stack Frame creation and destruction

The stack frame is always doubleword aligned (on 8 byte boundaries) by using padding, if necessary.

Figure 2 shows the stack frame organization including all optional areas.

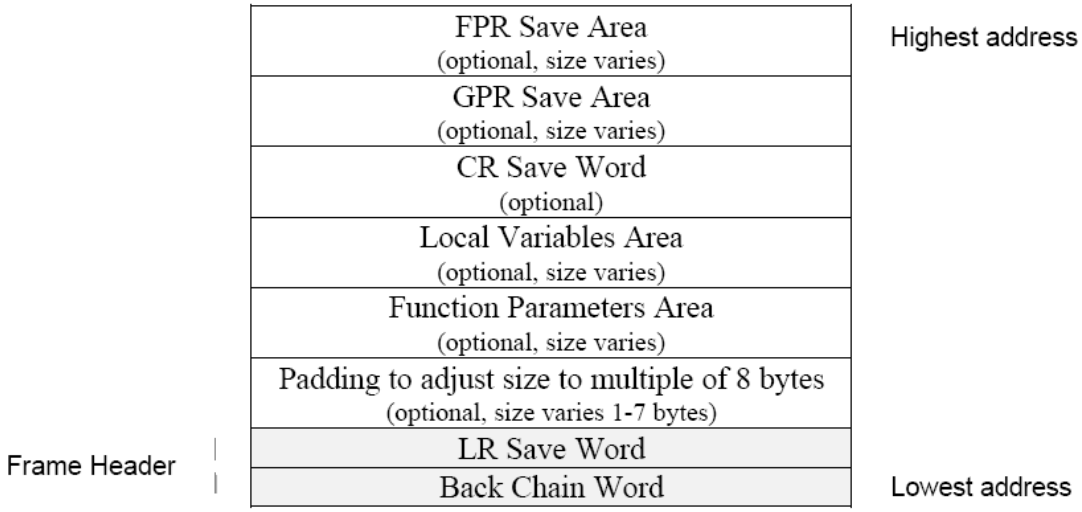


Figure 2 - EABI Stack Frame

All stack frames have a header consisting of two fields - the Back Chain Word and the Link Register (LR) Save Word. The Back Chain Word contains the address of the previous frames Back Chain Word field, thereby forming a linked-list of stack frames. The Back Chain Word is always located at the lowest address of the stack frame. The LR Save Word is used by functions to store the current value of the Link Register prior to modifying it. The value in the LR upon entry into a subroutine represents the return address to the calling function. It is located in the word immediately above the Back Chain Word field. The Function Parameter Area is optional and varies in size. It contains additional function arguments when there are to many to fit into the designated registers R3-R10. It is located just above the LR Save Word in the callers stack frame. The Local Variables Area is for functions local variables when there are more than can be contained in the available volatile registers. If a function modifies any of the nonvolatile Condition Register (CR) fields it must save the entire CR in the CR Save Area. The General Purpose Register (GPR) Save Area is optional and varies in size. When saving any GPR, all the GPRs from the lowest to be saved up through R31, inclusive, are saved. For example, if a function is modifying R17, it must create a stack frame large enough to contain R17 through R31 in its GPR Save Area. The same conventions apply for the Floating point Register Save Area for saving the FPR nonvolatile registers. Code for implementations of the PowerPC that do not have floating-point hardware would not

create the FP Save Area as there are no FPRs to save.

3.6. Parameter Passing

For PowerPC processors it is more efficient to pass arguments in registers than using memory. Up to eight scalar values are passed by using R3 through R10 and return values are passed back in R3 and R4. Up to eight arguments of the floating point data type can be passed using F1 to F8 and F1 is used to return a floating-point value. If there are more arguments than can be passed using the registers, space for the additional arguments is allocated for them in the stack frame's Function Parameters Area. Likewise, returned values that will not fit into R3 and R4 (or F1) are also passed by using the Function Parameters Area. The following C code fragment illustrates the concept.

```
#include "stdio.h"
void func1(int);
int var1;
main(){
var1 = 4;
func1(var1);
}
void func1(int arg1){
printf("func1 - arg1 value: %d\n",arg1);
}
```

To implement the C language statements the following assembly language instructions illustrate loading and passing the value in var1 to func1. After var1 is set to 4, R3 is loaded with the value in var1 in order to pass it as an argument. The lwz instruction is used to load R3. Notice that after the instructions to set var1 = 4, R12 contains the high order 16-bits of the address of var1 and is therefore used by the lwz instruction. R3 is used since it is the first available parameter passing register for integer values.

```
var1 = 4;
li %r11,4
addis %r12,%r0,var1@ha
stw %r11,var1@l(%r12)
```

```
func1(var1);
lwz %r3,var1@l(%r12)
bl func1
```

3.7. Small Data Areas

The EABI has a construct known as the Small Data Area (SDA) designed to take advantage of the PowerPC base plus displacement addressing mode. The displacement is a signed 16-bit value, therefore a total of 64k (plus or minus a 32K offset) bytes may be addressed without changing the value in a base register. The 16-bit displacement fits, along with the instruction op-code, into a single instruction word. This fact means it is a more memory efficient method of accessing a variable than referencing it by using a full 32-bit address. That's because only one instruction word is required instead of the two needed to access it as a 32-bit address. SDAs are useful for global and static variables and constants.

There are two SDAs, one for read-write variables and a second for read-only variables. The small data areas are referenced by a base register loaded once, when the C runtime environment is initialized. R2 is the base for the read-only (const type) small data area, and R13 is the base for the read-write (nonconst
type) small data area.
Variables for the R13 based read-write SDA are contained in one of two ELF segments, either .sdata, or .sbss. For initialized read-write variables, .sdata is used and .sbss is used for non-initialized read-write variables. Typically the .sbss variables are given a default initial value of 0 at run-time. Since this SDA is read-write it must be located in RAM.
Here is an example instruction to fetch the value of a read-write small data area variable. It is located at an offset of 32 bytes greater than the anchor value:

```
lwz r29,32(r13)
```

Variables for the R2 based read-only SDA are contained in one of the segments .sdata2 and .sbss2. For initialized read-only variables, .sdata2 is used. For non-initialized variables, .sbss2 is used. Typically, non-initialized variables are given a default initial

value of 0 at run-time. Since the SDA is read-only, it may be located in ROM as long as initialization of the .sbss2 segment contained variables is not required.

The PowerPC architecture treats R0 as a value of zero (not the content of R0) when used as the base register for the base + displacement addressing mode for some instructions. These instructions include the load, store, and various cache management instructions. Therefore, R0 acts as an anchor for a third, implicit, small data area which includes the lowest and highest 32k bytes of the processor memory address space.

With the IBM HighC/C++ compiler, placement of variables into SDAs is enabled using the pragma Push_small_data. By default, global variables are not placed into an SDA. The pragma can be invoked by using a compiler option. The option "-Hpragma=Push_small_data(4;0)" instructs that read-write variables that are 4 bytes or less in size should be stored in the read-write SDA. It also instructs that no read-only variables are to be placed into the read-only SDA by specifying the value 0, for the second argument. The following C program fragment will help illustrate the machine instructions used to access a global variable.

```
int var1;
main()
{
var1 = 4;
func1(var1);
}
```

Below are the assembly language instructions generated by the compiler for writing the value 4 to the global variable var1, when it is not in an SDA. Three instructions are required to store a value into var1.

```
li %r11,4
addis %r12,%r0,var1@ha
stw %r11,var1@l(%r12)
```

1. li gets the value to be set into register R11.

2. addis is used to load R12 with the high-order halfword of the address of var1.

3. stw uses the base + displacement addressing mode. The displacement, from the base address in R12, is the low halfword of var1's address.

When in the read-write SDA, the resulting two instructions for writing a value to var1 are:

```
li %r12,4
stw %r12,var1@sdaxr(%r13)
```

Note that to use any SDAs, you need the C runtime environment creation code to initialize the small data area anchor registers. For the IBM evaluation kit user, you can do this by adding the following code to the ./samples/bootllib.s routine right before the jump to the _kernel_entry routine. The macros _SDA_BASE_ and _SDA2_BASE_ are defined automatically by the linker if the associated SDAs are used. For _SDA_BASE_, the value is the address to which all data in the .sdata and .sbss sections can be addressed using a 16-bit signed offset. If an SDA is not used the associated macro's value will be zero.

```
!************************************************************
! INITIALIZATION OF BASE REGISTERS FOR SMALL DATA AREAS:
!************************************************************
lis %r2,_SDA2_BASE_@ha ! r2 is the read-only SDA anchor
addi %r2,%r2,_SDA2_BASE_@l
lis %r13,_SDA_BASE_@ha ! r13 is the read-write SDA anchor
addi %r13,%r13,_SDA_BASE_@l
```

3.8. Summary

The EABI provides for vendor independent tool interoperability via the ELF/DWARF file format standards. This allow developers to mix and match various EABI compliant components to create a software development tool chain for their needs. In addition, the EABI standards on register usage and parameter passing also allow independently developed code to be reused without modification.

4. PowerPC 4xx Interrupts and Exceptions

This chapter discusses in detail the following topics:

- Interrupt Classes
- Interrupt Processing
- Interrupt and Exception Types
- Partially Executed Instructions
- Interrupt Ordering and Masking
- Exception Priorities
- Exception Handling Registers
- Interrupt Definitions
- Interrupt Control Instructions

4.1 Overview

An Interrupt is the action in which the processor saves its old context (Machine State Register (MSR) and instruction pointer) and begins execution at a pre-determined interrupt-handler address, with a modified MSR. Exceptions are the events which will, if enabled, cause the processor to take an interrupt. In the IBM PowerPC Embedded Environment, exceptions are generated by signals from internal and external peripherals, instructions, the internal timer facility, debug events, or error conditions.

All interrupts, except Machine Check, are ordered within the two categories of non-critical and critical, such that only one interrupt of each category is reported, and when it is processed (taken) no program state is lost. Since Save/Restore Register pairs SRR0/SRR1 and SRR2/SRR3 are serially reusable resources used by all non-critical and critical interrupts respectively, program state may be lost when an unordered interrupt is taken. All interrupts, except Machine Check, are context synchronizing

4.2 Interrupt Classes

All interrupts, except for Machine Check, can be categorized according to three independent characteristics of the interrupt:

1. Synchronous/Asynchronous Interrupts

2. Precise/Imprecise Interrupts

3. Critical/Non-Critical Interrupts

4.2.1 Synchronous/Asynchronous Interrupts

Synchronous interrupts are those which are caused directly by the execution (or attempted execution) of instructions. Synchronous interrupts may be either precise or imprecise. Asynchronous interrupts are caused by events which are independent of instruction execution. All asynchronous interrupts are precise.

4.2.2 Precise/Imprecise Interrupts

Precise interrupts are those which precisely indicate the address of the instruction causing the exception which generated the precise, synchronous interrupt; or, for certain precise synchronous interrupts, the address of the immediately following instruction. For asynchronous precise interrupts the address reported to the exception handling routine is the address of the next instruction that would have been executed, had the interrupt not occurred.

4.2.2.1 Precise Interrupts

For synchronous precise interrupts, the following rules apply:

1. The save/restore register addresses either the instruction causing the exception or the immediately following instruction. Which instruction is addressed can be determined from the interrupt type and status bits.

2. An interrupt is generated such that all instructions preceding the instruction causing the exception appear to have completed with respect to the executing processor. However, some storage accesses generated by these preceding instructions may not have been performed with respect to all other processors and mechanisms.

3.The instruction causing the exception may appear not to have begun execution (except for causing the exception), may have partially completed, or may have completed, depending on the interrupt type 4.Architecturally, no subsequent instruction has begun

execution.

For asynchronous precise interrupts, the following rules apply:
1. All instructions prior to the one whose address is reported to the exception handling routine (in the save/restore register) have completed execution with respect to the interrupting processor. However, some storage accesses generated by these preceding instructions may not have been performed with respect to all other processors and mechanisms.
2. No instruction subsequent to one whose address is reported to the exception handling routine has begun execution.
3. The instruction whose address is reported to the exception handling routine may appear not to have begun execution, or may have partially completed.

4.2.2.2 Imprecise Interrupts

All Imprecise interrupts are synchronous and follow the rules outlined below:
1. The save/restore register addresses either the instruction causing the exception or some instruction following the instruction causing the exception that generated the interrupt.
2. An interrupt is generated such that all instructions preceding the instruction addressed by the save/restore register appear to have completed with respect to the executing processor.
3. If the imprecise interrupt is forced by the context synchronizing mechanism, due to an instruction that causes another exception which generates an interrupt (e.g., Alignment, Data Storage), then the save/restore register addresses the interrupt-forcing instruction, and the interrupt-forcing instruction may have been partially executed.
4. If the imprecise interrupt is forced by the execution synchronizing mechanism, due to executing an execution synchronizing instruction other than sync or isync, then the save/restore register addresses the interrupt-forcing instruction, and the interrupt-forcing instruction appears not to have begun execution (except for its forcing the imprecise interrupt). If the imprecise interrupt is forced by a sync or isync instruction, then the save/restore register may address either the sync or isync instruction or the following instruction.

5. If the imprecise interrupt is not forced by either the context synchronizing mechanism or the execution synchronizing mechanism, then the instruction addressed by the save/restore register may have been partially executed
6. No instruction following the instruction addressed by the save/restore register appears to have begun execution.

4.2.3 Critical/Non-Critical Interrupts

Critical interrupts are those which use Save/Restore Register pair SRR2/SRR3. Non-Critical interrupts use Save/Restore Register pair SRR0/SRR1.

4.2.4 Machine Check Interrupts

Machine Check interrupts are a special case. They are typically caused by some kind of hardware or storage subsystem failure, or by an attempt to access an invalid address. A Machine Check may be caused indirectly by the execution of an instruction, but not be recognized and/or reported until long after the processor has executed past the instruction which caused the Machine Check. As such, Machine Check interrupts cannot properly be thought of as synchronous or asynchronous, nor as precise or imprecise. They are handled
as critical class interrupts however. In the case of Machine Check, the following general rules apply:
1. No instruction after the one whose address is reported to the machine check handler software in the save/restore register has begun execution.
2. The instruction whose address is reported to the machine check handler software in the save/restore register, and all prior instructions, may or may not have completed successfully. All those instructions which are ever going to complete appear to have done so already, and have done so within the context existing prior to the machine check interrupt. No further interrupt (other than possible additional Machine Checks) will occur as a result of those instructions.

4.3 Interrupt Processing

Associated with each kind of interrupt is an Interrupt Vector, which is the address of the

initial instruction that is executed when the corresponding interrupt occurs.
Interrupt processing consists of saving a small part of the processor's state in certain registers, identifying the cause of the interrupt in another register, and continuing execution at the corresponding interrupt vector location. When an exception exists that will cause an interrupt to be generated and it has been determined that the interrupt can be taken, the following actions are performed in order:
1. SRR0 (for non-critical class interrupts) or SRR2 (for critical class interrupts) is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.
2. The ESR is loaded with information specific to the exception type. Note that many interrupt types can only be caused by a single type of exception event, and thus do not need nor use an ESR setting to indicate to the interrupt handling routine what the cause of the interrupt was.
3. SRR1 (for non-critical class interrupts) or SRR3 (for critical class interrupts) is loaded with a copy of the MSR..Interrupts and Exceptions
4. The MSR is modified in the following fashion:
MSR[ILE] is copied into MSR[LE], leaving MSR[ILE] unmodified.
MSR[APE, APA, WE, EE, PR, FPA, FE0, FE1, IR, DR] are cleared by all interrupts.
MSR[CE, DE] are cleared by critical class interrupts and left unchanged by non-critical class interrupts.
MSR[ME] is cleared by machine check interrupts and left unchanged by all other interrupts. Other defined MSR bits are left unchanged by all interrupts.
5. Instruction fetching and execution resumes, using the new MSR value, at a location specific to the interrupt type. The location is determined by concatenating the interrupt vector's offset to the right of the high-order 16 bits of the Exception Vector Prefix Register (EVPR). The contents of the EVPR are indeterminate upon reset, and must be initialized by system software via the mtspr instruction. Interrupts do not clear reservations obtained with lwarx. The operating system should do so at appropriate points, such as at process switch.
At the end of a non-critical interrupt handling routine, execution of a return from interrupt (rfi) instruction causes the contents of SRR0 and SRR1 to be restored to the

program counter and the MSR, respectively. Execution then resumes at the address in the program counter. Likewise, execution of a return from critical interrupt (rfci) instruction performs the same function at the end of a critical interrupt handling routine, using SRR2 and SRR3.

