Freescale PowerPC™ Architecture Primer

POWRPCARCPRMRM
Rev. 0.1, 6/2005

How to Reach Us:

Home Page:
www.freescale.com

email:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
(800) 521-6274
480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064, Japan
0120 191014
+81 2666 8080
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate,
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
(800) 441-2447
303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: POWRPCARCPRMRM
Rev. 0.1, 6/2005

Contents

Contents

Section **Title** **Page**

Chapter 1
Architectural Beginnings

Fifteen years ago, the PowerPC™ architects saw the necessity for an architecture that was scalable and adaptable to a computing landscape that they knew would continue to surprise and amaze even the most forward-thinking among us, an architecture with room to advance not only general-purpose computing but also networking, telecommunication, transportation, image processing, electronic gaming, any field where microprocessors fit—an architecture that would not only drive change but that would evolve with it.

The PowerPC architecture, although best known to the general public for enabling innovations of Apple's desktop and laptop computers, has become the most pervasive architecture in embedded networking and communications designs. Software and hardware designers have found uses for the architecture that the PowerPC architects could never have expected—which was, after all, exactly what they expected.

In 1991, desktop computers were beige and laptops were black. Portable music meant transistor radios and Walkmans, and it looked like modems were forever going to be stuck at 14.4 Kbps. If word got out that you owned a laser printer, your circle of friends swelled as if you had won the lottery. Java was something programmers drank too much of. You could get an especially tasty cup of it at a small chain of coffee shops in the Pacific northwest.

There were signs of change, though. While some of us were playing Wolfenstein 3D, others were hard at work. Tim Berners-Lee's "WorldWideWeb" hit the Internet that summer, unnoticed by a world that had not yet come to know web-surfing, Googling, blogging, and pop-ups.

Perhaps the most prophetic omens that year came when the publisher Hungry Minds produced its first …*for Dummies* book: *DOS for Dummies*. And Linus Torvalds posted a note on comp.os.minix, in hopes that a fellow netlander could point him to an FTP site with the latest POSIX rules. He was at work on a little project.

The PowerPC architecture helped accelerate the changes, and the PowerPC architecture has evolved as it continues to reshape how computing is done and expand the frontiers of what computers can do.

Divide, Conquer

To make the architecture consistent across generations of products, the PowerPC architects recognized the need for a well structured and clearly defined architecture, that would give rise to a large body of software with breadth and longevity.

To prevent application-level software from being tied to a supervisor level architecture that favored a particular type of OS, the architects made the PowerPC architecture definition modular, defining it in three books:

- The user instruction set architecture (UISA, or Book I)—Defines the application-level instructions and registers that work in any PowerPC computing environment. Implementing the UISA is what defines a PowerPC processor.

The UISA defines the general-purpose register files (GPRs) and floating-point register files (FPRs) through which the computational and load/store instructions pass operands. Integer and floating-point operations are supported by the condition register (CR) to record conditions used by branch instructions. The UISA also defines a comprehensive suite of on-chip special-purpose registers (SPRs) and the Move to/Move from SPR instructions (**mtspr** and **mfspr**) for reading and writing to them. SPRs are used for a variety of user- and supervisor-level functions, but at the UISA-level, they include the link register (LR) and count register (CTR), for determining fetch targets and loop counts.

Around that UISA, the PowerPC architecture has matured and diversified, ensuring binary compatibility across the spectrum of PowerPC processor and operating environments.

- The virtual environment architecture (VEA, or Book II)—Defines aspects of the time base facility and the cache model, It introduces the notion of paged memory (without specifying a page size).
- The operating environment architecture (OEA, or Book III)—Defines an interrupt model that defines offsets for architecturally defined interrupts and save/restore SPRs (SRR0 and SRR1) that automatically save machine state information and a return address when an interrupt is taken and restore that information when a Return from Interrupt (**rfi**) instruction is executed at the end of an interrupt handler.

  It also defines a segmented memory management model that provides an ability to define some memory spaces as variable-sized blocks, configurable through two sets of four pairs of block address translation registers (four pairs of instruction BATs and four pairs of data BATs).

Extension and Expansion

From its inception, it became clear that the PowerPC architecture would service not only computing applications but also a host of embedded applications. Breaking the architecture into three books ensured that the application-level resources defined in the UISA could be implemented independently of the operating system resources defined in the OEA. Early embedded cores in fact favored a software-managed MMU that retained only a basic similarity to the TLB-based resources defined by Books II and III.

By the late 1990s, the architects saw the need for a set of rules to manage such expansion; out of that need, the Book E architecture was born. Because Book E is more flexible both in its definition of operating system resources and its support for extensions to the UISA, the Freescale Book E implementation standards (EIS) provide a standardized set of architectural extensions to the Book E architecture. "Architectural Extensibility—Book E," on page 13, describes Book E in detail. "Architectural Extensibility—The Freescale Book E Implementation Standards (EIS)," on page 14, describes the EIS.

The Book E version of the architecture preserves the UISA intact. Although it retains many of the features defined in the VEA and OEA, Book E expands the definition of some features, and adds some new functionality, such as support for true little-endian byte ordering and the critical interrupt type.

Book E also defines software-managed page translation, offering both fixed- and variable-sized pages, with caching, protection, and memory characteristics (including true little-endian) on a per-page basis.

A software-managed MMU approach allows an operating system to use its own page table structure instead of the OEA-defined hashed reverse page table structure, because most current high-level operating systems (such as Linux, MacOS X, UNIX SysV Release 4, Solaris, Mach, and Windows NT) maintain their own structures to describe virtual memory and have a well-defined layer of code that translates these structures into machine-dependent translations. Book E's alternative software approach has improved performance and made it possible to eliminate complex on-chip hardware MMU assistance.