Programming Note
In general, at process switch, due to possible process interlocks and possible data availability requirements, the operating system needs to consider executing the following:
stwcx., to clear the reservation if one is outstanding, to ensure that a lwarx in the “old” process is not paired with a stwcx. in the “new” process.

sync, to ensure that all storage operations of an interrupted process are complete with respect to other processors before that process begins executing on another processor.

isync or rfi/rfci, to ensure that the instructions in the “new” process execute in the “new” context.

5. PowerPC Reset and Initialization

—Case Study: PPC 4xx

This chapter discusses the following topics:

- Reset and Initialization
- Reset Mechanisms
- Processor State After Reset
- Initialization Code Example

5.1 Reset and Initialization

This chapter describes the requirements for processor reset. This includes both the means of causing reset, and the specific initialization that is required to be performed automatically by the processor hardware. This chapter also provides an overview of the operations that should be performed by initialization software, in order to fully initialize the processor. In general, the specific actions taken by a processor upon reset are implementation dependent, and are described in the User's Manual for the implementation. Also, it is the responsibility of system initialization software to initialize the majority of processor and system resources after reset. Implementations are required to provide a minimum processor initialization such that this system software may be fetched and executed, hereby accomplishing the rest of system initialization.

5.2 Reset Mechanisms

This specification defines two processor mechanisms for internally invoking a reset operation. In addition, implementations will typically provide additional means for invoking a reset operation, via an external mechanism such as a signal pin which when activated will cause the processor to reset. The two internal mechanisms for invoking a reset operation are:

1. Debug Control Register (DBCR)
2.Timer Control Register (TCR).

5.2.1 Debug Control Register (DBCR)

The DBCR[RST] field may be written by software to a non-zero value in order to cause an immediate processor reset. The exact behavior which results from specific non-zero values written to this field is implementation-dependent. Writing a value of zero to this field will have no effect on the processor.

The Most Recent Reset field of the Debug Status Register (DBSR[MRR]) records information about the most recent reset operation which occurred, regardless of the mechanism which invoked the reset. See the User's Manual for the implementation for a definition of this field.

5.2.2 Timer Control Register (TCR)

The Watchdog Reset Control field of the Timer Control Register (TCR[WRC]) may be set to a non-zero value by software, in order to allow a Watchdog timeout event to automatically invoke a processor reset. The exact behavior which results from specific non-zero values written to this field is implementation dependent.

Once set to a non-zero value, this field may not be restored to a zero value by software, and will only be restored upon an actual reset operation. This is to prevent errant software from disabling the Watchdog reset safeguard, once it has been established by software.

The Watchdog Reset Status field of the Timer Status Register (TSR[WRS]) records information about only those reset operations which were actually invoked by a Watchdog timeout. See the User's Manual for the implementation for a definition of this field.

5.3 Processor State After Reset

The initial processor state is controlled by the register contents after reset. In general, the contents of most registers are undefined after reset.
The processor hardware only initializes those registers (or specific bits in registers) which must be initialized in order for software to be able to reliably perform the rest of system initialization.

5.4 Software Initialization Requirements

When reset occurs, the processor is initialized to a minimum configuration to start executing initialization code. Initialization code is necessary to complete the processor and system configuration. The initialization code described in this section is the minimum recommended for configuring the processor to run application code.
Initialization code should configure the following processor resources:

• Initialize DCWR to non-write-thru, to avoid potential alignment interrupts on store opera-tions (must be done before first store).

• Invalidate the i-cache and d-cache (implementation dependent).

• Enable cacheability for appropriate memory regions.

• Turn off guarded attribute (SGR) as appropriate for memory regions (to enable pre-fetch-ing for improved performance).

• Initialize system memory as required by the operating system or application code.

• Initialize processor registers as needed by the system.

• Initialize off-chip system facilities.

• Dispatch the operating system or application code.

5.4.1 Initialization Code Example

This section presents an example of initialization code to illustrate the steps that should be taken to initialize the processor before the operating system or user programs are executed. It is presented in pseudo-code with function calls similar to PowerPC instruction mnemonics. Specific implementations may require different ordering of these sections to ensure proper operation..Reset and Initialization

```
/* ———————————————————————— */
/* Initialization Pseudo Code */
/* ———————————————————————— */
@0xFFFFFFFC: /* Initial instruction fetch from 0xFFFFFFFC */
ba(init_code); /* branch from initial fetch address to init_code */
@init_code: /* Start of initialization psuedo code */
/* ———————————————————————— */
/* Clear DCWR to avoid potential alignment exceptions on stores. */
/* ———————————————————————— */
mtspr(DCWR, 0);
/* ———————————————————————— */
/* Invalidate both caches */
/* ———————————————————————— */
/* Implementation dependent code sequence to invalidate all lines in instruction cache*/
/* and data cache. */
```

```
/* ———————————————————————————————— */
/* Enable cacheability for appropriate regions of real storage */
/* ———————————————————————————————— */
mtspr(DCCR, d_cache_cacheability);/* enable data cacheability */
mtspr(ICCR, i_cache_cacheability); /* enable instruction cacheability */
/* ———————————————————————————————— */
/* Clear guarded attribute to allow improved performance via instruction prefetch, as */
/* appropriate. */
/* Can do as early as desired (e.g., before caches invalidated). */
/* ———————————————————————————————— */
mtspr(SGR, guarded configuration); /* mark appropriate regions as unguarded

/*————————————————————————————————*/
/* Load operating system and/or application code, including exception handlers, */
/* into memory. */
/* */
/* The example assumes that the system and/or application code is loaded */
/* immediately after the cache is initialized. */
/* The example assumes that the source and destination regions do not overlap and */
/* are aligned on cache line boundaries, and that cache lines consist of four words */
/* (16 bytes) */
/*————————————————————————————————*/
while (not_done) /* repeat until all code has been loaded. */
{
lmw(4, &code); /* load 4 words into 4 registers */
stmw(4, &new_location); /* store 4 words to d-cache */
dcbst(&new_location); /* store cache block to physical memory */
sync(); /* allow store to complete */
icbi(&new_location); /* clear any obsolete code from i-cache */
inc(&code); /* increment the code address by 4 words */
```

```
inc(&new_location); /* increment the new_location addr by 4 words */
}
isync(); /* discard prefetched instructions and re-fetch */
/*————————————————————————————— */
/* set the exception vector prefix */
/*————————————————————————————— */
mtspr(EVPR, prefix_addr); /* initialize exception vector prefix */
/*————————————————————————————— */
/* initialize and configure timer facilities */
/*————————————————————————————— */
mtspr(TBL, 0); /* reset time base lower first to avoid ripple */
mtspr(PIT,0); /* clear PIT so no PIT indication after TSR */
cleared */
mtspr(TSR, 0xFFFFFFFF); /* clear Timer Status Register */
mtspr(TCR, timer_enable); /* enable desired timers */
mtspr(TBU, time_base_u); /* set time base, upper first to catch possible */
ripple */
mtspr(TBL, time_base_l); /* set time base, lower */
mtspr(PIT, pit_count); /* set desired pit count */.Reset and Initialization 8-7
/* ————————————————————————————— */
/* Enable interrupts in the Machine State Register */
/* */
/* Interrupts should be enabled immediately after timer */
/* facilities to avoid missing a timer exception. */
/* */
/* The MSR also controls the user/supervisor mode, */
/* translation, and the wait state. */
/* These modes must be initialized by the operating */
/* system or application code. */
/* If enabling translation, code must initialize the TLB. */
/* */
```

```
/* ———————————————————————————— */
mtmsr(machine_state);
/* ———————————————————————————— */
/* initialization of non-processor facilities should be performed at this time */
/* ———————————————————————————— */
/* ———————————————————————————— */
/* branch to operating system or application code */
/* ———————————————————————————— */
ba(&code_load_address);
```

6. Synchronization Requirements

This chapter discusses in detail the following topics:

- Context Synchronization
- Execution Synchronization
- Synchronization Requirements

6.1 Context Synchronization

An instruction or event is context synchronizing if it satisfies the requirements listed below.

Such instructions and events are collectively called context synchronizing operations. Examples of context synchronizing operations include the sc instruction, the rfi/rfci instructions, and most interrupts.

1. The operation causes instruction dispatching (the issuance of instructions by the instruction fetch mechanism to any instruction execution mechanism) to be halted.
2. The operation is not initiated or, in the case of isync, is not completed, until all instructions already in execution have completed to a point at which they have reported all exceptions they will cause.

3. The instructions that precede the operation will complete execution in the context (privilege, relocation, storage protection, etc.) in which they were initiated.
4. If the operation directly causes an interrupt (e.g., sc directly causes a System Call interrupt) or is an interrupt, the operation is not initiated until no exception exists having higher priority than the exception associated with the interrupt.
5. The instructions that follow the operation will be fetched and executed in the context established by the operation. (This requirement dictates that any prefetched instructions be discarded, which in turn requires that any effects and side effects of speculatively executing them also be discarded. The only side effects of these instructions that are permitted to survive are those related to Out of Order operations.

A context synchronizing operation is necessarily execution synchronizing; Unlike the sync instruction, a context synchronizing operation need not wait for storage-related operations to complete on other processors.

6.2 Execution Synchronization

An instruction is execution synchronizing if all previously initiated instructions appear to have completed before the instruction is initiated or, in the case of sync and isync, before the instruction completes. Examples of execution synchronizing instructions are sync (see the PowerPC User Instruction Set Architecture and the IBM PowerPC Embedded Virtual Environment Section 2.7.1.2, “Synchronize (sync),” on page 2-14) and mtmsr. Also, all context synchronizing instructions (see Section 11.1, “Context Synchronization,” on page 11-1) are execution synchronizing. Unlike a context synchronizing operation, an execution synchronizing instruction need not ensure that the instructions following that instruction will execute in the context established by that instruction. This new context becomes effective sometime after the execution synchronizing instruction completes and before or at a subsequent context synchronizing operation.

6.3 Synchronization Requirements

This section discusses synchronization requirements for special registers and translation lookaside buffers. Changing the value in certain system registers, and invalidating TLB

entries, can have the side effect of altering the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed. For example, changing MSR[IR]=0 to MRS[IR]=1 has the side effect of enabling translation of instruction addresses. These side effects need not occur in program order (program order refers to the execution of instructions in the strict order in which they occur in the program), and therefore may require explicit synchronization by software.

An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed, is called a “context-altering instruction.”

7 Linux Kernel Bootup and Initialization for PPC 6xx and 4xx

Since this chapter, we will start to go through the Linux kernel for PPC

For different CPU architectures, the boot loader codes vary. From /arch/ppc/Makefile, we can find the related information.

```
ifdef CONFIG_4xx
$(BOOT_TARGETS): $(CHECKS) vmlinux
@$(MAKETREEBOOT) $@
endif

ifdef CONFIG_8xx
$(BOOT_TARGETS): $(CHECKS) vmlinux
@$(MAKECOFFBOOT) $@
@$(MAKEMBXBOOT) $@
endif

ifdef CONFIG_6xx
```

ifndef CONFIG_8260
$(BOOT_TARGETS): $(CHECKS) vmlinux
@$(MAKECOFFBOOT) $@
@$(MAKEBOOT) $@
@$(MAKECHRPBOOT) $@
We will first go through the ppc6xx one and then go back to investigate the boot loader for ppc4xx .

7.1 Moving boot loader to the link address

When powered on, ROM will loads the boot loader (./arch/ppc/boot/head.S) to some arbitary location and jump to the start as the entry address. Then the boot loader will have to move it self to the link address (8M), which is defined by ZLINKFLAGS = -T ../vmlinux.lds -Ttext 0x00800000 in /arch/ppc/boot/makefile.

It first exams whether or not it was already located at the link address by comparing the current position to the symbol address determined during the linkage time. If already located at the link address, it computes the code size and jumps to the start_ldr. Otherwise, it will move itself to the link address.

The detailed codes are explained below.

/arch/ppc/boot/head.S:

.globl start

//The entry point
start:

// By using this jump instruction, the lr register value get updated
// We will use this to decide if we are already located at the 8M link address

```
bl start_
start_:
mr r11,r3 /* Save pointer to residual/board data */
mr r25,r5 /* Save OFW pointer */
// MSR_IP is the Exception Prefix.
// Note that, this field is not used in PPC 4xx.
li r3,MSR_IP /* Establish default MSR value */
// After the mtmsr r3, 0xFFn_nnnn is the exception vector place
mtmsr r3

/* check if we need to relocate ourselves to the link addr or were we
loaded there to begin with -- Cort */

// We retrieve the value of symbol “start” which is defined as 0x800000 at linkage
time.
lis r4,start@h
ori r4,r4,start@l
//The following two instructions is used to get where we were loaded currently.
mflr r3
subi r3,r3,4 /* we get the nip, not the ip of the branch */
mr r8,r3
// Compare and see if we already moved ourself to the link address
cmp 0,r3,r4
// If not, jump forward to the place marked by label 1010.
bne 1010f

1010:
/*
* no matter where we're loaded, move ourselves to -Ttext address
*/
relocate:
```

```
mflr r3 /* Compute code bias */
subi r3,r3,4
mr r8,r3
// With the following 4 instructions, we can compute the size of our image.
lis r4,start@h
ori r4,r4,start@l
lis r5,end@h
ori r5,r5,end@l
addi r5,r5,3 /* Round up - just in case */
sub r5,r5,r4 /* Compute # longwords to move */
// Make sure r5 is aligned to 4 bytes.
srwi r5,r5,2
// r5 contains the size of codes we need to move
mtctr r5
mr r7,r5
// r6 is for the check-sum
li r6,0
subi r3,r3,4 /* Set up for loop */
subi r4,r4,4
// When reach here, r3 contains the address of where we are
// r4 is the pointer of the target address, namely, the link address
00: lwzu r5,4(r3)
stwu r5,4(r4)
xor r6,r6,r5
bdnz 00b

// Up to know, all codes have been moved to the link address
// We retrieve the start_ldr address and jump to there.

lis r3,start_ldr@h
ori r3,r3,start_ldr@l
```

```
mtlr r3 /* Easiest way to do an absolute jump */
blr
```

7.1.2 Clean-up before decompress the kernel

When reach here, the boot loader will do some clean-up work including clearing the BSS section and then call the “decompress_kernel” function which is defined within /arch/ppc/boot/misc.c

```
start_ldr:
/* Clear all of BSS */
// Find the end of data section or the beginning of bss section
lis r3,edata@h
ori r3,r3,edata@l
// Get the end of image
lis r4,end@h
ori r4,r4,end@l
// Prepare to start the loop
subi r3,r3,4
subi r4,r4,4
li r0,0
// Clear bss section
50: stwu r0,4(r3)
cmp 0,r3,r4
bne 50b

90: mr r9,r1 /* Save old stack pointer (in case it matters) */
// load the address of the “stack”
lis r1,.stack@h
ori r1,r1,.stack@l
```

// “stack” is defined with the size of 4096*2.

// Please refer to the definition in the end of this head.S

```
addi r1,r1,4096*2
subi r1,r1,256
li r2,0x000F /* Mask pointer to 16-byte boundary */
andc r1,r1,r2
/* Run loader */
mr r3,r8 /* Load point */
mr r4,r7 /* Program length */
mr r5,r6 /* Checksum */
mr r6,r11 /* Residual data */
mr r7,r25 /* OFW interfaces */
bl decompress_kernel
```

7.1.3 Decompress the kernel

The first things after in decompress kernel is to disable the MMU settings; otherwise, things will get messed up when we set up the BATs and others.