While the UISA defined a set of instructions common to all implementations, it became desirable to have a mechanism to add application-appropriate instructions and functionality that could be common to a subset of applications. For this purpose Book E also introduces the notion of optional extensions to the programming model, called APUs (auxiliary processing units). APUs aren't necessarily, or even typically, units in the hardware sense, but rather are extensions to the PowerPC programming model. They allow the development of instructions, registers, and interrupts that bring very focussed functionality to specific families of processors but that may not be useful enough generally to be made part of the architecture. APUs preserve the PowerPC unified programming model, yet bring extremely powerful functionality, such as the AltiVec™ technology, to niches within the computing environment.

The first 15 years of PowerPC computing was marked both with the widespread acceptance of home computing and a greater and greater diversification of the computing spaces, in which technological advances could be driven by the portability of the PowerPC architecture. The architecture provides both a solid foundation and an extensible framework from which Freescale Semiconductor is committed to continue delivering powerful computing devices, finely tuned to the increasingly diverse demands across the computing spectrum.

Freescale's PowerPC Processor Families

The first 15 years of PowerPC computing has seen a diversity of processors that matches the diversity of the computing environments the architecture was designed to address. Some family traits arise directly from the architecture specification itself, and others result from design practices that maximize the reuse and elaboration of hardware components and design techniques.

The PowerPC family shows longevity and stability as well as innovation, with devices like the next-generation e700 core, which will offer a huge leap in performance, with the ability to run both 32- and 64-bit PowerPC instructions and scaling to 3 GHz and beyond.

Architecture and Microarchitecture

An architecture is a specification of the functionality that must be provided on a microprocessor. Although it is meticulously and legalistically specific about how instructions must behave, the architecture makes no requirements on how that instruction must be implemented in hardware, or even that it must be implemented in hardware.

Put more simply, the PowerPC ISA is an agreement between software and the hardware platform for which it is executed.

Design features such as pipelining, superscalar dispatch, and parallel execution are part of an implementation's microarchitecture. Such features are commonly associated with RISC architecture, and the PowerPC architecture is defined to make it easy to implement such features, but the architecture does not require them.

In a sense, the architecture is the genetic code that guarantees a common, clearly defined functionality and allows some traits to be pervasive across the broader family, others to be common to smaller groups within a family, and others unique to each individual.

A Common Microarchitectural Model

The MPC603 is the patriarch of the Freescale's PowerPC family, in that its instruction pipeline provides the model for most subsequent generations of PowerPC processors.

The dual-dispatch, superscalar device is built around instruction and completion queues that maintain program order and a precise exception model, but still allowing the integer, floating-point, and load/store units to execute instructions out of order. Although these parallel units can execute instructions out of order, the completion queue ensures that they complete, and update architected registers such as GPRs, FPRs, and other resources, in program order.

Some execution units, such as the load/store unit (LSU) and floating-point unit (FPU), are pipelined; that is they are constructed in stages like an assembly line. As one instruction moves on to the next stage, it vacate the previous stage, allowing the next instruction to perform a portion of its execution. Even though it takes three clock cycles to execute most floating-point instructions, when the FPU pipeline is full, it can finish one instruction per clock cycle. That is, it has a maximum throughput of one instruction per clock cycle.

The MPC604, MPC750, and the MPC74xx processors capitalized on the successes of the MPC603, increasing the number of parallel execution units, enlarging the queueing structure for loads and stores, eliminating bottlenecks in the instruction pipeline by introducing reservation stations and issue logic that reduce stalls in the pipeline, providing branch prediction logic that increases the likelihood of correctly predicting branches, and reducing the refetch latency when a prediction is incorrect.

Instruction parallelism increased with a further refinement on the integer units. The fully-featured integer unit of the 603 was split into two subtypes: a complex integer unit that handled longer latency operations, such as multiplication and division, and a simple integer unit that provided an express lane for shorter latency operations, such as addition, logical operations, shifts, and comparisons.

By providing multiple instances of the smaller simple integer units and only a single complex integer unit, instruction throughput increased without the die-space and power costs of replicating fully-featured integer units.

The AltiVec APU, described in "AltiVec APU," on page 24, increased instruction parallelism with the added vector permute, vector floating-point, simple vector integer, and complex vector integer execution units—all using the 32 128-bit vector registers (VRs) for source and target operations.

While instruction unit designers for the e600 processors continue to capitalize on the advantages the PowerPC RISC architecture brings to designing pipelined processors with many execution units, the scalability of the architecture has made it possible to increase performance marks within the power, size and thermal constraints of the embedded segments.

The e500 and e200 cores efficiently address the needs of communication, digital signal processing (DSP), automotive, and pervasive computing environments where interrupt responsiveness and low power consumption are essential. Both implement EIS-defined embedded floating-point and vector instructions that use extended 64-bit GPRs, which are also used for 32-bit instructions defined by the UISA, Book E, and the EIS.

The e500 can dispatch and complete two instructions per clock cycle. It has five execution units: a branch unit, an LSU, and three integer units. The e500v2 additionally implements the embedded double-precision floating-point APU, which also uses the 64-bit GPRs

For the automotive industry, control, responsiveness, and reliability are a matter of life and death, so the e200 PowerPC processor was designed to be simpler, and thus easier to verify, compared to the higher-performance members of the PowerPC family. But while it's not the processor you would turn to for 3D animation and scientific computing, it brings greater performance and on-chip functionality to high-performance microcontrollers such as the MPC5554 to meet system demands for increased system performance and complexity in a complex, real-time environment.

As processor microarchitecture has advanced, so have semiconductor processes: speeds increased, power consumption was reduced, and die sizes continued to shrink.

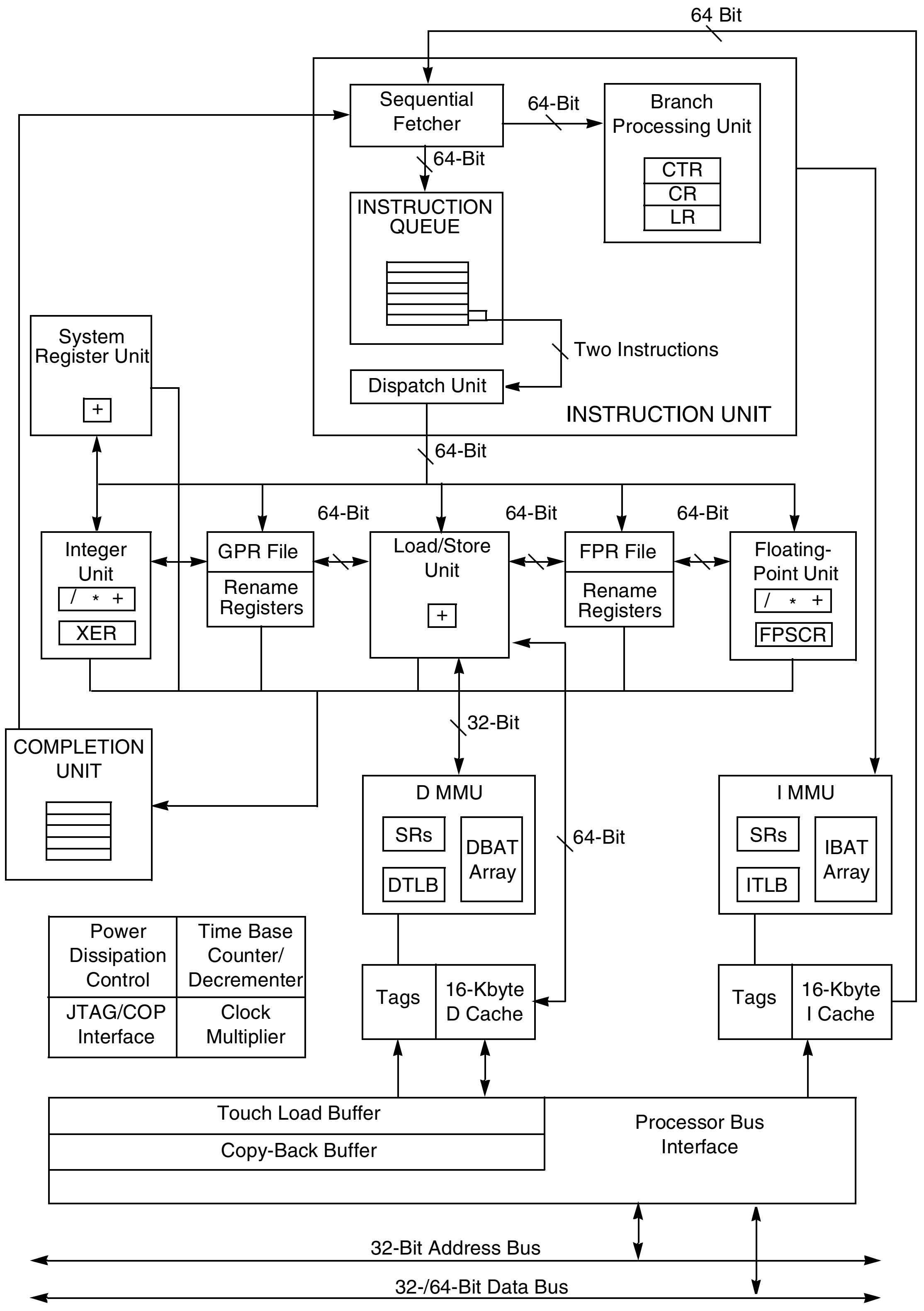


Figure 1. MPC603 Block Diagram

Figure 2 compares the pipelines of the PowerPC cores.