We first flush instruction cache, disable data cache by setting up the HID0 register and then disable the instruction and data translation bits in MSR register. After then, the system will run in real-mode. Please refer to PowerPC specification for details. The corresponding codes are listed below:

```
flush_instruction_cache();
_put_HID0(_get_HID0() & ~0x0000C000);
_put_MSR((orig_MSR = _get_MSR()) & ~0x0030);
```

In /arch/ppc/boot/head.S, several assemble routines are defined including the above ones and others. Listed below are related codes.

```
.globl _get_HID0
```

```
_get_HID0:
mfspr r3,HID0
blr

.globl _put_HID0
_put_HID0:
mtspr HID0,r3
blr

.globl _get_MSR
_get_MSR:
mfmsr r3
blr

.globl _put_MSR
_put_MSR:
mtmsr r3
blr

/*
* Flush instruction cache
*
*/
_GLOBAL(flush_instruction_cache)
// Save the return address to r5
mflr r5
// Flush the data cache
bl flush_data_cache
// Retrieve the current HID0 value
mfspr r3,HID0 /* Caches are controlled by this register */
li r4,0
```

// HID0_ICE: Instruction Cache Enable
// HID0_ICFI: instruction Cache Flash Invalidate
// Setting ICFI clears all the valid bits of the blocks and the PLRU bits. Hardware automatically resets these bits in the next cyble.
// An invalidate operation is issued that marks the state of each instruction cache block as invalid without writing back modified cache blocks to memory. Cache access is blocked during this time. But accesses to the cache are signaled as a miss during invalidate-all operations.
ori r4,r4,(HID0_ICE|HID0_ICFI)
or r3,r3,r4 /* Need to enable+invalidate to clear */
mtspr HID0,r3
andc r3,r3,r4
// Enable I cache. Is it needed?
ori r3,r3,HID0_ICE /* Enable cache */
mtspr HID0,r3
// Restore the return address
mtlr r5
blr

// We have total 32K data cache size

#define NUM_CACHE_LINES 128*8
#define CACHE_LINE_SIZE 32
#define cache_flush_buffer 0x1000

/*
* Flush data cache
*
*/
_GLOBAL(flush_data_cache)
lis r3,cache_flush_buffer@h

```
ori r3,r3,cache_flush_buffer@l
li r4,NUM_CACHE_LINES
mtctr r4
// A load instruction will fill in one data cache line.
// And thus move any modified data(if any) to write back to memory
00: lwz r4,0(r3)
// Adjust the memory pointer by 32 bytes. So that we can make sure that
// a following load instruction will fill in a new cache line
addi r3,r3,CACHE_LINE_SIZE /* Next line, please */
bdnz 00b
// Up to here, we flush all data cache lines.
10: blr
```

decompress_kernel function will retrieve information about zimage_start and the corresponding size and then call gunzip to unzip the compressed image to physical 0 to 4M places.

```
gunzip(0, 0x400000, zimage_start, &zimage_size);
```

Note that ppc-linux image text section starts from 0xC0000000, whereas i386-linux image text section starts from 0xC0100000. Please refer to the makefile and link scripts, respectively under /arch/ppc and /arch/i386.

7.1.4 Ready to jump to kernel

After unzip the kernel image, the boot loader will be ready to jump to the kernel entry address. The start address of kernel codes is retrieved from the coff entry.

```
/* tell kernel we're prep */
/*
* get start address of kernel code which is stored as a coff
```

```
* entry. see boot/head.S -- Cort
*/
li r9,0x4
mtlr r9
lis r10,0xdeadc0de@h
ori r10,r10,0xdeadc0de@l
li r9,0
stw r10,0(r9)
// We use r9 as a pointer and save an instruction into the r9 pointed memory—
0x00000004.

And then we disable all BATs settings.
li r8,0
mtspr DBAT0U,r8
mtspr DBAT1U,r8
mtspr DBAT2U,r8
mtspr DBAT3U,r8
mtspr IBAT0U,r8
mtspr IBAT1U,r8
mtspr IBAT2U,r8
mtspr IBAT3U,r8
//Jump to kernel
blr
```

7.2 Boot loader for ppc 4xx

If the target CPU is 4xx, for example, PowerPC 405GP, linux will use /arch/ppc/treeboot for its bootloader. Please refer to /arch/ppc Makefile for details.

In the following, we illustrate the related codes that are found in Monta Vista’s Linux

distribution.

Makefile:

```
ifdef CONFIG_IBM405
LD_ARGS = -e _start -T ld.script -Ttext 0x00450000 -Bstatic
else
LD_ARGS = -e _start -T ld.script -Ttext 0x00200000 -Bstatic
endif



treeboot: $(OBJS) ld.script
$(LD) -o $@ $(LD_ARGS) $(OBJS) $(LIBS)

zImage: vmlinux.img

zImage.initrd: vmlinux.initrd.img

treeboot.image: treeboot vmlinux.gz
$(OBJCOPY) --add-section=image=vmlinux.gz treeboot $@

treeboot.initrd: treeboot.image ramdisk.image.gz
$(OBJCOPY) --add-section=initrd=ramdisk.image.gz treeboot.image $@

vmlinux.img: treeboot.image
$(OBJDUMP) --syms treeboot.image | grep irSectStart > irSectStart.txt
$(MKIRIMG) treeboot.image treeboot.image.out irSectStart.txt
$(MKEVIMG) treeboot.image.out $@
$(RM) treeboot.image treeboot.image.out irSectStart.txt

vmlinux.initrd.img: treeboot.initrd
```

```
$(OBJDUMP) --all-headers treeboot.initrd | grep irSectStart > irSectStart.txt
$(MKIRIMG) treeboot.initrd treeboot.initrd.out irSectStart.txt
$(MKEVIMG) treeboot.initrd.out $@
$(RM) treeboot.initrd treeboot.initrd.out irSectStart.txt

vmlinux.gz: $(TOPDIR)/vmlinux
$(OBJCOPY) -S -O binary $(TOPDIR)/vmlinux vmlinux
$(GZIP) vmlinux
```

Boot loader execution entry point is the “_start” symbol, which is defined within /arch/ppc/treeboot/crt0.S.

First, the bootload clears the BSS section, flush the caches and set up a stack for later usage.

```
#include "../kernel/ppc_asm.tmpl"
.text
.globl _start
_start:

## The bootrom knows that the address of the bootrom read only
## structure is 4 bytes after _start. Ugly, but not as ugly as
## other possible ways for the bootrom to communicate with
## treeboot

// Jump forward. The 4 bytes after _start contains some real only data structure.
b 1f
.long 0x62726f6d # structure ID - "brom"
.long 0x5f726f00 # - "_ro\0"
```

```
.long 1 # structure version
.long bootrom_cmdline # address of *bootrom_cmdline
1:

## Clear out the BSS as per ANSI C requirements

// _end and _bss_start symbols are created when doing the linkage.

lis r7,_end@ha #
addi r7,r7,_end@l # r7 = &_end
lis r8,__bss_start@ha #
addi r8,r8,__bss_start@l # r8 = &_bss_start

## Determine how large an area, in number of words, to clear

subf r7,r8,r7 # r7 = &_end - &_bss_start + 1
addi r7,r7,3 # r7 += 3
// r7 contains the size in bytes.
// we need divide it by 4 to obtain the size in words.
// The below instruction shift r7 2 bits and set flags if any.
srwi. r7,r7,2 # r7 = size in words.
beq 2f # If the size is zero, do not bother
// Prepare the offsetd because we will use stwu instruction for the save purpose
addi r8,r8,-4 # r8 -= 4
// Set up the counter
mtctr r7 # SPRN_CTR = number of words to clear
// Initialize the zero value
li r0,0 # r0 = 0
1: stwu r0,4(r8) # Clear out a word
bdnz 1b # If we are not done yet, keep clearing
// Up to here, the bss section is cleared.
```

```
## Flush and invalidate the caches for the range in memory covering
## the .text section of the boot loader

2: lis r9,_start@h # r9 = &_start
lis r8,_etext@ha #
addi r8,r8,_etext@l # r8 = &_etext
// dcbf is to fluch the cache line containing the value of address r0+r9
3: dcbf r0,r9 # Flush the data cache
// icbi is to invalidate the cache line containing the value of address r0+r9
icbi r0,r9 # Invalidate the instruction cache
// Move to next cache line by add 32 bytes
// Per cache line are 32 bytes size
addi r9,r9,0x10 # Increment by one cache line
cmplwi cr0,r9,r8 # Are we at the end yet?
blt 3b # No, keep flushing and invalidating

## Set up the stack

// Get the text section entry address
lis r9,_start@h # r9 = &_start (text section entry)
addi r9,r9,_start@l
// Set up the stack pointer-r1
subi r1,r9,64 # Start the stack 64 bytes below _start
clrrwi r1,r1,4 # Make sure it is aligned on 16 bytes.
// clear the first location of the stack to be zero. This is required by EABI convention.
// The zero value means that it is no any more stack chain. In other words, this is the root
// of the stack.
li r0,0
stwu r0,-16(r1)
// Set up the return address.
```

```
mtlr r9
// Jump to main.c in /arch/ppc/treeboot
b start # All done, start the real work.
```

In the following, we will go through the main.c to see how bootloader prepares the linux image, including gunzip and cache flush and transferring the control to linux image.

```
/* Linux kernel image section */

// imageSect_start means the linux image start address. Please refer to the makefile.
// Basically, the compressed linux image (vmlinux.gz) is inserted into the boot loader as a
// section named “image” by using $(OBJCOPY) --add-section=image=vmlinux.gz
// treeboot $@

im = (unsigned char *)(imageSect_start);
// Get the length of the image section
len = imageSect_size;
// We have to uncompress the compressed image into the physical address 0x0 so that we
//can map the corresponding virtual address start 0xC0000000.
// #define RAM_PBASE 0x00000000
// #define RAM_VBASE 0xC0000000
// #define RAM_START RAM_PBASE
// #define PROG_START RAM_START

dst = (void *)PROG_START;

// Check the sanity
/* Check for the gzip archive magic numbers */
if (im[0] == 0x1f && im[1] == 0x8b) {
```

```
/* The gunzip routine needs everything nice and aligned */

void *cp = (void *)ALIGN_UP(RAM_FREE, 8);
avail_ram = (void *)(cp + ALIGN_UP(len, 8)); /* used by zalloc() */
memcpy(cp, im, len);

/* I'm not sure what the 0x200000 parameter is for, but it works. */
/* It tells gzip the end of the area you wish to reserve, and it
* can use data past that point....unfortunately, this value
* isn't big enough (luck ran out). -- Dan
*/
// The maximum image size would be 4M.

gunzip(dst, 0x400000, cp, (int *)&len);
} else {
// If reach here, it means that the image is not compressed and thus we simply copy it to
the destination.
memmove(dst, im, len);
}

// After moving the linux image to the physical addressed starting from the zero, we need
// to flush all the instruction and data caches in order not to face any confusing in the
// future. We will explain this function later.
flush_cache(dst, len);

// Get the start entry address of the linux image, which should be defined
by ./arch/ppc/kernel/headxx.S, for ppc405, it is the head4xxS.

sa = (unsigned long)dst;
// Now, we are ready to jump into the REAL linux world!
(*(void (*)())sa)(&board_info,
```

```
initrd_start,
initrd_start + initrd_size,
cmdline,
cmdline + strlen(cmdline));

pause();



/*
* Flush the dcache and invalidate the icache for a range of addresses.
*
* flush_cache(addr, len)
*/
.global flush_cache
flush_cache:
mfpvr r5 # Get processor version register
extrwi r5,r5,16,0 # Get the version bits
cmpwi cr0,r5,0x0020 # Is this a 403-based processor?
beq 1f # Yes, it is
// Each cache line in 405 CPU has the size of 32 bytes.
li r5,32 # It is not a 403, set to 32 bytes
// The following two instructions are to convert the len(r4) to number of lines
// len +=line_size  - 1 is to round up the length; and then right shift 5 bits so as to
// make sure the number of lines will cover the last 32 bytes(cache line).
// For example, if len =34 bytes, then the number of cache line need to be cover is
// 2.
addi r4,r4,32-1 # len += line_size - 1
srwi. r4,r4,5 # Convert from bytes to lines
b 2f
// Each cache line in 403 CPU has the size of 16 bytes.
```

1: li r5,16 # It is a 403, set to 16 bytes

addi r4,r4,16-1 # len += line_size - 1

srwi. r4,r4,4 # Convert from bytes to lines

2: mtctr r4 # Set-up the counter register

beqlr # If it is 0, we are done

3:

// dcbf stands for data cache block flush. Its format is:

// dcbf RA, RB.

/*

(RA|0) + (RB) ¬EA

DCBF(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise. If the data block corresponding to the EA is in the data cache and marked as modified (stored into), the data block is copied back to main storage and then marked invalid in the data cache. If the data block is not marked as modified, it is simply marked invalid in the data cache. The operation is performed whether or not the EA is marked as cachable. If the data block at the EA is not in the data cache, no operation is performed.

*/

dcbf r0,r3 # Flush and invalidate the data line

// icbi stands for instruction cache block invalidate. Its format is:

// icbi RA, RB.

/*

|0) + (RB)A(R ¬EA

ICBI(EA)

An effective address (EA) is formed by adding an index to a base address. The index is the contents

of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA otherwise.

If the instruction block at the EA is in the instruction cache, the cache block is marked invalid.
If the instruction block at the EA is not in the instruction cache, no additional operation is performed.
*/

icbi r0,r3 # Invalidate the instruction line
// By adding 32 or 16 bytes, we move to the second cache line.
add r3,r3,r5 # Move to the next line
bdnz 3b # Are we done yet?
// make sure all data over the bus will be done 100%
sync
isync
blr # Return to the caller

8. Kernel Initialization

This part codes are found in /arch/ppc/kernel/head.S, head4xx.S or head8xx.S, depending on the target CPUs.

For head4xx.S, it starts from the “_GLOBAL(_start)”, which virtual address is the 0xC0000000. After being here, all the kernel image is already loaded into physical address 0x00000000. Therefore, the first thing the kernel need to do is to set up the mapping of its virtual address to physical address in order to transfer its control the “start_here”. Also, the head4xx.S will set up the exception/interrupt vector starting from 0xC0000100 to 0xC0002000. The corresponding codes are illustrated below:

.text
_GLOBAL(_stext)
// _start is the entry address of the kernel image. Bootloader transfers its CPU control to here after the kernel image is uncompressed.

// _start’s virtual address is: 0xC000000; its physical address is: 0x0000000.

// The reason why we don’t need MMU enabled here is: MMU is useless until a reference to a virtual address, for example, a “b” or “a “bl” instruction will be used.

_GLOBAL(_start)

Save residual data, init RAM disk, and command line parameters

// Those parameters are passed by bootloader.

mr r31,r3

mr r30,r4

mr r29,r5

mr r28,r6

mr r27,r7

Set the ID for this CPU

li r24,0

Invalidate all TLB entries

// Invalidate all TLB entries. Note that tlbia is an optional PowerPC instruction. For

// example, tlbia is not available in ppc750 CPU.

// If not a 4xx CPU, tlbia will be implemented as a macro

// #if !defined(CONFIG_4xx) && !defined(CONFIG_8xx)

// #define tlbia \

// li r4,1024; \

// mtctr r4; \

// lis r4,KERNELBASE@h; \

// 0: tlbie r4; \

// addi r4,r4,0x1000; \

// bdnz 0b

// #endif

tlbia # remove all stale TLB entries
// make sure we finish the tlb invalidation.
sync

We should still be executing code at physical address 0x0000xxxx
at this point. However, start_here is at virtual address
0xC000xxxx. So, set up a TLB mapping to cover this once
translation is enabled.
// Ready to set up the MMU. For 4xx CPU, there is no explicit page table, providing a read/write-able TLB for the mapping purpose. For 6xx CPU, for instance, 750, TLB is not read/write-able by system software, but controlled by hardware. Intel X86 also works this way.
// KERNELBASE is defined as the value of 0xC000000. Please refer to ./include/asm-
// ppc/page.h
// #define PAGE_OFFSET 0xc0000000
// #define KERNELBASE PAGE_OFFSET

lis r3,KERNELBASE@h # Load the kernel virtual address
ori r3,r3,KERNELBASE@l
// tophys is to convert a virtual address into a physical address.
// Its definition is:
// #define tophys(rd,rs) addis rd,rs,-KERNELBASE@h;
// For example, for virtual address 0xC000000, its corresponding physical address is:0x0
// For virtual address 0xC0100000, the corresponding physical address is: 0x00100000.

tophys(r4,r3) # Load the kernel physical address

// Save the existing PID value into r7 and assign the new PID value as zero.
// The PID value will contribute to the TLB entry setting up. Please refer to
// PPC405 manual for more details.

Save the existing PID and load the kernel PID.

mfspr r7,SPRN_PID # Save the old PID
li r0,0
mtspr SPRN_PID,r0 # Load the kernel PID
sync

// Ready to set up TLB; r4 contains the physical address; r3 contains the virtual address

Configure and load entry into TLB slot 0.
// Clear the right 10 bites of r4 ‘cause the least page table size is 1K.
// These 10 bits will be used for the TLB attributes, like Read/Write, Executable, Zone
// information (4xx specific) and WIMG information. Please refer to PPC4XX
// specification for more details.
clrrwi r4,r4,10 # Mask off the real page number
// Set this page as write-able and executable
ori r4,r4,(TLB_WR | TLB_EX) # Set the write and execute bits
//Clear the right 10 bits of r3 for setting up the high word of an TLB entry.
clrrwi r3,r3,10 # Mask off the effective page number
// Set this TLB entry is valid and will cover 16M from 0xC0000000 to 0xC1000000.
ori r3,r3,(TLB_VALID | TLB_PAGESZ(PAGESZ_16M))

li r0,0 # TLB slot 0
// Fill in the TLB entry.
tlbwe r4,r0,TLB_DATA # Load the data portion of the entry
tlbwe r3,r0,TLB_TAG # Load the tag portion of the entry

// Below is for setting up the TLB entry for covering the UART addresses.