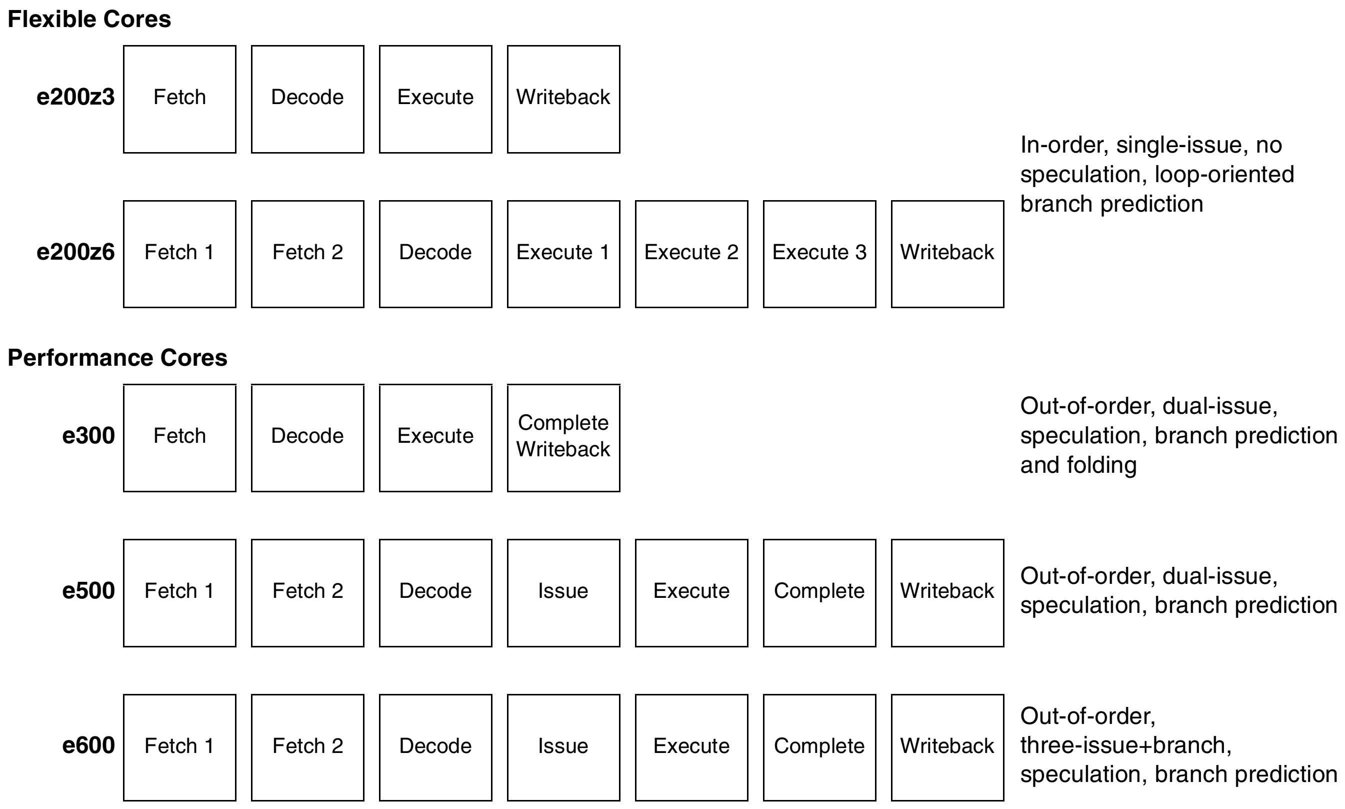


Figure 2. Freescale PowerPC Cores—Pipeline Comparison

A Family Portrait—Cores and Platforms

A system is made up of more than just a processor core that executes instructions. Over the past decade, processor design has evolved to integrate both processor cores and associated system logic intellectual property (IP) to form complete systems on a chip (SoCs). Leveraging its advanced chip-level integration capabilities, Freescale is now delivering common SoC platforms that wed the company's PowerPC core portfolio with its broad intellectual property (IP) portfolio, offering connectivity and integration to the networking, communications, and pervasive computing markets.

During more than ten years of PowerPC core development, Freescale has delivered increasingly higher-performance cores, bringing together such features as the AltiVec vector processing APU with other advanced process technologies, such as silicon-on-insulator (SOI). Freescale also has been an innovator in connectivity solutions in areas such as network acceleration, RapidIO interconnect, and passive optical networking. Freescale pioneered processor integration in the communications market and continues to lead the way with the ever-evolving PowerQUICC™ family. The PowerPC-based MPC500 family is suitable for complex real-time control, such as automotive, requiring computationally complex algorithms including double-precision floating-point arithmetic. Coupling a PowerPC core with Freescale's advanced time processing unit (TPU) puts Freescale on the cutting-edge of real-time engine control.

Freescale's PowerPC SoC platforms are engineered to deliver scalable performance to meet a wide spectrum of processing and I/O requirements. Designed for developing both standard and semicustom products, these platforms are supported by a comprehensive system of development tools from Freescale's Smart Networks Alliance Program.

e200 Cores and Platforms

The e200 family of embedded cores is highly configurable and customizable, and is intended for cost-conscious, embedded real-time applications. e200 cores and complexes are offered as a hard or soft deliverable, based on customer requirements.

e200 family cores offer low interrupt latency, low-power design through clock gating, variable cache sizes, Nexus 2/3 support, and a 64-bit AMBA™ AHB bus interface unit.

Platforms may include optionally defined elements such as cache and MMU interfaces, Nexus Class 2 or 3 debug, and signal processing engine (SPE) and floating-point APUs, interrupt controllers, and peripheral interfaces.

The e200z6, implemented in the MPC5556 microcontroller, is code-compatible with e500 core (including **isel**, SPE, and single-precision floating-point APUs). The e200z6 has a unified 32-Kbyte 8-way set-associative cache, with a 32-entry unified MMU, and an AMBA AHB Bus interface. It features a single-issue, in-order, seven-stage pipeline. To accelerate loops, it incorporates a branch target address cache (BTAC).

e300 Cores and Platform

The ongoing presence of the e300 testifies to the longevity of both the PowerPC architecture and the microarchitecture. The e300 was introduced in 1994 as the MPC603, running at 100 MHz, and continues to be proliferated more than a decade later reaching speeds exceeding 600 MHz.

The e300 PowerPC core and platform address the low- to mid-range performance needs of the market. The e300 core is an enhanced version of the popular PowerPC 603e core used in previous-generation PowerQUICC II devices that scale from 266 to 667 MHz in 130 nm process technology. Enhancements include twice as much L1 cache as the 603e (32-Kbyte data and instruction caches) with integrated parity checking and other performance-oriented features. The e300 core is fully software-compatible with existing 603e core-based products and provides the processing foundation for the company's new MPC8349E PowerQUICC II™ Pro communications processor family.

e500 Cores and Platform

The e500 core delivers flexibility for application-specific optimizations and leverages both EIS-defined and implementation-specific APU extensions geared especially for the needs of the embedded market, enabling flexible SoC platform solutions that balance performance and advanced features with the need for low power consumption.

The e500 implements an alternative, cost-efficient embedded-friendly floating point and vector instruction set with extended 64-bit GPRs, using the same GPRs for UISA-defined instructions as well. The e500 can dispatch two instructions per clock cycle to five execution units.

The e500 core and platform, which power Freescale's PowerQUICC III™ communications processors, currently scales frequencies exceeding 1 GHz.

e600 Cores and Platforms

The e600 core and platform offer higher performance than the G4 core used in MPC74xx family of PowerPC processors, while maintaining pin and instruction set compatibility. The e600 core is planned to scale beyond 2 GHz and to support chip multiprocessing (CMP). Like the G4 core, the e600 core issues as many as four instructions per clock cycle (three instructions plus one branch) into eleven independent execution units. It also includes a full 128-bit implementation of the AltiVec APU.