#ifdef CONFIG_DEBUG_BRINGUP /* MVISTA_LOCAL - begin nomerge */

```
#ifdef CONFIG_IBM405
## also see arch/ppc/mm/init.c

#if 0
# This is an example of how to print single character for early bringup
# debugging, once the uart0 TLB mapping is in place
lis r3,UART0_BASE@h
addi r3,r3,UART0_BASE@l
li r0, 0x61 # ascii 'a'
stb r0,0(r3) # write a byte to uart0
#endif

## uart0 base is at 0xef600300, page starts at 0xef600000
## uart1 base is at 0xef600400
## page size of 4k will include both uarts
#define UART0_BASE 0xef600300

## set up a TLB mapping to cover uart0
lis r3,UART0_BASE@h # Load the virtual address
addi r3,r3,UART0_BASE@l

ori r4, r3, 0 # Load the physical address

clrrwi r4,r4,10 # Mask off the real page number
# write, execute, cache inhibit, guarded
ori r4,r4,(TLB_WR | TLB_EX | TLB_I | TLB_G)

clrrwi r3,r3,10 # Mask off the effective page number
ori r3,r3,(TLB_VALID | TLB_PAGESZ(PAGESZ_4K))

li r0,1 # TLB slot 1
```

```
tlbwe r4,r0,TLB_DATA # Load the data portion of the entry
tlbwe r3,r0,TLB_TAG # Load the tag portion of the entry

#endif
#endif /* MVISTA_LOCAL - end nomerge */
isync
// Restore the previous process ID.

#if !defined(CONFIG_IBM405)
/* ftr revisit - why is the PID changed to something other than
** the kernel PID??
*/
mtspr SPRN_PID,r7 # Restore the existing PID
#endif

// Read to set up the exception vector
// The exception vector will start at 0xC0000100.

## Establish the exception vector base

lis r4,KERNELBASE@h # EVPR only uses the high 16-bits
tophys(r0,r4) # Use the physical address
// Note that, the SPRN_EVPR register has to contain a physical address, which will be
// used by hardware when an exception happens to locate the corresponding exception
// handler.

// The exception handler codes will be explained in the following. Codes for this part are
written very sophisticated.
mtspr SPRN_EVPR,r0
```

Enable the MMU and jump to the main PowerPC kernel start-up code

#ifdef CONFIG_405
// MSR_KERNEL is defined as MSR_IR | MSR_DR | MSR_ME | MSR_DE,
// which means, enable instruction mapping, data mapping, machine check and debug
// interrupt bits.
li r0,MSR_KERNEL
#else
// for other 4xx CPUs
mfmsr r0 # Get the machine state register
ori r0,r0,(MSR_DR | MSR_IR) # Enable data and instr. translation
#endif
// set up the srr0 and srr1 registers
// srr1 will contain the future MSR value, including enabling the MMU.
mtspr SPRN_SRR1,r0 # Set up the new machine state register
// Retrieve the address of “start_here” symbol as the jump address. Note that, in order to
// do this, we have to enable MMU. Before reach here, we are ok without MMU enabled.
lis r0,start_here@h
ori r0,r0,start_here@l
mtspr SPRN_SRR0,r0 # Set up the new instruction pointer
// Ready to go.
rfi # Jump to start_here w/ translation on

After the system jumps into start_here, it is ready to do some real setup work for the linux kernel.

First, we set up the init_task_union, clear the kernel bss section, set up the kernel stack and then initialize the MMU. After those finished, we jump to the start_kernel.

init_task_union is declared in ./arch/ppc/kernel/process.c. Note that the task struct is

combined with the kernel stack that is 8k size. In other words, for every task union data structure, it occupies 8k memory, in which, the low part is the task_struct; the high part is the kernel stack.

```
#ifndef INIT_TASK_SIZE
# define INIT_TASK_SIZE 2048*sizeof(long)
#endif

union task_union {
struct task_struct task;
unsigned long stack[INIT_TASK_SIZE/sizeof(long)];
};



/* this is 16-byte aligned because it has a stack in it */
union task_union __attribute((aligned(16))) init_task_union = {
INIT_TASK(init_task_union.task)
};

## Establish a pointer to the current task

lis r2,init_task_union@h
ori r2,r2,init_task_union@l
// r2 points to the current task data structure union

## Clear out the BSS as per ANSI C requirements

lis r7,_end@ha
addi r7,r7,_end@l
lis r8,__bss_start@ha
addi r8,r8,__bss_start@l
```

```
subf r7,r8,r7
addi r7,r7,3
srwi. r7,r7,2
beq 2f
addi r8,r8,-4
mtctr r7
li r0,0
3: stwu r0,4(r8)
bdnz 3b
```

Set up the kernel stack of init task. This includes setting up the r1 register and clean up the top stack frame. For linux, the minum stack frame is the size of 16, which is defined as #define STACK_FRAME_OVERHEAD 16 /* size of minimum stack frame */ in ./include/asm-ppc/ptrace.h

```
## Stack
// Add 8K size so that r1 points to the kernel stack top
2: addi r1,r2,TASK_UNION_SIZE
li r0,0
//Clean up the  - 16(r1) and also updates the value of r1 to the position of  - 16(r1).
stwu r0,-STACK_FRAME_OVERHEAD(r1)
```

Now it is time to do the MMU init by calling the MMU_init, which is defined within arch/ppc/mm/init.c. For 4xx CPU, we will fill out the TLB entries to map the virtual address from 0xC0000000. The page size we used is 16M.

```
// Figure out the size of DRAM we have so that we know how many TLB entries we need use.
pinned_tlbs = (size_DRAM / SIZE_16MB) + (size_DRAM % SIZE_16MB ? 1: 0);
start_vaddr = KERNELBASE;
start_paddr = 0x0;
```

```
for (i = 0; i < pinned_tlbs; i++) {
PPC4xx_tlb_pin(start_vaddr, start_paddr, TLB_PAGESZ(PAGESZ_16M), 1);
start_vaddr += SIZE_16MB;
start_paddr += SIZE_16MB;
}
```

```
/*
* Find the top of physical memory and map all of it in starting
* at KERNELBASE. end_of_DRAM is a virtual address.
*/

end_of_DRAM = ppc4xx_find_end_of_memory();

mapin_ram();
```

Besides the virtual and physical address mapping, MMU_init will also do some cache settings for real-mode. PPC 4xx has DCWR, DCCR and ICCR registers to control the caching behaviors when in real mode.

```
/*
* Set up the real-mode cache parameters for the exception vector
* handlers (which are run in real-mode).
*
* They don't affect virtual accesses.
*/

mtspr(SPRN_DCWR, 0x00000000); /* All caching is write-back */

/*
* Cache instruction and data space where the exception
* vectors and the kernel live in real-mode.
```

```
*/
/* ftr_revisit - memory size > or < 128 MB */
mtspr(SPRN_DCCR, 0x80000000); /* 128 MB of data space at 0x0. */
mtspr(SPRN_ICCR, 0x80000000); /* 128 MB of instr. space at 0x0. */
```

After finishing the MMU initialization, we first change back to MMU-disabled and then set up the kernel context. After finished, we enable the MMU and jump to the kernel C level—start_kernel().

```
## Go back to running unmapped so that we can change to our
## exception vectors.

lis r4,2f@h
ori r4,r4,2f@l
tophys(r4,r4)
li r3,MSR_KERNEL & ~(MSR_IR|MSR_DR)
mtspr SPRN_SRR0,r4 # Set up the instruction pointer
mtspr SPRN_SRR1,r3 # Set up the machine state register

rfi

## Load up the kernel context

2: SYNC # Force all PTE updates to finish

## Set up for using our exception vectors
// r4 will hold the physical address of the init thread/task data structure.
tophys(r4,r2) # Pointer to physical current thread
addi r4,r4,THREAD # The init task thread
// SPRN_SPRG3 holds the pointer to the tss thread_struct.
```

```
mtspr SPRN_SPRG3,r4 # Save it for exceptions later
li r3,0 #
#ifdef CONFIG_IBM405
# ftr revisit
nop
nop
nop
nop
nop
nop
nop
nop
#endif
mtspr SPRN_SPRG2,r3 # 0 implies r1 has kernel stack pointer

## Really turn on the MMU and jump into the kernel

lis r4,MSR_KERNEL@h
ori r4,r4,MSR_KERNEL@l
lis r3,start_kernel@h
ori r3,r3,start_kernel@l
mtspr SPRN_SRR0,r3 # Set up the instruction pointer
mtspr SPRN_SRR1,r4 # Set up the machine state register
rfi # Enable the MMU, jump to the kernel
```

9 . Kernel Setup---start_kernel

Start_kernel() is the entry for all linux platforms after the kernel finishes the initialization.

“The start_kernel() function initializes all kernel data and then starts the "init" kernel thread. One of the first things that happens in start_kernel() is a call to setup_arch(), an architecture-specific setup function which handles low-level initialization details. For ppc

platforms, that function lives in arch/ppc/kernel/setup.c.
The first memory-related thing setup_arch() does is compute the number of low-memory and high-memory pages available; the highest page numbers in each memory type get stored in the global variables highstart_pfn and highend_pfn, respectively. High memory is memory not directly mappable into kernel VM; this is discussed further below.
Next, setup_arch() calles init_bootmem() to initialize the boot-time memory allocator. The bootmem allocator is used only during boot, to allocate pages for permanent kernel data. We will not be too much concerned with it henceforth. The important thing to remember is that the bootmem allocator provides pages for kernel initialization, and those pages are permanently reserved for kernel purposes, almost as if they were loaded with the kernel image; they do not participate in any MM activity after boot.
setup_arch() will also call a call-back function --ppc_md.setup_arch() to initialize the ppc-machine dependent architecture. This is unique for powerpc linux ‘cause different powerpc CPU may have total different architectures. For example, ppc4xx and ppc6xx are much different for their MMU, TLB and so on.

When system boots up, the identify_machine() function (./arch/ppc/kernel/setup.c) will be called inside the head_4xx.S before the MMU_init. Within identify_machine(), system will invoke the function ppc405_init(), which resides in ./arch/ppc/kernel/ppc405_setup.c. ppc405_init() will fill in the call-back function entries for a global structure ppc_md that is defined as struct machdep_calls ppc_md in ./arch/ppc/kernel/setup.c. For ppc405, the ppc_md will be set up as follows:

/* Initialize machine-dependency vectors */

```
ppc_md.setup_arch = ppc405_setup_arch;
ppc_md.setup_residual = ppc4xx_setup_residual;
ppc_md.get_cpuinfo = ppc4xx_get_cpuinfo;
ppc_md.irq_cannonicalize = NULL;
ppc_md.init_IRQ = ppc405_init_IRQ;
ppc_md.get_irq = ppc405_get_irq;
```

```
ppc_md.init = NULL;

ppc_md.restart = ppc405_restart;
ppc_md.power_off = ppc405_power_off;
ppc_md.halt = ppc405_halt;

ppc_md.time_init = ppc405_time_init;
ppc_md.set_rtc_time = ppc405_set_rtc_time;
ppc_md.get_rtc_time = ppc405_get_rtc_time;
ppc_md.calibrate_decr = ppc405_calibrate_decr;

ppc_md.heartbeat = NULL;
ppc_md.heartbeat_reset = 0;
ppc_md.heartbeat_count = 0;

ppc_md.progress = ppc4xx_progress;

ppc_md.nvram_read_val = NULL;
ppc_md.nvram_write_val = NULL;

ppc_md.kbd_setkeycode = NULL;
ppc_md.kbd_getkeycode = NULL;
ppc_md.kbd_translate = NULL;
ppc_md.kbd_unexpected_up = NULL;
ppc_md.kbd_leds = NULL;
ppc_md.kbd_init_hw = NULL;

#if defined(CONFIG_MAGIC_SYSRQ)
ppc_md.ppc_kbd_sysrq_xlate = NULL;
#endif
```

```
/*
** ppc_md.pcibios_read_config_*()
** ppc_md.pcibios_write_config_*()
*/
#ifdef CONFIG_PCI
set_config_access_method(ppc405);
ppc_md.pcibios_fixup_bus = ppc405_pcibios_fixup_bus;
#else
ppc_md.pcibios_fixup_bus = NULL;
#endif
ppc_md.pcibios_fixup = NULL;
```

With those call-back pointers, for example, ppc_md.setup_arch= ppc405_setup_arch, kernel will be able to correctly invoke the corresponding codes for different ppc CPUs.

After setup_arch(), we do some additional setup of other kernel subsystems, some of which allocate additional kernel memory using the bootmem allocator. Important among these, from the MM point of view, is kmem_cache_init(), which initializes the slab allocator data.

Shortly after kmem_cache_init() is called, we call mem_init(). This function completes the freelist initialization begun in free_area_init() by clearing the PG_RESERVED bit in the zone data for free physical pages; clearing the PG_DMA bit for pages that can't be used for DMA; and freeing all usable pages into their respective zones. That last step, done in free_all_bootmem_core() in mm/bootmem.c, is interesting: it builds the buddy bitmaps and freelists describing all existing non-reserved pages by simply freeing them and letting free_pages_ok() do the right thing. Once mem_init() is called, the bootmem allocator is no longer usable, since all its pages have been freed into the zone allocator's world.

Besides those sub-systems’ initializations for memory related, start_kernel() will also

initialize other sub-systems like: interrupt handler, scheduler, file system, ipc, to name a few.

sched_init() will move itself into lazy tlb mode after initilizing three bottom half queues including timer's timer_bh, tqueue_bh and an immediate_bh.

```
init_bh(TIMER_BH, timer_bh);
init_bh(TQUEUE_BH, tqueue_bh);
init_bh(IMMEDIATE_BH, immediate_bh);

/*
* The boot idle thread does lazy MMU switching as well:
*/
atomic_inc(&init_mm.mm_count);
enter_lazy_tlb(&init_mm, current, cpu);
```

Lastly, start_kernel() will activate other CPUs by calling the smp_init(); and then spawn the kernel thread—init. When falling through the init(), the kernel status is that none of the devices have been touched yet, but the CPU subsystem is up and running, and memory and process management works. Init() will first invoke the do_basic_setup() to do device drivers' setup including pci setup, usb init and the network init by using sock_init(); also the mounting file system get called here. For ppc part, the ppc_init() will be called.

init() will eventually try to execute one of the following init processes until one succeeds.

```
execve("/sbin/init",argv_init,envp_init);
execve("/etc/init",argv_init,envp_init);
execve("/bin/init",argv_init,envp_init);
execve("/bin/sh",argv_init,envp_init);
```

After finishing the creation of kernel thread of init, start_kernel() will fall through idle state and give away CPU. Therefore, init() thread will then get the CPU control and the

whole system get running normally by then.

```
unlock_kernel();
current->need_resched = 1;
cpu_idle();

asmlinkage void __init start_kernel(void)
{
char * command_line;
unsigned long mempages;
extern char saved_command_line[];
/*
* Interrupts are still disabled. Do necessary setups, then
* enable them
*/
lock_kernel();
printk(linux_banner);
setup_arch(&command_line);
printk("Kernel command line: %s\n", saved_command_line);
parse_options(command_line);
/* The above two lines will install interrupt handlers
trap_init();
init_IRQ();

sched_init();
time_init();
softirq_init();

/*
* HACK ALERT! This is early. We're enabling the console before
* we've done PCI setups etc, and console_init() must be aware of
```

```
* this. But we do want output early, in case something goes wrong.
*/
console_init();
#ifdef CONFIG_MODULES
init_modules();
#endif
if (prof_shift) {
unsigned int size;
/* only text is profiled */
prof_len = (unsigned long) &_etext - (unsigned long) &_stext;
prof_len >>= prof_shift;

size = prof_len * sizeof(unsigned int) + PAGE_SIZE-1;
prof_buffer = (unsigned int *) alloc_bootmem(size);
}

kmem_cache_init();
sti();
calibrate_delay();
#ifdef CONFIG_BLK_DEV_INITRD
if (initrd_start && !initrd_below_start_ok &&
initrd_start < min_low_pfn << PAGE_SHIFT) {
printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "
"disabling it.\n",initrd_start,min_low_pfn << PAGE_SHIFT);
initrd_start = 0;
}
#endif
mem_init();
kmem_cache_sizes_init();
#ifdef CONFIG_3215_CONSOLE
con3215_activate();
```

```
#endif
#ifdef CONFIG_PROC_FS
proc_root_init();
#endif
mempages = num_physpages;

fork_init(mempages);
proc_caches_init();
vfs_caches_init(mempages);
buffer_init(mempages);
page_cache_init(mempages);
kiobuf_setup();
signals_init();
bdev_init();
inode_init(mempages);
#if defined(CONFIG_SYSVIPC)
ipc_init();
#endif
#if defined(CONFIG_QUOTA)
dquot_init_hash();
#endif
check_bugs();
printk("POSIX conformance testing by UNIFIX\n");

/*
* We count on the initial thread going ok
* Like idlers init is an unlocked kernel thread, which will
* make syscalls (and thus be locked).
*/
smp_init();
kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
```

```
unlock_kernel();
current->need_resched = 1;
cpu_idle();
}



void __init mem_init(void)
{
extern char *sysmap;
extern unsigned long sysmap_size;
unsigned long addr;
int codepages = 0;
int datapages = 0;
int initpages = 0;
#ifdef CONFIG_HIGHMEM
unsigned long highmem_mapnr;

highmem_mapnr = total_lowmem >> PAGE_SHIFT;
highmem_start_page = mem_map + highmem_mapnr;
max_mapnr = total_memory >> PAGE_SHIFT;
totalram_pages += max_mapnr - highmem_mapnr;
#else
max_mapnr = max_low_pfn;
#endif /* CONFIG_HIGHMEM */

high_memory = (void *) __va(max_low_pfn * PAGE_SIZE);
num_physpages = max_mapnr; /* RAM is assumed contiguous */

totalram_pages += free_all_bootmem();
```

```
#ifdef CONFIG_BLK_DEV_INITRD
/* if we are booted from BootX with an initial ramdisk,
make sure the ramdisk pages aren't reserved. */
if (initrd_start) {
for (addr = initrd_start; addr < initrd_end; addr += PAGE_SIZE)
clear_bit(PG_reserved, &virt_to_page(addr)->flags);
}
#endif /* CONFIG_BLK_DEV_INITRD */

#if defined(CONFIG_ALL_PPC)
/* mark the RTAS pages as reserved */
if ( rtas_data )
for (addr = (ulong)__va(rtas_data);
addr < PAGE_ALIGN((ulong)__va(rtas_data)+rtas_size) ;
addr += PAGE_SIZE)
SetPageReserved(virt_to_page(addr));
#endif /* defined(CONFIG_ALL_PPC) */
if ( sysmap_size )
for (addr = (unsigned long)sysmap;
addr < PAGE_ALIGN((unsigned long)sysmap+sysmap_size) ;
addr += PAGE_SIZE)
SetPageReserved(virt_to_page(addr));

for (addr = PAGE_OFFSET; addr < (unsigned long)end_of_DRAM;
addr += PAGE_SIZE) {
if (!PageReserved(virt_to_page(addr)))
continue;
if (addr < (ulong) etext)
codepages++;
else if (addr >= (unsigned long)&__init_begin
```

```
&& addr < (unsigned long)&__init_end)
initpages++;
else if (addr < (ulong) klimit)
datapages++;
}

#ifdef CONFIG_HIGHMEM
{
unsigned long pfn;

for (pfn = highmem_mapnr; pfn < max_mapnr; ++pfn) {
struct page *page = mem_map + pfn;

ClearPageReserved(page);
set_bit(PG_highmem, &page->flags);
atomic_set(&page->count, 1);
__free_page(page);
totalhigh_pages++;
}
totalram_pages += totalhigh_pages;
}
#endif /* CONFIG_HIGHMEM */

printk("Memory: %luk available (%dk kernel code, %dk data, %dk init, %ldk
highmem)\n",
(unsigned long)nr_free_pages()<< (PAGE_SHIFT-10),
codepages<< (PAGE_SHIFT-10), datapages<< (PAGE_SHIFT-10),
initpages<< (PAGE_SHIFT-10),
(unsigned long) (totalhigh_pages << (PAGE_SHIFT-10)));
mem_init_done = 1;
}
```

10 Exception Handler:

Exception handlers are set up in the head4xx.S by using several well-written macros. We first explain those macros.

```
###
### Macros for specific exception types
###
```

// This macro is used to define the location of an exception handler. For example, START_EXCEPTION(0x0100, CriticalInterrupt) means that CriticalInterrupt handler will start at address 0x0100.

```
#define START_EXCEPTION(n, label) \
. = n; \
label:
```

// This macro is used to invoke the transfer_to_handler function which is used as doing
// some preparation work before doing the real work specified by the “func” function.
// ret_from_except is a function defined in ./arch/ppc/kernel/entry.S. It is used to restore
// the previous context back to the one before this exception happens.

// Within transfer_to_handler or before we go to the exception handler, MMU will be
// enabled in order to do the real work, for example, do_page_fault().

// Very interestingly, we don’t need enable MMU for the “bl transfer_to_handler”.
// The reason is: bl instruction is implemented as CIA+OFFSET, not a absolute jump.
// In other words, the PC counter is able to reach the transfer_to_handler function codes
// without having any virtual address references.

```
#define FINISH_EXCEPTION(func) \
```

```
bl transfer_to_handler; \
.long func; \
.long ret_from_except

// This is the entry macro for defining a standard exception handler.

#define STND_EXCEPTION(n, label, func) \
// Specify the addresses where the handler will be located
START_EXCEPTION(n, label); \
// Common exception code for standard (non-critical) exceptions.
STND_EXCEPTION_PROLOG; \
addi r3,r1,STACK_FRAME_OVERHEAD; \
li r7,STND_EXC; \
li r20,MSR_KERNEL; \
// Go to exception processing.
FINISH_EXCEPTION(func)

// This is the entry macro for defining a critical exception handler.

#define CRIT_EXCEPTION(n, label, func) \
// Specify the addresses where the handler will be located
START_EXCEPTION(n, label); \
// Common exception code for critical exceptions.
CRIT_EXCEPTION_PROLOG; \
addi r3,r1,STACK_FRAME_OVERHEAD; \
li r7,CRIT_EXC; \
li r20,MSR_KERNEL; \
// Go to exception processing.
FINISH_EXCEPTION(func)
```

COMMON_PROLOG contains the common codes for all exception handling. Basically, it sets up the stack and then save some contexts (for example, LR, CTR and XER) for the

exception handling. The stack here is the current kernel stack which physical address is saved in SPRN_SPRG2. SPRG3 has the physical address of the current task thread_struct. Please refer to the _switch(previous task, next task) routine in ./arch/ppc/kernel/entry.S. We will explain this later. Note also that the kernel uses the SPRN_SPRGx to temporally save register values, instead of using some pre-defined memory areas.

```
#define COMMON_PROLOG
// Temporally save the r20 and r21 to SPRG0 and SPRG1. \
0: mtspr SPRN_SPRG0,r20; /* We need r20, move it to SPRG0 */\
mtspr SPRN_SPRG1,r21; /* We need r21, move it to SPRG1 */\
// Use r20 to hold the CR flag
mfcr r20; /* We need the CR, move it to r20 */\
// SPRG2 contains the address of current kernel stack
mfspr r21,SPRN_SPRG2; /* Exception stack to use */\
// Detect to see if this exception is from user mode or RTAS.
// When with RTAS, the SPRG value would be zero. Please refer to the enter_rtas()
// defined in entry.S
cmpwi cr0,r21,0; /* From user mode or RTAS? */\
bne 1f; /* Not RTAS, branch */\
//Yes. It is from user mode, then the stack will work on the corresponding kernel stack
// Before MMU enabled, we have to work on physical addresses.
tophys(r21, r1); /* Convert vka in r1 to pka in r21 */\
// Reserve the stack frame for handling this exception. INT_FRAME_SIZE is defined in
// /arch/ppc/kernel/mk_defs.c
subi r21,r21,INT_FRAME_SIZE; /* Allocate an exception frame */\
1: stw r20,_CCR(r21); /* Save CR on the stack */\
stw r22,GPR22(r21); /* Save r22 on the stack */\
stw r23,GPR23(r21); /* r23 Save on the stack */\
mfspr r20,SPRN_SPRG0; /* Get r20 back out of SPRG0 */\
stw r20,GPR20(r21); /* Save r20 on the stack */\
mfspr r22,SPRN_SPRG1; /* Get r21 back out of SPRG0 */\
```

```
stw r22,GPR21(r21); /* Save r21 on the stack */\
mflr r20; \
stw r20,_LINK(r21); /* Save LR on the stack */\
mfctr r22; \
stw r22,_CTR(r21); /* Save CTR on the stack */\
mfspr r20,XER; \
stw r20,_XER(r21); /* Save XER on the stack */
```

COMMON_EPILOG macro is used to save some GPRs including r0, r1 and r2 to the stack and then set up the new kernel stack pointer. Note that the r1 will contain the virtual address in preparation to the kernel exception handler. In other words, the kernel exception handler will work on virtual addresses after then.

```
#define COMMON_EPILOG \
stw r0,GPR0(r21); /* Save r0 on the stack */\
stw r1,GPR1(r21); /* Save r1 on the stack */\
stw r2,GPR2(r21); /* Save r2 on the stack */\
stw r1,0(r21); \
tovirt(r1,r21); /* Set-up new kernel stack pointer */\
SAVE_4GPRS(3, r21); /* Save r3 through r6 on the stack */\
SAVE_GPR(7, r21); /* Save r7 on the stack */

## Common exception code for standard (non-critical) exceptions.

#define STND_EXCEPTION_PROLOG \
COMMON_PROLOG; \
mfspr r22,SPRN_SRR0; /* Faulting instruction address */\
mfspr r23,SPRN_SRR1; /* MSR at the time of fault */\
COMMON_EPILOG;

## Common exception code for critical exceptions.

#define CRIT_EXCEPTION_PROLOG \
```

```
COMMON_PROLOG; \
mfspr r22,SPRN_SRR2; /* Faulting instruction address */\
mfspr r23,SPRN_SRR3; /* MSR at the time of fault */\
COMMON_EPILOG;
```

Transfer Exception to Exception Handler

After kernel finishes some common codes described above, it is ready to transfer the control to the exception handler, for example, do_page_fault(). This is done by the transfer_to_handler() function defined in head4xx.S.

```
###
### This code finishes saving the registers to the exception frame
### and jumps to the appropriate handler for the exception, turning
### on address translation.
###

_GLOBAL(transfer_to_handler)
// r22 and r23 contains the faulting instruction address and the msr value at the time of fault.
stw r22,_NIP(r21) # Save the faulting IP on the stack
stw r23,_MSR(r21) # Save the exception MSR on the stack
// We have saved r[0,7] already when in processing COMMON_EPILOG

SAVE_4GPRS(8, r21) # Save r8 through r11 on the stack
SAVE_8GPRS(12, r21) # Save r12 through r19 on the stack
SAVE_8GPRS(24, r21) # Save r24 through r31 on the stack
// Detect if it was in user space at time time of fault.
andi. r23,r23,MSR_PR # Is this from user space?
// SPRG3 contains the pointer to current task's thread_struct.
mfspr r23,SPRN_SPRG3 # If from user, fix up THREAD.regs
```

```
// if equal, it means that MSR_PR bit is clear, which means the exception happened when
in kernel space/mode.
beq 2f # No, it is from the kernel; branch.
// Exception from user space/mode,
addi r24,r1,STACK_FRAME_OVERHEAD

stw r24,PT_REGS(r23) #
2: addi r2,r23,-THREAD # Set r2 to current thread
tovirt(r2,r2)
mflr r23
andi. r24,r23,0x3f00 # Get vector offset
stw r24,TRAP(r21)
li r22,RESULT
stwcx. r22,r22,r21 # Clear the reservation
li r22,0
stw r22,RESULT(r21)
mtspr SPRN_SPRG2,r22 # r1 is now the kernel stack pointer
addi r24,r2,TASK_STRUCT_SIZE # Check for kernel stack overflow
cmplw cr0,r1,r2
cmplw cr1,r1,r24
crand cr1,cr1,cr4
bgt- stack_ovf # If r2 < r1 < r2 + TASK_STRUCT_SIZE
lwz r24,0(r23) # Virtual address of the handler
lwz r23,4(r23) # Handler return pointer
cmpwi cr0,r7,STND_EXC # What type of exception is this?
bne 3f # It is a critical exception...

## Standard exception jump path

mtspr SPRN_SRR0,r24 # Set up the instruction pointer
mtspr SPRN_SRR1,r20 # Set up the machine state register
```

```
mtlr r23 # Set up the return pointer
SYNC
rfi # Enable the MMU, jump to the handler

## Critical exception jump path

3: mtspr SPRN_SRR2,r24 # Set up the instruction pointer
mtspr SPRN_SRR3,r20 # Set up the machine state register
mtlr r23 # Set up the return pointer
SYNC
rfci # Enable the MMU, jump to the handler
```

Below gives an example for how to install Data Storage Exception handler.

```
### 0x0300 - Data Storage Exception
// This handler is located at 0x0300.
START_EXCEPTION(0x0300, DataAccess)
// Standard codes for non-critical exceptioins
STND_EXCEPTION_PROLOG
// Crab some information about the exception. ESR and DEAR contains the exception
reasons.
mfspr r5,SPRN_ESR # Grab the ESR, save it, pass as arg3
stw r5,_ESR(r21)
mfspr r4,SPRN_DEAR # Grab the DEAR, save it, pass as arg2
stw r4,_DEAR(r21)
addi r3,r1,STACK_FRAME_OVERHEAD
li r7,STND_EXC # This is a standard exception
li r20,MSR_KERNEL
rlwimi r20,r23,0,16,16 # Copy EE bit from the saved MSR
```

// Go to exception handler.
FINISH_EXCEPTION(do_page_fault) # do_page_fault(regs, ESR, DEAR)

11 Memory Management

We will discuss ppc-based linux kernel part related to memory management.
We will start from Data and Instruction TLB Miss.

For powerpc 405, please note that there is no page table needed for MMU manipulation, in contrast with x86. Kernel need explicitly read/write the TLB entries of ppc 405 MMU. More in-depth details will be given below.

When kernel reaches here, the regs is the pointer to stack area in which voltile registers and some SPR got saved; address is the value of DABR register for ppc 405; error_code is the value of register ESR for ppc 405 and DSISR for ppc 6xx or ppc 8xx.

/*
* This struct defines the way the registers are stored on the
* kernel stack during a system call or other kernel entry.
*
* this should only contain volatile regs
* since we can keep non-volatile in the thread_struct
* should set this up when only volatiles are saved
* by intr code.
*
* Since this is going on the stack, *CARE MUST BE TAKEN* to insure
* that the overall structure is a multiple of 16 bytes in length.
*
* Note that the offsets of the fields in this struct correspond with
* the PT_* values below. This simplifies arch/ppc/kernel/ptrace.c.
*/

```
#include

#ifndef __ASSEMBLY__
#ifdef CONFIG_PPC64BRIDGE
#define PPC_REG unsigned long /*long*/
#else
#define PPC_REG unsigned long
#endif
struct pt_regs {
PPC_REG gpr[32];
PPC_REG nip;
PPC_REG msr;
PPC_REG orig_gpr3; /* Used for restarting system calls */
PPC_REG ctr;
PPC_REG link;
PPC_REG xer;
PPC_REG ccr;
PPC_REG mq; /* 601 only (not used at present) */
/* Used on APUS to hold IPL value. */
PPC_REG trap; /* Reason for being here */
PPC_REG dar; /* Fault registers */
PPC_REG dsisr;
PPC_REG result; /* Result of a system call */
};
#endif
```

For Data TLB Miss and Instruction TLB Miss, ppc405 linux exception handler will convey the control to the PPC4xx_dtlb_miss and PPC4xx_itlb_miss, correspondingly.