Table 1 summarizes the features of some of the PowerPC family members.

Table 1. Freescale PowerPC Processor Family Summary

Platform	Traits/Features	Target Market	Integrated Device
e200 (Derived from MPC5xx)	Fully synthesized, register-based. Complexes and platforms offered as a hard or soft deliverable. Single-issue, supporting UISA and SPE, embedded vector/scalar SPFP, debug interrupt APUs Book E/EIS-compliant MMU, 32-entry fully associative TLB supporting 4-Kbyte to 256-Mbyte page sizes, 8-bit PID0 supports up to 255 translation IDs at any time. Platforms may include optionally defined elements, such as support for a multiple-master system bus, cache and MMU interfaces, Nexus Class 2 or 3 debug, interrupt controllers, and peripheral interfaces.	Low-power cost- and time-to-market–sensitive, real-time embedded applications needing complex real-time control and low interrupt latency. Harsh environments, such as automotive and avionics, with full support for temperature ranges from -40°C to +125°C.	MPC5554 high performance microcontroller
e300 (MPC603/G2)	Enhanced version of the 603e PowerPC core used in PowerQUICC™ II processors. Scales from 266 to 667 MHz in 130nm-process technology. Doubled L1 cache (32-Kbyte data and instruction caches) with integrated parity-checking and other performance-oriented features. Fully software-compatible with 603e-based products	Low- to mid-range performance	PowerQUICC II, PowerQUICC II Pro communications processor family.

Table 1. Freescale PowerPC Processor Family Summary (continued)

Platform	Traits/Features	Target Market	Integrated Device
e500	Low-power, high-performance embedded cores. SPE, single-precision vector and scalar floating-point APUs, double-precision scalar floating-point APU (e500v2), cache locking, machine check APUs. The high-performance e500 core delivers flexibility for application-specific optimizations and leverages APUs for instruction set extensions.	Embedded implementations that require highly configurable cores Enabling flexible SoC platform solutions that provide an optimal balance of performance, advanced features and power consumption.	PowerQUICCIII family
e600 74xx/G4	Based on MPC74xx/G4 processors. The MPC7448 represents another performance leap in Freescale's discrete processor line. It provides more than 1.5 GHz of processing power, increased L2 cache (1 Mbyte), and a higher-speed bus than its predecessors. It is also pin-for-pin and software compatible with the MPC74xx processors—an easy drop-in to existing designs. The MPC7448 keeps the power low and can run on less than 10 Watts at 1.4 GHz.	Computing applications both in consumer and in embedded spaces.	MPC8641/MPC8641D (dual-core) processors.

Semicustom Processor Platforms

Freescale Semiconductor's Application-Specific Products Operation offers scalable high-performance PowerPC platforms based on the e300 core from the PowerQUICC II Pro family and the e500 core from the PowerQUICC III family.

By designing all of our products for reuse, Freescale enables rapid development of SoC semiconductor devices. The ability to develop products within a 6- to 9-month time frame accelerates our customers' time to market. The broad acceptance of the PowerPC architecture allows significant reuse of their software investment.

Chapter 2 Architecture Definition

This chapter describes the evolution of the PowerPC architecture, and how it has been extended through the introduction of Book E and the Freescale Book E implementation specification (EIS).

The PowerPC Architecture as a RISC Architecture

The breadth of Freescale's PowerPC devices is made possible by the PowerPC architecture, a RISC architecture that was designed to be extensible and scalable.

A RISC architecture is one with simplified, single-purpose instructions. Most notably, computation instructions don't access memory directly, but instead use a set of on-chip registers to hold source and destination operands, which are read from and written to memory by a separate set of load and store instructions. This decoupling is fundamental to RISC and critical to performance, and it is why a RISC architecture is sometimes called a load/store architecture or a register-to-register architecture. Beyond that, a RISC instruction set reduces arithmetical operations to the basics—addition, multiplication, and division—rather than combining operations so that a single instruction can perform complex functions, such as sine and cosine trigonometric functions.

At first glance, knowing that RISC stands for reduced instruction-set computing may conceal more than it reveals, if one infers that reduced means fewer instructions. Decoupling computation from memory accesses and simplifying instructions such that they perform simple, equal- or at least low-latency, operations increases the number of instructions.

So, in a consumer-oriented RISC architecture such as the PowerPC architecture what gets reduced is the complexity of individual instructions. In contrast, a CISC architecture, which more tellingly stands for complex instruction-set computing, can have many high-latency, special-purpose, microcode-based instructions. In the 1980s, memory was a premium, so being able to specify a sequence of operations in a single microcode-based instruction made more efficient use of the instruction cache and memory. However, as memory became cheaper, it became possible to exploit a direct marriage between the binary code and the underlying machine executing it.

In addition to its memory savings, at first glance, having such a complex instruction may seem useful in some applications, but it is difficult to optimize a processor for which instructions behind it must wait until the entire function is performed. A RISC instruction set reduces the functionality of individual instructions such that their latency can be as short as possible and as nearly deterministic as possible. Although such an approach may increase the number of instructions, it makes it much easier to optimize software to take advantage of the similar, predictable latencies of multi-purpose instructions.

Moreover, a CISC architecture typically must create operand registers that can be used only by specific instructions; this increases die size and complicates the design.

Having an orthogonal instruction set, one where all computational instructions can access the same register file, avoids replication of functionality among instructions and registers. However, there are instances where it makes sense to break with this guiding principle so that a single instruction may perform two functions. For example, floating-point units in most PowerPC processors are optimized such that a floating-point multiply-add instruction (**fmadd** or **fmadds**) executes with the same latency as either a floating-point multiply or add. There is no duplication of add or multiply logic, because floating-point multiply, add, and multiply-add all share the same execution unit and FPR register file, and being able to combine the operations improves performance.