0x1100 - Data TLB Miss Exception

STND_EXCEPTION(0x1100, DTLBMiss, PPC4xx_dtlb_miss)

0x1200 - Instruction TLB Miss Exception

STND_EXCEPTION(0x1200, ITLBMiss, PPC4xx_itlb_miss)

Let’s first explore the PPC4xx_dtlb_miss handler.
Firstly, we get the exception address by reading the ppc405 DEAR address. DEAR is set to the effective address of the failed access. We also then investigate the DST bit of ESR register to see if it is a write operation. And then we increase the data tlb miss count. After that done, we are ready to call the mainline of the TLB miss handler—tlbMiss(); If tlbMiss() returns 0, that means the tlb mapping for that badaddr is resident in TLB array and will then simply return from exception. If tlbMiss() returns a value of 1, that means we can’t find any page information in the page table for that badaddr, or that address is not with write permission and then we can’t simply load the TLB into TLB array. When reaching these cases, PPC4xx_dtlb_miss() will route its control to handle_page_fault() and then re-try loading the tlb entry by calling the tlbMiss() functioin again.

```
PPC4xx_dtlb_miss(struct pt_regs *regs)
{
// Retrieve the badaddr, which caused the data TLB miss exception.
unsigned long badaddr = mfspr(SPRN_DEAR);
// If this time is a write access?
int wasWrite = mfspr(SPRN_ESR) & ESR_DST;

dtlb_miss_count++;
// Try to fill out the TLB entry if possible. Return value 0:success; 1:failed
if (tlbMiss(regs, badaddr, wasWrite, 1)) {
//We can’t fill an TLB entry either because that no corresponding data in page table or it is a write access to a non-writtable page. Thus we have to do the do_page_fault()
```

```
// enable the interrupt
sti();
do_page_fault(regs, badaddr, wasWrite);
// clear the interrupt
cli();
tlbMiss(regs, badaddr, wasWrite, 0);
}
}



/*
* Mainline of the TLB miss handler. The above inline routines should fold into
* this one, eliminating most function call overhead.
*/

static inline int tlbMiss(struct pt_regs *regs, unsigned long badaddr, int wasWrite,
unsigned int count_increment)
{
int spid, ospid;
struct mm_struct *mm;
pgd_t *pgd;
pmd_t *pmd;
pte_t *pte;
// user_mode is a macro, defined as : user_mode(regs) ((regs)->msr & MSR_PR)
// if badaddr is NULL and the exception happened when in priviledged/kernel mode, or
badaddr is bigger than the 0xC0000000, we think this exception is from kernel part.
if ((badaddr == 0x0L && !user_mode(regs)) || (badaddr >= KERNELBASE)) {

ktlb_miss_count += count_increment;
// We will then work on the init_mm ‘cuase the exception is from kernel side. init_mm
```

provides all memory mapping information for the kernel space starting from 0xC0000000.

```
mm = &init_mm;
// Set up the pid as zero, this is required for ppc405 TLB handling.
spid = 0;
}
else {
// This exception is from user space of a user process
utlb_miss_count += count_increment;
// mm points to user process' mm data structure
mm = current->mm;
if (mm == NULL)
goto NOGOOD;
// Obtain the user process's pid
spid = mfspr(SPRN_PID);
}
```

// Before we can do the TLB miss handling, we need first detect if there is the corresponding mappings in this process's page table. Otherwise, we can do nothing but simply return with failure, which will go for do_page_fault() instead.

```
// if there is a corresponding entry from current process's root page table?
pgd = pgd_offset(mm, badaddr);
if (pgd_none(*pgd))
goto NOGOOD;
// if there is a corresponding entry from current process's second level page table?
// This is trivial for most CPUs. We all use two level page table structure.
pmd = pmd_offset(pgd, badaddr);
if (pmd_none(*pmd))
goto NOGOOD;
```

// if there is a corresponding entry from current process's page table? Or we say if there is a mapping for that particular page in which badaddr belongs to.

```
pte = pte_offset(pmd, badaddr);
if (pte_none(*pte))
```

```
goto NOGOOD;
// if this page was allocated before but currently is not resident in main memory, we still are not able to do anything.
if (!pte_present(*pte))
goto NOGOOD;
// We need see if a write access is allowable or not. If not, we get to do the do_page_fault, instead.
if (wasWrite) {
if (!pte_write(*pte)) {
goto NOGOOD;
}
// Writable and then we change the attributes of this page as dirty
set_pte(pte, pte_mkdirty(*pte));
}
// Update this page attribute to avoid being swapped out by kernel
set_pte(pte, pte_mkyoung(*pte));
// When reaching here, we are ok to do the TLB manipulating.
// We need save the current PID
ospid = mfspr(SPRN_PID);
// Set up the spid for the coreesponding badaddr.
mtspr(SPRN_PID, spid);
// Load an TLB entry along with the PID
mkTlbEntry(badaddr, pte);
//Restore the old pid.
mtspr(SPRN_PID, ospid);

return 0;

NOGOOD:
return 1;
}
```

The following gives the mkTlbEntry fuction, which is used to do the real work when a TLB miss happens.

```
static inline void
mkTlbEntry(unsigned long addr, pte_t *pte)
{
unsigned long ov = 0;
unsigned long tlbhi;
unsigned long tlblo;
int found = 1;
int idx;

/* ftr revisit
** Make sure the new entry doesn't overlap with an existing
** pinned entry
*/

/*
* Construct the TLB entry.
*/
// addr is the virtual/effective address. And after the AND operation, tlbhi will hold the value of the page--EPN
tlbhi = addr & ~(PAGE_SIZE-1);
// tlblo will then hold the RPN, the high order 22 bits of the page table entry.
tlblo = pte_val(*pte) & PAGE_MASK;

/* ftr revisit - test_mmap --> machine check w/o _PAGE_RW */
/* if (pte_val(*pte) & _PAGE_HWWRITE) */

// With the following codes, we will compose the TLB attributes/tags by using those in its
```

```
PTE attributes.
.// if read and write allowed.
if (pte_val(*pte) & (_PAGE_HWWRITE | _PAGE_RW))
tlblo |= TLB_WR;
// if no-cacheable
if (pte_val(*pte) & _PAGE_NO_CACHE)
tlblo |= TLB_I;
// if guarded
if (pte_val(*pte) & _PAGE_GUARDED)
tlblo |= TLB_G;

tlblo |= TLB_EX; /* ftr revisit: why _always_ execute? */
// if this page is not kernel address, we also add the zone protection.
if (addr < KERNELBASE) /* ftr - revisit */
/* tlblo |= TLBLO_Z_USER; */
tlblo |= TLB_ZSEL(1);

/* ftr revisit - need vassert() */
#if DEBUG
if (PAGE_SIZE != 4096)
panic("mkTlbEntry() - PAGE_SIZE hardcoded to 4K");
#endif
// Make sure we are using 4K page size
tlbhi |= TLB_PAGESZ(PAGESZ_4K);
// Set this tlb with valid attribute
tlbhi |= TLB_VALID;

/*
* See if a match already exists in the TLB.
*/
asm("tlbsx. %0,0,%2;beq 1f;li %1,0;1:" : "=r" (idx), "=r" (found) : "r" (tlbhi));
```

```
if (found) {
/*
* Found an existing entry. Just reuse the index.
*
* insert tag as invalid, insert new data, insert new tag
*/
asm volatile("tlbwe %0,%1,0" : : "r" (ov), "r" (idx));
asm volatile("tlbwe %0,%1,1" : : "r" (tlblo), "r" (idx));
asm volatile("tlbwe %0,%1,0" : : "r" (tlbhi), "r" (idx));
}
else {
/*
* Do the more expensive operation
*/
tlbDropin(tlbhi, tlblo);
}
}

/*
* TLB miss handling code.
*/
/*
* Handle TLB faults. We should push this back to assembly code eventually.
* Caller is responsible for turning off interrupts ...
*/
static inline void
tlbDropin(unsigned long tlbhi, unsigned long tlblo)
{
unsigned long ov = 0;
//Find the first available tlb entry from the global pin_table data structure
while (pin_table[tlb_next_replace].e_pinned) {
```

```
tlb_next_replace++;
if (tlb_next_replace >= PPC4XX_TLB_SIZE)
tlb_next_replace = 0;
}

/* insert tag as invalid, insert new data, insert new tag */
asm volatile("tlbwe %0,%1,0" : : "r" (ov), "r" (tlb_next_replace));
asm volatile("tlbwe %0,%1,1" : : "r" (tlblo), "r" (tlb_next_replace));
asm volatile("tlbwe %0,%1,0" : : "r" (tlbhi), "r" (tlb_next_replace));
asm volatile("isync;sync");

tlb_next_replace++;
if (tlb_next_replace >= PPC4XX_TLB_SIZE)
tlb_next_replace = 0;
}
```

Some ppc405 tlb supporting data structures and functions:

pin_table is defined as a global structure to record the status of a ppc4xxx tlb.

```
/* Type Definitions */

typedef struct pin_entry_s {
unsigned int e_pinned: 1, /* This TLB entry is pinned down. */
e_used: 23; /* Number of users for this mapping. */
} pin_entry_t;

/* Global Variables */
/* Record all ppc4xx tlb entries usage */
static pin_entry_t pin_table[PPC4XX_TLB_SIZE];
/*Record the next available tlb entry */
```

```
static unsigned long tlb_next_replace = 0;

Linux kernel will detect if a tlb entry has been used or not via this below approach:
if (pin_table.e_pinned == 0) {
tlb entry I is already used;
} else{

tlb entry i still available;
}

void
PPC4xx_tlb_pin(unsigned long va, unsigned long pa, int pagesz, int cache)
{
int i, found = FALSE;
unsigned long tag, data;
unsigned long opid;
unsigned long ov = 0;

/* ftr revisit
** - check that entry doesn't already exist in the TLB
** - check that overlapping entry doesn't exist in the TLB
** (the quick & dirty way for now is to flush the TLB)
**
** remove function prototype when PPC4xx_tlb_flush_all() is
** removed from here
*/

/* don't do this until the kernel is pinned */
if (pin_table[0].e_pinned)
PPC4xx_tlb_flush_all();
```

```
opid = mfspr(SPRN_PID);
mtspr(SPRN_PID, 0);
asm volatile("sync");

data = (pa & TLB_RPN_MASK) | TLB_WR;

if (cache)
data |= (TLB_EX);
else
data |= (TLB_G | TLB_I);

tag = (va & TLB_EPN_MASK) | TLB_VALID | pagesz;

for (i = 0; i < PPC4XX_TLB_SIZE; i++) {
// Find the first tlb entry
if (pin_table.e_pinned == 0) {
found = TRUE;
break;
}
}

if (found) {
/* insert tag as invalid, insert new data, insert new tag */
asm volatile("tlbwe %0,%1,0" : : "r" (ov), "r" (i));
asm volatile("tlbwe %0,%1,1" : : "r" (data), "r" (i));
asm volatile("tlbwe %0,%1,0" : : "r" (tag), "r" (i));
asm volatile("isync");
pin_table.e_pinned = 1;
pin_table.e_used++;
}
```

```
mtspr(SPRN_PID, opid);
asm volatile("sync");

return;
}


void
PPC4xx_tlb_flush_all(void)
{
int i;
unsigned long flags;
unsigned long ov = 0;

save_flags(flags);
cli();

for (i = 0; i < PPC4XX_TLB_SIZE; i++) {

if (pin_table.e_pinned)
continue;

/* unset valid bit */
asm volatile("tlbwe %0,%1,0" : : "r" (ov), "r" (i));
}

asm volatile("sync;isync");

restore_flags(flags);
}
```

```
void
PPC4xx_tlb_flush(unsigned long va, int pid)
{
unsigned long i, tag, flags, found = 1, opid;

save_flags(flags);
cli();

opid = mfspr(SPRN_PID);
mtspr(SPRN_PID, pid);

asm("tlbsx. %0,0,%2;"
"beq 1f;"
"li %1,0;"
"1:"
: "=r" (i), "=r" (found) : "r" (va));

if (found && pin_table.e_pinned == 0) {
asm("tlbre %0,%1,0" : "=r" (tag) : "r" (i));
tag &= ~ TLB_VALID;
asm("tlbwe %0,%1,0" : : "r" (tag), "r" (i));
}

mtspr(SPRN_PID, opid);

restore_flags(flags);
}
```

When a PPC 405 Data Storage Exception happens, the kernel will catch and transfer it to

do_page_fault. This is defined in ./arch/ppc/kernel/head_4xx.S

0x0300 - Data Storage Exception

START_EXCEPTION(0x0300, DataAccess)
STND_EXCEPTION_PROLOG(0x0300)
mfspr r5,SPRN_ESR # Grab the ESR, save it, pass as arg3
stw r5,_ESR(r21)
mfspr r4,SPRN_DEAR # Grab the DEAR, save it, pass as arg2
stw r4,_DEAR(r21)
addi r3,r1,STACK_FRAME_OVERHEAD
li r7,STND_EXC # This is a standard exception
li r20,MSR_KERNEL
rlwimi r20,r23,0,16,16 # Copy EE bit from the saved MSR
FINISH_EXCEPTION(do_page_fault) # do_page_fault(regs, ESR, DEAR)

From the definition of FINISH_EXCEPTION, we can easily find that the exception handler will eventually call the “func” function.
#define FINISH_EXCEPTION(func) \
bl transfer_to_handler; \
.long func; \
.long ret_from_except

First we get the current mm context by *mm = current->mm.
And then, we detect if the cause of this page fault is because of a writing operation.
#if defined(CONFIG_4xx)
int is_write = error_code & ESR_DST;
Below is the definition for ppc405 when a data storage exception happens. ESR register

bits will be set corresponding to different causes.

SRR0 Written with the EA of the instruction causing the data storage interrupt
SRR1 Written with the value of the MSR at the time of the interrupt
MSR WE, EE, PR, DWE, IR, DR0
CE, ME, DE unchanged
PC EVPR[0:15] || 0x0300
DEAR Written with the EA of the failed access
ESR DST 1 if excepting operation is a store (includes dcbi and dcbz)
DIZ 1 if access failure caused by a zone protection fault (ZPR[Z n]=00in
user mode)
U0F 1 if access failure caused by a U0 fault (the U0 storage attribute is
set and CCR0[U0XE] = 1)
MCI unchanged
All other bits are cleared.

For ppc6xx or 800-family processors , an 32-bit DSISR is used to identify the cause of DSI and alignment exceptions. The bits definition of DSISR when a Data storage exception happens is described below:

DSISR 0 Set if a load or store instruction results in a direct-store error exception; otherwise cleared.
Note: The direct-store facility is being phased out of the architecture and is not likely to be
supported in future devices.
1 Set if the translation of an attempted access is not found in the primary hash table entry group
(HTEG), or in the rehashed secondary HTEG, or in the range of a DBAT register (page fault
condition); otherwise cleared.
2 – 3 Cleared

4 Set if a memory access is not permitted by the page or DBAT protection mechanism; otherwise
cleared.
5 Set if the eciwx, ecowx, lwarx, or stwcx. , instruction is attempted to direct-store interface
space, or if the lwarx or stwcxinstruction is used with addresses that are marked as write-through.
Otherwise cleared to 0.
Note: The direct-store facility is being phased out of the architecture and is not likely to be
supported in future devices.
6 Set for a store operation and cleared for a load operation.
7 – 8 Cleared
9 Set if a DABR match occurs. Otherwise cleared.
10 Cleared
11 Set if the instruction is an eciwx or ecowx and EAR[E] = 0; otherwise cleared.
12 – 31 Cleared
Due to the multiple exception conditions possible from the execution of a single instruction, the
following combinations of bits of DSISR may be set concurrently:

- Bits 1 and 11
- Bits 4 and 5
- Bits 4 and 11
- Bits 5 and 11

Additionally, bit 6 is set if the instruction that caused the exception is a store, ecowx, dcbz, dcba, or
dcbi and bit 6 would otherwise be cleared. Also, bit 9 (DABR match) may be set alone, or in
combination with any other bit, or with any of the other combinations shown above.

From the above, we can find that the bit 6 will be set if the current exception is a write operation. Therefore, we use int is_write = error_code & 0x02000000 to get the is_write variable value.

We then detect if the system, when this page fault happened, is in interrupt level or not. Also, see whether or not the current mm context is NULL. If either one is true, we go to bad_page_fault.

We will then get the lock of the current mm and try to find the corresponding vma by going through the current mm's vma linked list to see if there is a match for that cause address.

For find_vma() function, it is a machine independent rountine. Please refer to any linux kernel books for its behavior. We will only address those parts that are ppc specific.

In short, if the returned vma is NULL, that means we could not find any vma area related to this cause address and transfer control to bad area processing; if the cause address happens to fall inside this found vma(vma->vm_start <= address), we are happy and then go to good_area processing; if the vma->vm_start > address and moreover this vma's attributes are not allowed to GROWSDOWN, (if is not the stack area), we have to go for bad area processing; Otherwise, this vma is the stack area and then we expand the vma/stack. Note that vma is the first one with address < vma->vm_end, and even address < vma->vm_start. We hve to extend vma.

If this exception is from a write operation, kernel will then see if this vma area is opened for writing. If not prohibited for writing, we will go for bad_area processing. This is done by

```
/* a write */
if (is_write) {
if (!(vma->vm_flags & VM_WRITE))
goto bad_area;
```

For a read operation exception, we wil first see if this exception is because the protection. This can be detected by watching the bit of ESR for ppc4xx or the bit of DSISR for ppc6xx and ppc8xx. If a protection error, we go to bad_area processing. Also, if this VMA area is not allowed for reading and executing, we also go to bad_area processing.

Up to know, we have done all the pre-processing work for this page fault. Now, start to do the real work—handle_mm_fault(), which is defined in linux/mm/memory.c. Please note that by the time we get here, we already hold the mm semaphore.

handle_mm_fault():

We first get the root page directory for this exception address by using pgd_offset, which is defined as follows:

```
/* PMD_SHIFT determines the size of the area mapped by the second-level page tables
*/
#define PMD_SHIFT 22
#define PMD_SIZE (1UL << PMD_SHIFT)
#define PMD_MASK (~(PMD_SIZE-1))

/* PGDIR_SHIFT determines what a third-level page table entry can map */
#define PGDIR_SHIFT 22
#define PGDIR_SIZE (1UL << PGDIR_SHIFT)
#define PGDIR_MASK (~(PGDIR_SIZE-1))

/* to find an entry in a page-table-directory */
#define pgd_index(address) ((address) >> PGDIR_SHIFT)
#define pgd_offset(mm, address) ((mm)->pgd + pgd_index(address))
```

After we obtain the entry in the root page directory, we can easily get the entry of the middle page directory. For most of CPU platforms, linux actually only have two level

page tables and thus simply do nothing but return the root page table directory entry back.
pmd = pmd_alloc(pgd, address);

Now kernel starts to grab the page table content from the page table by using the pte_alloc().

```
/* PAGE_SHIFT determines the page size */
#define PAGE_SHIFT 12
#define PAGE_SIZE (1UL << PAGE_SHIFT)
#define PAGE_MASK (~(PAGE_SIZE-1))
/*
* entries per page directory level: our page-table tree is two-level, so
* we don't really have any PMD directory.
*/
#define PTRS_PER_PTE 1024
#define PTRS_PER_PMD 1
#define PTRS_PER_PGD 1024
#define USER_PTRS_PER_PGD (TASK_SIZE / PGDIR_SIZE)
#define FIRST_USER_PGD_NR 0

extern inline pte_t * pte_alloc(pmd_t * pmd, unsigned long address)
{
// Get the offset in the page table. PAGE_SHIFT is 12, which mean
// the page size is 4k. We first shift right 12 bits and then use the
// PTRS_PER_PTR to mask the last 10 bits. So that the result address will
// only hold the value of the middle 10 bits of the original addresss value.
// And then we get the offset for the page table.

address = (address >> PAGE_SHIFT) & (PTRS_PER_PTE - 1);
// *pmd is the content of the target page table entry in the second level page
// directory. If it is NULL, that means
```

```
// this page table entry is not allocated or mapped yet.
if (pmd_none(*pmd)) {
// If no corresponding page table entry yet, kernel will start to allocate a new page
// to hold 1024 entries for the page table pointed by pmd.
// Allocate a new page from kernel pte cache list for holding the ptes
pte_t * page = (pte_t *) get_pte_fast();

if (!page)
// Allocate a new page from kernel memory for holding the ptes
return get_pte_slow(pmd, address);
// Fill out the pmd entry in the middle level page directory
pmd_val(*pmd) = (unsigned long) page;
// return the leaf page table entry in the new allocated page table.
return page + address;
}
if (pmd_bad(*pmd)) {
__bad_pte(pmd);
return NULL;
}
// Return the value of this leaf page table entry
// address is the offset for the page table pointe by pmd.
return (pte_t *) pmd_page(*pmd) + address;
}
```

When kernel back from pte_alloc(), kernel already got the leaf page table entry content and then will start to see why this page fault exception happened by invoking handle_pte_fault(), which id defined as handle_pte_fault(mm, vma, address, write_access, pte). This function is also platform independent. It will first check if this pte is present or not in memory.

```
entry = *pte;
// Check if this pte entry contains valid pte information
```

```
if (!pte_present(entry)) {
// Or this entry is not mapped yet with zero value?
if (pte_none(entry))
// Try to map this effective/virtual addresses
return do_no_page(mm, vma, address, write_access, pte);
// That means this page was swapped out by kernel and was set to be
// invalid state.
// Kernel will then swap it back.
return do_swap_page(mm, vma, address, pte, pte_to_swp_entry(entry), write_access);
}
```

Now let’s investigate how the kernel allocates a new page frame for this page.

```
/*
* do_no_page() tries to create a new page mapping. It aggressively
* tries to share with existing pages, but makes a separate copy if
* the "write_access" parameter is true in order to avoid the next
* page fault.
*
* As this is called only for pages that do not currently exist, we
* do not need to flush old virtual caches or the TLB.
*
* This is called with the MM semaphore held.
*/
static int do_no_page(struct mm_struct * mm, struct vm_area_struct * vma,
unsigned long address, int write_access, pte_t *page_table)
{
struct page * new_page;
pte_t entry;

if (!vma->vm_ops || !vma->vm_ops->nopage)
return do_anonymous_page(mm, vma, page_table, write_access, address);
```

```
/*
* The third argument is "no_share", which tells the low-level code
* to copy, not share the page even if sharing is possible. It's
* essentially an early COW detection.
*/
new_page = vma->vm_ops->nopage(vma, address & PAGE_MASK, (vma->vm_flags &
VM_SHARED)?0:write_access);
if (new_page == NULL) /* no page was available -- SIGBUS */
return 0;
if (new_page == NOPAGE_OOM)
return -1;
++mm->rss;
/*
* This silly early PAGE_DIRTY setting removes a race
* due to the bad i386 page protection. But it's valid
* for other architectures too.
*
* Note that if write_access is true, we either now have
* an exclusive copy of the page, or this is a shared mapping,
* so we can make it writable and dirty to avoid having to
* handle that later.
*/
flush_page_to_ram(new_page);
flush_icache_page(vma, new_page);
entry = mk_pte(new_page, vma->vm_page_prot);
if (write_access) {
entry = pte_mkwrite(pte_mkdirty(entry));
} else if (page_count(new_page) > 1 &&
!(vma->vm_flags & VM_SHARED))
entry = pte_wrprotect(entry);
```

```
set_pte(page_table, entry);
/* no need to invalidate: a not-present page shouldn't be cached */
update_mmu_cache(vma, address, entry);
return 2; /* Major fault */
}

/*
* Establish a new mapping:
* - flush the old one
* - update the page tables
* - inform the TLB about the new one
*/
static inline void establish_pte(struct vm_area_struct * vma, unsigned long address, pte_t
*page_table, pte_t entry)
{
flush_tlb_page(vma, address);
set_pte(page_table, entry);
update_mmu_cache(vma, address, entry);
}
```

12 Process Management

This chapter will explain how ppc4xx linux kernel finish its context switch. We will only focus on those codes that are CPU dependent, more specifically, the ppc4xx dependent.

The fundemental, maybe also the most important, function for process scheduling is the _switch, which is defined in arch/ppc/kernel/entry.S. Kernel, based on the switch, implement the other scheduling functions.
Kernel invokes this switch rountine by providing the old and new THREAD pointers that belong to two different processes. For every process in linux, there is a threadstruct defined to hold a process’s CPU related states.

_switch rountine will save all non-volitle information of the old thread into the corresponding thread data structure;load the next thread’s state information to the CPU context and then make a switch or we say, context switch/scheduling. Illustrated below are the codes of the _switch,

_switch(old_thread, new_thread);

```
/*
* This routine switches between two different tasks. The process
* state of one is saved on its kernel stack. Then the state
* of the other is restored from its kernel stack. The memory
* management hardware is updated to the second process's state.
* Finally, we can return to the second process, via ret_from_except.
* On entry, r3 points to the THREAD for the current task, r4
* points to the THREAD for the new task.
*
* Note: there are two ways to get to the "going out" portion
* of this code; either by coming in via the entry (_switch)
* or via "fork" which must set up an environment equivalent
* to the "_switch" path. If you change this (or in particular, the
* SAVE_REGS macro), you'll have to change the fork code also.
*
* The code which creates the new task context is in 'copy_thread'
* in arch/ppc/kernel/process.c
*/
_GLOBAL(_switch)
// Set up a stack frame for saving context information
stwu r1,-INT_FRAME_SIZE(r1)
// Store the r0
stw r0,GPR0(r1)
```

```
// Store the original r1 value. We are using EABI convention for the stack usage
// and thus 0(r1) contains the original stack pointer
lwz r0,0(r1)
stw r0,GPR1(r1)
// Note that we don’t have to save r3 to r13 into the THREAD data structure as
// part of the context information. For powerpc register profile-EABI, r3 to r13 are
// defined as temporary usage.
/* r3-r13 are caller saved -- Cort */

//Store r2
SAVE_GPR(2, r1)
//Store r14 to r21
SAVE_8GPRS(14, r1)
//Store r22 to r31
SAVE_10GPRS(22, r1)
// We start to save special registers
// lr and msr are important information we have to protected during context switch
mflr r20 /* Return to switch caller */
mfmsr r22
/* MVISTA_LOCAL - begin nomerge */
#if !defined(CONFIG_IBM405) /* ftr revisit - kgdb */
/* MVISTA_LOCAL - end nomerge */
li r0,MSR_FP /* Disable floating-point */
/* MVISTA_LOCAL - begin nomerge */
#endif
/* MVISTA_LOCAL - end nomerge */
#ifdef CONFIG_ALTIVEC
oris r0,r0,MSR_VEC@h
#endif /* CONFIG_ALTIVEC */
/* MVISTA_LOCAL - begin nomerge */
#if !defined(CONFIG_IBM405) /* ftr revisit - kgdb */
```

```
/* MVISTA_LOCAL - end nomerge */
andc r22,r22,r0
/* MVISTA_LOCAL - begin nomerge */
#endif
/* MVISTA_LOCAL - end nomerge */
stw r20,_NIP(r1)
stw r22,_MSR(r1)
// NIP is equal to the LINK, the position after the _switch() function
stw r20,_LINK(r1)
//Save the cr, ctr and xer
mfcr r20
mfctr r22
mfspr r23,XER
stw r20,_CCR(r1)
stw r22,_CTR(r1)
stw r23,_XER(r1)

li r0,0x0ff0
stw r0,TRAP(r1)
//Save the old stack pointer in THREAD data structure
stw r1,KSP(r3) /* Set old stack pointer */
sync
//Get the physical address of the next THREAD data structure
tophys(r0,r4)
CLR_TOP32(r0)
mtspr SPRG3,r0 /* Update current THREAD phys addr */
#ifdef CONFIG_8xx
/* XXX it would be nice to find a SPRGx for this on 6xx,7xx too */
lwz r9,PGDIR(r4) /* cache the page table root */
tophys(r9,r9) /* convert to phys addr */
mtspr M_TWB,r9 /* Update MMU base address */
```

```
tlbia
SYNC
#endif /* CONFIG_8xx */
lwz r1,KSP(r4) /* Load new stack pointer */
/* save the old current 'last' for return value */
mr r3,r2
addi r2,r4,-THREAD /* Update current */
lwz r9,_MSR(r1) /* Returning to user mode? */
andi. r9,r9,MSR_PR
beq+ 10f /* if not, don't adjust kernel stack */
8: addi r4,r1,INT_FRAME_SIZE /* size of frame */
stw r4,THREAD+KSP(r2) /* save kernel stack pointer */
tophys(r9,r1)
CLR_TOP32(r9)
mtspr SPRG2,r9 /* phys exception stack pointer */
// Load the context of the next process
10: lwz r2,_CTR(r1)
lwz r0,_LINK(r1)
mtctr r2
mtlr r0
lwz r2,_XER(r1)
lwz r0,_CCR(r1)
mtspr XER,r2
mtcrf 0xFF,r0
/* r3-r13 are destroyed -- Cort */
REST_GPR(14, r1)
REST_8GPRS(15, r1)
REST_8GPRS(23, r1)
REST_GPR(31, r1)
lwz r2,_NIP(r1) /* Restore environment */
/*
```

```
* We need to hard disable here even if RTL is active since
* being interrupted after here trashes SRR{0,1}
* -- Cort
*/
// We need disable interrupt to avoid the srr0 and srr1 being overwritten
mfmsr r0 /* Get current interrupt state */
rlwinm r0,r0,0,17,15 /* clear MSR_EE in r0 */
mtmsr r0 /* Update machine state */

lwz r0,_MSR(r1)
mtspr SRR0,r2
FIX_SRR1(r0,r2)
mtspr SRR1,r0
// Bring the original r0, r1 and r2 back.
lwz r0,GPR0(r1)
lwz r2,GPR2(r1)
lwz r1,GPR1(r1)
SYNC
// Use the RFI instruction to make a safely context switch
RFI
```

_switch_to():

_switchto() function is the one who will invoke the _switch assemble rountine.

```
void
_switch_to(struct task_struct *prev, struct task_struct *new,
struct task_struct **last)
{
struct thread_struct *new_thread, *old_thread;
```

```
unsigned long s;
//Save flags
__save_flags(s);
// Disable the interrupt
__cli();
#if CHECK_STACK
check_stack(prev);
check_stack(new);
#endif

#ifdef SHOW_TASK_SWITCHES
printk("%s/%d -> %s/%d NIP %08lx cpu %d root %x/%x\n",
prev->comm,prev->pid,
new->comm,new->pid,new->thread.regs->nip,new->processor,
new->fs->root,prev->fs->root);
#endif
#ifdef CONFIG_SMP
/* avoid complexity of lazy save/restore of fpu
* by just saving it every time we switch out if
* this task used the fpu during the last quantum.
*
* If it tries to use the fpu again, it'll trap and
* reload its fp regs. So we don't have to do a restore
* every switch, just a save.
* -- Cort
*/
if ( prev->thread.regs && (prev->thread.regs->msr & MSR_FP) )
giveup_fpu(prev);
#ifdef CONFIG_ALTIVEC
/*
* If the previous thread 1) has some altivec regs it wants saved
```

```
* (has bits in vrsave set) and 2) used altivec in the last quantum
* (thus changing altivec regs) then save them.
*
* On SMP we always save/restore altivec regs just to avoid the
* complexity of changing processors.
* -- Cort
*/
if ( (prev->thread.regs && (prev->thread.regs->msr & MSR_VEC)) &&
prev->thread.vrsave )
giveup_altivec(prev);
#endif /* CONFIG_ALTIVEC */
prev->last_processor = prev->processor;
current_set[smp_processor_id()] = new;
#endif /* CONFIG_SMP */
/* Avoid the trap. On smp this this never happens since
* we don't set last_task_used_altivec -- Cort
*/
/* MVISTA_LOCAL - begin nomerge */
#if !defined(CONFIG_IBM405) /* ftr revisit - kgdb */
/* MVISTA_LOCAL - end nomerge */
if ( last_task_used_altivec == new )
new->thread.regs->msr |= MSR_VEC;
/* MVISTA_LOCAL - begin nomerge */
#endif
/* MVISTA_LOCAL - end nomerge */
new_thread = &new->thread;
old_thread = ☐t->thread;
*last = _switch(old_thread, new_thread);
__restore_flags(s);
}
```