Duplication of functionality can be especially powerful when it comes to the definition of vector instructions. Although the PowerPC architecture defines no vector instructions, the AltiVec extension to the PowerPC architecture allows execution of the same instruction across several parallel data elements—up to 16 operations per clock cycle—an even more compelling argument that it sometimes makes sense to violate a strict, academic definition of RISC computing. Here computational logic of scalar execution units is replicated, but the performance increases greatly outweigh any increase in die size and microarchitectural complexity.

Another feature that has come to be associated with RISC architecture is the definition of an instruction set whose opcodes are all the same size. All PowerPC instruction opcodes are 32 bits wide. Many instructions use all of the opcode bits, while others leave some bits unused and could be smaller. However having same-sized instructions simplifies the logic; the fetching mechanism does not need to check for instruction size, and the size of the instruction cannot contribute to fetch misalignment.

Architectural Extensibility—AltiVec Technology

The first major extension to the PowerPC architecture came with the introduction of the MPC7400 processor, which implemented the AltiVec SIMD (single-instruction/multiple data) instruction set. It is important to note that the AltiVec technology is not part of the PowerPC architecture definition, but is an extension to it; that is, an APU. The functionality of the base UISA is extended with a comprehensive set of 128-bit vector instructions.

The AltiVec APU was developed to provide the following:

- The ability to equip a single high-performance RISC microprocessor with DSP-like computing power for controller and signal-processing functions
- A one-part/one–code-base approach to product design that also offers a tremendous jump in performance
- A vector processing engine that provides highly parallel operations, allowing simultaneous execution of up to 16 operations per clock cycle
- Wide data paths and wide field operations that offer acceleration for a broad array of traditional computing and embedded processing operations
- A programmable solution that can easily migrate by using software upgrades to follow changing standards and customer requirements
- An integrated solution 100% compatible with the industry-standard PowerPC architecture that simplifies design and support

The ability to simultaneously perform mathematical computation, logical operations, and bit manipulation offers designers a competitive edge in realms of computing far removed from those envisioned by the AltiVec architects.

AltiVec defines the following:

- 162 instructions that are an extension to the PowerPC definition
- Four-operand, non-destructive instructions
  - Up to three source operands and a single destination operand
  - Supports advanced “multiply-add/sum” and permute primitives
- Simplified load/store architecture
  - Simple byte, half-word, word and quad-word loads and stores
  - Virtually no misaligned accesses—software managed via permute instruction

The AltiVec technology not only provided powerful new computing resources, it promoted an important concept—the value of making architectural extensions to the architectural programming and interrupt model that, although its instructions and registers adhere to the conventions laid out in the PowerPC architecture, is not part of the architecture itself. The AltiVec extension provided a prototype for the concept of an APU (auxiliary processing unit) which has become a key factor in Freescale’s strategy of developing PowerPC microprocessors and embedded cores that precisely target specialized computing niches and that provide a wider range of design trade-offs.

Architectural Extensibility—Book E

The ability to extend the ISA is central to the PowerPC Book E architecture philosophy. Book E defines reusable opcode space that can be defined across multiple APUs. For example, opcodes for instructions defined by the signal-processing engine (SPE) APU implemented on the e500 and e200 cores overlap the opcode space defined by the AltiVec extension. Other APUs can reuse those same opcodes to bring focussed functionality to other processor families.

APUs extend the architecture. they can be as simple as a single instruction (for example, the Integer Select APU which consists of the **isel** instruction) or a register or a set of fields within a register defined by the PowerPC architecture. It can also be as expansive as the SPE APU, which extends the PowerPC architectural definition of the 32-bit GPRs to support 64-bit vector operations and defines new instructions, new registers, new fields within existing registers, and new interrupts.

Book E also provides alternatives to the MMU model defined in the classic architecture’s Books II and III. Where the classic architecture supports hardware-based page address translation with fixed 4-Kbyte pages, the Book E MMU is strictly software-managed and supports fixed- and variable-sized pages. Where the classic architecture defined block address translation (BAT) SPRs that could provide a single translation for variable-sized blocks of memory space, Book E processors do this with support for variable-sized pages. For more information, see “Memory Management Unit (MMU) Model,” on page 44.

Architectural Extensibility—The Freescale Book E Implementation Standards (EIS)

Although the PowerPC architecture was designed such that devices could implement the UISA without fully implementing the VEA and OEA, the original, classic architecture definition had no framework to manage evolution of architectural extensions. New processors often solved problems and defined features differently from one another. A mechanism was needed to manage such architectural evolution so consistency and reuse could be maintained without hindering extensibility.

Recognizing that need for both flexibility and consistency, Book E architects defined many features in a more general way, leaving specific details to the implementation. For example, the Book E MMU model defines the TLB Read Entry and TLB Write Entry instructions (**tlbre** and **tlbwe**) for reading and writing the TLBs. However, details of how this is accomplished are left as implementation dependent. Rather than leaving this up to individual implementations, the Freescale architects defined an MMU architecture for all Freescale Book E devices. This MMU model defines MMU assist (MAS) SPRs, which hold the translation, memory protection, and other memory attributes necessary for page translation. It also defines that the **tlbre** and **tlbwe** instructions transfer this configuration information between the MAS registers and the TLBs.

This MMU model is part of the Freescale Book E Implementation Standards (EIS). Note that because individual Freescale processors have different requirements (for example e200 cores have a unified memory model and e500 cores have a Harvard-style architecture), which MAS registers and which fields within those registers are used is left up to the implementation or family of implementations. Likewise, the ability of the e500v2 to use a 36-bit address space is made possible by the MAS7 register, which extends the 32-bit address space.