Switch_to()

```
#define switch_to(prev,next,last) _switch_to((prev),(next),&(last))

/*
* 'schedule()' is the scheduler function. It's a very simple and nice
* scheduler: it's not perfect, but certainly works for most things.
*
* The goto is "interesting".
*
* NOTE!! Task 0 is the 'idle' task, which gets called when no other
* tasks can run. It can not be killed, and it cannot sleep. The 'state'
* information in task[0] is never used.
*/
asmlinkage void schedule(void)
{
struct schedule_data * sched_data;
struct task_struct *prev, *next, *p;
struct list_head *tmp;
int this_cpu, c;

// It is illegal that the current process has no valid active_mm data structure
if (!current->active_mm) BUG();
// See if we have to finish some left work before doing the real scheduling.
if (tq_scheduler)
goto handle_tq_scheduler;
tq_scheduler_back:

prev = current;
this_cpu = prev->processor;
// We are in interrupt handler?
```

```
if (in_interrupt())
goto scheduling_in_interrupt;

release_kernel_lock(prev, this_cpu);

/* Do "administrative" work here while we don't hold any locks */
if (softirq_state[this_cpu].active & softirq_state[this_cpu].mask)
goto handle_softirq;
handle_softirq_back:

/*
* 'sched_data' is protected by the fact that we can run
* only one process per CPU.
*/
sched_data = & aligned_data[this_cpu].schedule_data;

spin_lock_irq(&runqueue_lock);

/* move an exhausted RR process to be last.. */
if (prev->policy == SCHED_RR)
goto move_rr_last;
move_rr_back:

switch (prev->state & ~TASK_EXCLUSIVE) {
case TASK_INTERRUPTIBLE:
if (signal_pending(prev)) {
prev->state = TASK_RUNNING;
break;
}
default:
del_from_runqueue(prev);
```

```
case TASK_RUNNING:
}
prev->need_resched = 0;

/*
* this is the scheduler proper:
*/

repeat_schedule:
/*
* Default process to select..
*/
next = idle_task(this_cpu);
c = -1000;
if (prev->state == TASK_RUNNING)
goto still_running;

still_running_back:
list_for_each(tmp, &runqueue_head) {
p = list_entry(tmp, struct task_struct, run_list);
if (can_schedule(p)) {
int weight = goodness(p, this_cpu, prev->active_mm);
if (weight > c)
c = weight, next = p;
}
}

/* Do we need to re-calculate counters? */
if (!c)
goto recalculate;
/*
```

```
* from this point on nothing can prevent us from
* switching to the next task, save this fact in
* sched_data.
*/
sched_data->curr = next;
#ifdef CONFIG_SMP
next->has_cpu = 1;
next->processor = this_cpu;
#endif
spin_unlock_irq(&runqueue_lock);

if (prev == next)
goto same_process;

#ifdef CONFIG_SMP
/*
* maintain the per-process 'average timeslice' value.
* (this has to be recalculated even if we reschedule to
* the same process) Currently this is only used on SMP,
* and it's approximate, so we do not have to maintain
* it while holding the runqueue spinlock.
*/
{
cycles_t t, this_slice;

t = get_cycles();
this_slice = t - sched_data->last_schedule;
sched_data->last_schedule = t;

/*
* Exponentially fading average calculation, with
```

```
* some weight so it doesnt get fooled easily by
* smaller irregularities.
*/
prev->avg_slice = (this_slice*1 + prev->avg_slice*1)/2;
}

/*
* We drop the scheduler lock early (it's a global spinlock),
* thus we have to lock the previous process from getting
* rescheduled during switch_to().
*/

#endif /* CONFIG_SMP */

kstat.context_swtch++;
/*
* there are 3 processes which are affected by a context switch:
*
* prev == .... ==> (last => next)
*
* It's the 'much more previous' 'prev' that is on next's stack,
* but prev is set to (the just run) 'last' process by switch_to().
* This might sound slightly confusing but makes tons of sense.
*/
prepare_to_switch();
{

struct mm_struct *mm = next->mm;
struct mm_struct *oldmm = prev->active_mm;
if (!mm) { // if this ready task’s MM is NULL, that means that it is a lazy
// MMU process.
```

```
//A lazy MMU task’s active MM must not be NULL.
if (next->active_mm) BUG();
//Simply switch the active_mm to the current one so as to avoid
//TLB flushing and other cost.
next->active_mm = oldmm;
// We need record this in case mm structure will not be released by
//accident.
atomic_inc(&oldmm->mm_count);
// Enter lazy tlb mode. For PPC, this functioin does nothing.
enter_lazy_tlb(oldmm, next, this_cpu);
} else {
// If a task’s MM is NOT NULL, then its active MM must be
//equal to its MM.
if (next->active_mm != mm) BUG();
// Switch the MM context.
switch_mm(oldmm, mm, next, this_cpu);
}
// If the prev is also a lazy tlb task, let’s reset its active_mm pointer
if (!prev->mm) {
prev->active_mm = NULL;
// see if the old mm should be released
mmdrop(oldmm);
}
}

/*
* This just switches the register state and the
* stack.
*/
switch_to(prev, next, prev);
__schedule_tail(prev);
```

```
same_process:
reacquire_kernel_lock(current);
return;

recalculate:
{
struct task_struct *p;
spin_unlock_irq(&runqueue_lock);
read_lock(&tasklist_lock);
for_each_task(p)
p->counter = (p->counter >> 1) + p->priority;
read_unlock(&tasklist_lock);
spin_lock_irq(&runqueue_lock);
}
goto repeat_schedule;

still_running:
c = prev_goodness(prev, this_cpu, prev->active_mm);
next = prev;
goto still_running_back;

handle_softirq:
do_softirq();
goto handle_softirq_back;

handle_tq_scheduler:
/*
* do not run the task queue with disabled interrupts,
* cli() wouldn't work on SMP
*/
```

```
sti();
run_task_queue(&tq_scheduler);
goto tq_scheduler_back;

move_rr_last:
if (!prev->counter) {
prev->counter = prev->priority;
move_last_runqueue(prev);
}
goto move_rr_back;

scheduling_in_interrupt:
printk("Scheduling in interrupt\n");
BUG();
return;
}
```

13 Interrupt Handling routines

For each CPU, there are some corresponding interrupt handling rountines, for example, the cli, sti, __save_flags and __restore_flags.

For powerpc based kernel, system uses a global data structure called int_control_struct to maintain the function pointers. This is very similar to that of usage when we learn the ppc_md data structure, which is used for holding all machine depenedent driver function pointers.

Below is the defintion of int_control_struct, in which, the cli, sti, restore_flags, save_flags and set_lost function pointers are get defined. This part of code can be found at ./linux/include/asm-ppc/hw_irq.h

```
struct int_control_struct
{
void (*int_cli)(void);
void (*int_sti)(void);
void (*int_restore_flags)(unsigned long);
void (*int_save_flags)(unsigned long *);
void (*int_set_lost)(unsigned long);
};
extern struct int_control_struct int_control;
extern void __no_use_sti(void);
extern void __no_use_cli(void);
extern void __no_use_restore_flags(unsigned long);
extern void __no_use_save_flags(unsigned long *);
extern void __no_use_set_lost(unsigned long);

#define __cli() int_control.int_cli()
#define __sti() int_control.int_sti()
#define __save_flags(flags) int_control.int_save_flags(&flags)
#define __restore_flags(flags) int_control.int_restore_flags(flags)
#define __save_and_cli(flags) ({__save_flags(flags);__cli();})
#define __set_lost(irq) ({ if ((ulong)int_control.int_set_lost)
int_control.int_set_lost(irq); })
```

Linux ppc kernel defines a int_control variable in ./arch/ppc/kernel/setup.c

```
struct int_control_struct int_control =
{
__no_use_cli,
__no_use_sti,
```

```
__no_use_restore_flags,
__no_use_save_flags
};
```

The above 4 funciton pointers are defined in ./arch/ppc/kernel/misc.S, in which lots of ppc supporting assemble routines are implemented.

```
/* void __no_use_save_flags(unsigned long *flags) */
_GLOBAL(__no_use_save_flags)
//get msr
mfmsr r4
//save the value
stw r4,0(r3)
blr

/* void __no_use_restore_flags(unsigned long flags) */
_GLOBAL(__no_use_restore_flags)
/*
* Just set/clear the MSR_EE bit through restore/flags but do not
* change anything else. This is needed by the RT system and makes
* sense anyway.
* -- Cort
*/
mfmsr r4
/* Copy all except the MSR_EE bit from r4 (current MSR value)
to r3. This is the sort of thing the rlwimi instruction is
designed for. -- paulus. */
rlwimi r3,r4,0,17,15
/* Check if things are setup the way we want _already_. */
cmpw 0,r3,r4
beqlr
```

```
/* are we enabling interrupts? */
rlwinm. r0,r3,0,16,16
beq 1f
/* if so, check if there are any lost interrupts */
lis r7,ppc_n_lost_interrupts@ha
lwz r7,ppc_n_lost_interrupts@l(r7)
cmpi 0,r7,0 /* lost interrupts to process first? */
bne- do_lost_interrupts
1: sync
mtmsr r3
isync
blr

_GLOBAL(__no_use_cli)
mfmsr r0 /* Get current interrupt state */
rlwinm r3,r0,16+1,32-1,31 /* Extract old value of 'EE' */
rlwinm r0,r0,0,17,15 /* clear MSR_EE in r0 */
sync /* Some chip revs have problems here... */
mtmsr r0 /* Update machine state */
blr /* Done */

_GLOBAL(__no_use_sti)
lis r4,ppc_n_lost_interrupts@ha
lwz r4,ppc_n_lost_interrupts@l(r4)
mfmsr r3 /* Get current state */
ori r3,r3,MSR_EE /* Turn on 'EE' bit */
cmpi 0,r4,0 /* lost interrupts to process first? */
bne- do_lost_interrupts
sync /* Some chip revs have problems here... */
mtmsr r3 /* Update machine state */
blr
```

14 System Call handling

System calling handler is another part that is CPU depenedent.

```
/*
* Handle a system call.
*/
.text
_GLOBAL(DoSyscall)
stw r0,THREAD+LAST_SYSCALL(r2)
lwz r11,_CCR(r1) /* Clear SO bit in CR */
lis r10,0x1000
andc r11,r11,r10
stw r11,_CCR(r1)
#ifdef SHOW_SYSCALLS
#ifdef SHOW_SYSCALLS_TASK
lis r31,show_syscalls_task@ha
lwz r31,show_syscalls_task@l(r31)
cmp 0,r2,r31
bne 1f
#endif
lis r3,7f@ha
addi r3,r3,7f@l
lwz r4,GPR0(r1)
lwz r5,GPR3(r1)
lwz r6,GPR4(r1)
lwz r7,GPR5(r1)
lwz r8,GPR6(r1)
lwz r9,GPR7(r1)
bl printk
lis r3,77f@ha
addi r3,r3,77f@l
```

```
lwz r4,GPR8(r1)
lwz r5,GPR9(r1)
mr r6,r2
bl printk
lwz r0,GPR0(r1)
lwz r3,GPR3(r1)
lwz r4,GPR4(r1)
lwz r5,GPR5(r1)
lwz r6,GPR6(r1)
lwz r7,GPR7(r1)
lwz r8,GPR8(r1)
1:
#endif /* SHOW_SYSCALLS */

/*
** args passed to functions in sys_call_table:
**
** r3 arg1
** r4 arg2
** r5 arg3
** r6 arg4
** r7 arg5
** r8 arg6
** r9 *regs
**
** I am restoring ALL of them here so that if any of these registers
** are trashed in head(_4xx).S [and some of them are currently],
** the args will get passed correctly here. Yes, this is extra
** overhead in the syscall path.
*/
```

```
lwz r3,GPR3(r1)
lwz r4,GPR4(r1)
lwz r5,GPR5(r1)
lwz r6,GPR6(r1)
lwz r7,GPR7(r1)
lwz r8,GPR8(r1)

cmpi 0,r0,0x7777 /* Special case for 'sys_sigreturn' */
beq- 10f
lwz r10,TASK_PTRACE(r2)
andi. r10,r10,PT_TRACESYS
bne- 50f
cmpli 0,r0,NR_syscalls
bge- 66f
lis r10,sys_call_table@h
ori r10,r10,sys_call_table@l
slwi r0,r0,2
lwzx r10,r10,r0 /* Fetch system call handler [ptr] */
cmpi 0,r10,0
beq- 66f
mtlr r10
addi r9,r1,STACK_FRAME_OVERHEAD
blrl /* Call handler */
.globl ret_from_syscall_1
ret_from_syscall_1:
20: stw r3,RESULT(r1) /* Save result */
#ifdef SHOW_SYSCALLS
#ifdef SHOW_SYSCALLS_TASK
cmp 0,r2,r31
bne 91f
#endif
```

```
mr r4,r3
lis r3,79f@ha
addi r3,r3,79f@l
bl printk
lwz r3,RESULT(r1)
91:
#endif
li r10,-_LAST_ERRNO
cmpl 0,r3,r10
blt 30f
neg r3,r3
cmpi 0,r3,ERESTARTNOHAND
bne 22f
li r3,EINTR
22: lwz r10,_CCR(r1) /* Set SO bit in CR */
oris r10,r10,0x1000
stw r10,_CCR(r1)
30: stw r3,GPR3(r1) /* Update return value */
b ret_from_except
66: li r3,ENOSYS
b 22b
/* sys_sigreturn */
10: addi r3,r1,STACK_FRAME_OVERHEAD
bl sys_sigreturn
cmpi 0,r3,0 /* Check for restarted system call */
bge ret_from_except
b 20b
/* Traced system call support */
50: bl syscall_trace
lwz r0,GPR0(r1) /* Restore original registers */
lwz r3,GPR3(r1)
```

```
lwz r4,GPR4(r1)
lwz r5,GPR5(r1)
lwz r6,GPR6(r1)
lwz r7,GPR7(r1)
lwz r8,GPR8(r1)
lwz r9,GPR9(r1)
cmpli 0,r0,NR_syscalls
bge- 66f
lis r10,sys_call_table@h
ori r10,r10,sys_call_table@l
slwi r0,r0,2
lwzx r10,r10,r0 /* Fetch system call handler [ptr] */
cmpi 0,r10,0
beq- 66f
mtlr r10
addi r9,r1,STACK_FRAME_OVERHEAD
blrl /* Call handler */
.globl ret_from_syscall_2
ret_from_syscall_2:
stw r3,RESULT(r1) /* Save result */
stw r3,GPR0(r1) /* temporary gross hack to make strace work */
li r10,-_LAST_ERRNO
cmpl 0,r3,r10
blt 60f
neg r3,r3
cmpi 0,r3,ERESTARTNOHAND
bne 52f
li r3,EINTR
52: lwz r10,_CCR(r1) /* Set SO bit in CR */
oris r10,r10,0x1000
stw r10,_CCR(r1)
```

60: stw r3,GPR3(r1) /* Update return value */

bl syscall_trace

b ret_from_except

66: li r3,ENOSYS

b 52b

#ifdef SHOW_SYSCALLS

7: .string "syscall %d(%x, %x, %x, %x, %x, "

77: .string "%x, %x), current=%p\n"

79: .string " -> %x\n"

.align 2,0

#endif

15. How to build a PowerPC EABI Cross Compiler

15.1 Build PowerPC EABI Cross Compiler

Below are a script example for building the cross compilers for powerpc-eabi.

Purpose: Build a powerpc-eabi cross-compiler

Host OS/CPU:

SunOS solix 5.6 sun4u sparc SUNW,Ultra-4

Target CPU:

powerpc-eabi

Step 0:

* Move your current directory path to your root dir.

* create a dir called gnu_source by using "mkdir gnu_source"

* create a dir called compilers by using "mkdir compilers"

Step 1:

* Visit ftp://ftp.gnu.org/gnu/
* Download required compressed files into your gnu_source directory. After all decompressed, you should get these following.
----binutils-2.9.1
----gcc-2.95.1/
-----newlib-1.8.2/
----- gdb-5.0/
* Remove all the zip files.

Step 2:

Copy and save the following scripts as whatever file name you can offer. For example, offer the filename as "crossMake"

```
# ------------------------------------------------------------
DIR = ~bnn
target=powerpc-eabi

prefix= $(DIR)/compilers/powerpc-eabi

mkdir build-binutils build-gcc build-newlib build-gdb

# Configure, build and install binutils
cd build-binutils
../binutils-2.9.1/configure --target=$target --prefix=$prefix -v make all install

# Configure, build and install gcc
cd ../build-gcc
../gcc-2.95.1/configure --target=$target --prefix=$prefix --with-newlib --with-headers=$
$(DIR)/gnu_source/newlib-1.8.2/newlib/libc/include --with-gnu-as --with-gnu-ld --
enable-languages="c"
-v make all install
```

#Configure, build and install libc
pwd
cd ../build-newlib
../newlib-1.8.2/configure --target=$target --prefix=$prefix -v
make all install

#Configure, build and install gdb
pwd
cd ../build-gdb
../gdb-5.0/configure --target=$target --prefix=$prefix -v
make all install

#Clear history directory/files
pwd
cd ..
rm -rf build-*
#----------

Step 3:

* Add the path to your .cshrc. A line like this:
"~bnn/compilers/powerpc-eabi/bin "
By the way, please replace the "bnn" with your corresponding login name.

* Execute the "source .cshrc" with your unix shell. This is REQUIRED by building the gcc.

#if the crossMake is the script name of yours.
crossMake is resident in your gnu_source dir.
Execute the crossMake script

./crossMake

Step 4:

Wait for 20 minutes while the scripts is busy working.

Step 5:

* Reap your results in your ./compilers.
* You will find a dir called powerpc-eabi is automatically created.
* Check out the powerpc-eabi/bin. The gcc is over there!! If otherwise, something wrong.

15.2 Tips and Bugs

* signed char or unsigned char

By default, the powerpc-eabi-gcc will adopt unsigned char for char type. Please make sure your application aware of this. You can use the "-fsigned-char" in your compiler line or your makefile options.

*sdata and sbss

With the binutils-2.9.1 and gcc-2.95.1, the linker will not support sdata and sbss well. It is adviced that people use "-Map" to investiage your executable image.

A simple fix is offered by using "-msdata=none" to merge all sdata and sbss sections to the corresponding data and bss sections.

*EABI

It is prefereable that people explicitely use the "-meabi" to compile the c source codes.

15.3 CFLAGS Example,

A classical CFLAGS for a makefile to build a embedded application is given below.

```
CFLAGS = -c -g -Wall -Wno-implicit -Wno-format \
-fno-builtin \
-msdata=none \
-mcpu=403 \
-msoft-float \
-meabi \
-fsigned-char \
-O1 -G 1
```

Note that, in the above options, we disable the sdata, specify the CPU type of 403, soft simulating float computing, eabi register/stack convention, signed char as well.

It is desirable to notice that GNU gcc could not support explictely CPU type 405 yet, but support 403. In some cases when/if your assemble routines happen to have some 405 specific instructions, for example, TLB Hi/Lo write, you have to write down its instructio opcodes inside your assemble codes by defining it like this:

```
.word 0x12345678
#0x12345678 is hardcoded representing an 405 #specific instruction, which gcc could not recognize #with -mcpu=403 option
```