The EIS ensures consistency across all Freescale Book E devices and provides a common layer of architecture between Book E and the implementation, further ensuring a unified programming model beyond the PowerPC architecture itself.

The EIS defines the following:

- Extension to the Book E MMU model, allowing the implementation of fixed- or variable-sized pages, or both, managed through software
- L1 cache model, including L1 cache configuration registers and L1 control and status registers
- All APUs that are intended for reuse, guaranteeing consistency across processor families
  - AltiVec APU
  - Signal-processing engine APU
  - Embedded floating-point APUs
    - Single-precision vector and scalar APU
    - Double-precision scalar APUs
  - Cache locking APU. Allows instructions and data to be locked into their respective caches on a cache line basis. Locking is performed by a set of touch and lock set instructions. This functionality can be enabled for user mode by setting MSR[UCLE]. Cache locking requires the appropriate access to the address. The APU also provides resources for detecting and handling

overlocking conditions (that is, attempting to lock values into portions of the cache that are already locked).

  — Machine check APU. Includes an enhanced definition of the machine check interrupt type similar to the Book E–defined critical interrupt. The Return from Machine Check Interrupt instruction, **rfmci**, restores data from MCSRR0 and MCSRR1.
  — Performance monitor APU. The EIS defines the performance monitor facility as an APU. Software communication with the performance monitor APU is achieved through performance monitor registers (PMRs) rather than SPRs. The PMRs are used for enabling conditions that can trigger an APU-defined performance monitor interrupt.
  — Alternate time base APU. This APU, implemented on the e500v2, defines a 64-bit time base counter that differs from the PowerPC defined time base in that it is not writable and counts at a different, and typically much higher, frequency. The alternate time base always counts up, wrapping when the 64-bit count overflows.

  Note that Freescale processors are not required to implement any of the APUs, but if one is implemented, it must adhere to the EIS architectural definition.

- Other aspects of instruction behavior defined by Book E as implementation dependent

The landscape for embedded computing is vast and varied. Transportation, telecommunications, network, and electronic gaming implementations share a need for a powerful core tightly coupled with the surrounding integrated logic. These applications share a common need to leverage a support ecosystem that is larger than what their application space supports alone. In the early days of PowerPC computing, those needs were met with implementation-specific solutions, but as the embedded market has developed, the PowerPC architecture partners recognized the value of a consistent platform from which very specific needs from among the various niches in embedded computing could be met.

PowerPC Architecture Overview

This section provides an overview of the programming, interrupt, cache, and MMU models as they are defined by the PowerPC architecture and the EIS.

A Common User Instruction Set Architecture

The instructions and registers that make up the user instruction set architecture (UISA) make it possible for the PowerPC architecture to reach into so many computing spaces, from high-end, multiprocessor, highly parallel workstation and server environments to low-power, special-purpose embedded devices that use real-time operating systems.

Having a unified application level is critical, but to serve the ever-broadening computing spectrum, the PowerPC architecture has provided flexibility at the supervisor level to allow for a coherent architecture that eloquently addresses the varying critical needs of operating systems.

The classic and Book E definitions of the PowerPC architecture share a common user instruction set (UISA), which makes PowerPC cores binary compatible. The EIS extends this flexibility even further, with auxiliary processing units (APUs) that define application- and supervisor-level extensions to the instruction set, making available powerful computing tools where they are needed, such as the AltiVec APU, without forcing those tools to be a formal, required part of the PowerPC architecture itself.

In addition, individual processor implementations may provide additional extensions to the architectural descriptions and may optimize implementations of UISA instructions to more efficiently suit the needs of the computing segment for which they are designed. Such variations are explained in an implementation’s reference manual.

If the architecture provides the genetic material for the evolving family of PowerPC processors, the UISA has been the unchanging mitochondrial DNA, defining not only the instructions that comprise the application-level machine code, but the forms and addressing modes for instructions defined by other levels of the architecture, by Book E, by the EIS, and by individual implementations. As the instruction set continues to be extended, users can rely on familiar register-to-register computation instructions that retain the same basic form and syntax. Most of those instructions use architecture-defined GPRs and FPRs, but others use alternate register files, such as AltiVec’s vector registers (VRs) and the signal processing engine’s enhanced GPRs, that are widened to accommodate two-element 64-bit operands.

Whereas the classic architecture defines special-purpose registers (SPRs) and two instructions to access them (Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**)), Book E takes that model and defines optional device control registers (DCRs) and **mtdcr** and **mfdcr** instructions, and the EIS-defined performance monitor APU defines performance monitor registers (PMRs) and **mtpmr** and **mfpmr** instructions, all based on models provided by the UISA.

The Book E conventions follow the basic descriptions in the classic PowerPC architecture with some changes in terminology. For example, distinctions between user- and supervisor-level instructions are maintained, but the classic PowerPC designations—UISA, VEA, and OEA—are not retained for the discussion of Book E instructions.

A notable difference in the Book E definition is the introduction of auxiliary processing units (APUs). APUs allocate opcode and register space for extending the instruction set without affecting the instruction set defined by Book E. This facilitates the development of special-purpose resources that are useful to some embedded environments but impractical for others. Instructions from multiple APUs may be assigned the same opcode numbers of the allocated opcode space.

The EIS defines many APUs. These APUs are not required on all devices, but devices that implement them do so strictly following the EIS architectural definition. In addition, an implementation may provide its own APU that is not a part of the EIS. For example, the e500 core provides an APU for locking branch predictions in its branch target buffer (BTB), a feature that is not part of the EIS.

APUs may be any combination of instructions, optional behavior of Book E–defined instructions, registers, register files, fields within Book E–defined registers, interrupts, or exception conditions within Book E–defined interrupts.

All instructions defined by the PowerPC architecture have the following characteristics:

- Data organization in memory and data transfers—Bytes in memory are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

  Memory operands can be bytes, half words, words, or double words or, for the load/store multiple instruction type and load/store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

Note that APUs, such as the AltiVec APU, can have larger operands than those defined by the PowerPC architecture.

- Alignment and misaligned accesses—The operand of a single-register memory access instruction has an alignment boundary equal to its length. An operand's address is misaligned if it is not a multiple of its width.

  Some instructions require their memory operands to have certain alignment. In addition, alignment can affect performance. For single-register memory access instructions, the best performance is obtained when memory operands are aligned.

Instruction Model

This section describes instructions and instruction classes defined for both classic PowerPC and Book E definitions. These instructions are divided into the following functional categories:

- Integer instructions—These include arithmetic and logical instructions. See "Integer Instructions,"
- Floating-point instructions—These include the floating-point instructions defined by the PowerPC architecture and the floating-point vector and scalar arithmetic instructions defined by the EIS. See "Floating-Point Instructions," on page 19.
- Load and store instructions—These move data between registers and system memory. See "Load and Store Instructions," on page 21.
- Branch and flow control instructions—These include branching instructions, CR logical instructions, trap instructions, and other instructions that affect instruction flow. See "Branch and Flow Control Instructions," on page 21.
- Processor control instructions—These instructions are used for accessing architecturally defined registers, such as SPRs, the condition register, and the MSR. See "Processor Control Instructions,"on page 23.
- Memory synchronization instructions—These ensure that access to memory and memory resources occur in correct order with respect to memory operations generated by other instructions or by other memory devices. See "Memory Synchronization Instructions,"
- Memory control instructions—These instructions provide control of caches and TLBs. See "Memory Control Instructions," on page 23.

Integer instructions operate on word operands and use the GPRs. Floating-point instructions operate on single- and double-precision floating-point operands. The PowerPC classic architecture–defined floating-point instructions use FPRs, while embedded floating-point APUs use the GPRs. Instructions are 4 bytes (one word) long and word-aligned, and there is a penalty if instructions are not word aligned. The architecture provides byte, half-word, and word operand loads and stores between memory and a set of 32 general-purpose registers (GPRs). Classic PowerPC provides for word and double-word operand loads and stores between memory and a set of 32 floating-point registers (FPRs). When data is loaded from memory to an FPR, the architecture requires that both double-precision and single-precision data be stored in double-precision format.

Arithmetic and logical instructions do not read or modify memory. To use the contents of a memory location in a computation and then modify the same or another location, the memory contents must be loaded into a register, modified, and then written to the target location using load and store instructions.

Note that the PowerPC architecture allows out-of-order, parallel execution but requires in-order completion. Some operations, especially those that update the processor state, must be performed in an order that guarantees that adjacent instructions complete in the proper context. Such serialization is handled by the instruction pipeline microarchitecture.

Similarly, it is sometimes necessary to insert instructions into the program flow that ensures that instructions that generate accesses to memory and memory resources such as TLBs to be performed in the desired order. Memory synchronization instructions control the order in which memory operations performed with respect to asynchronous events and the order in which memory operations are seen by other mechanisms that access memory.

Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of instructions not described in that processor's reference manual.

Simplified Mnemonics

The description of each instruction in the architecture includes the mnemonic and a formatted list of operands. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for some of the frequently used instruction variants, such as branch conditional, compare, trap, and rotate and shift instructions. These simplified mnemonics extend the mnemonics to incorporate numerical information provided in operands. For example, there are simplified mnemonics for the **mtspr** and **mfspr** instructions that, instead of requiring the SPR number as operand, incorporate the name of the SPR into the mnemonic. To load a value from a GPR into the count register, **mtctr r**S can be used instead of **mtspr 9,r**S.

The **sub r**A**,r**B**,r**C simplified mnemonic reorders the operands of as **subf r**A**,r**C**,r**B.

The **bdnz** target simplified mnemonic (decrement CTR and branch if CTR is non-zero) incorporates the BO and BI encodings of the more complex **bc** 16,0**,**target instruction.

Simplified mnemonics for individual processors are listed in each reference manual.

Instruction Set Overview

This section provides a brief overview of the PowerPC instruction set. A full list of instructions can be found in the *Programming Environments Manual* (PEM) and the *EREF: A Reference for Freescale Book E and the e500 Core* (EREF).

The instructions defined by the PowerPC architecture occupy specifically defined spaces in the opcode space. The PowerPC architecture allocates a portion of the opcode space not occupied by defined instructions for use (and reuse) by extensions to the instruction set architecture. For example, the AltiVec and SPE instruction sets overlap within that allocated space. For backwards compatibility, the PowerPC architecture preserves opcodes for discontinued instructions (such as some POWER architecture instructions) and refers to all other undefined opcode space as reserved. An implementation that attempts to execute a reserved instruction, or any other instruction that is not implemented, may generate an interrupt.

Integer Instructions

This section describes the integer instructions. These instructions are user-level instructions and consist of the following:

- Integer arithmetic instructions
- Integer compare instructions
- Integer logical instructions
- Integer rotate and shift instructions

Integer instructions use the content of the GPRs as source operands and place results into GPRs and the XER and CR fields. Integer instructions are shown in Table 2.

Table 2. Integer Instructions

Instruction Category	Instruction Name	Options
Integer arithmetic (**add***x*, **div***x*, **mul***x*, **neg***x*, **subf***x*)	Add	Carrying, extended, immediate, shifted, minus one, zero
	Divide	Word, unsigned
	Multiply	High word, low word, unsigned, immediate
	Negate	—
	Subtract from	Carrying, extended, immediate, minus one, zero
Integer compare (**cmp***x*)	Compare	Immediate, logical
Integer logical (**and***x*, **cnt**, **eqv**, **exts***x*, **nand**, **nor***x*, **or***x*, **xor***x*)	AND	Immediate, shifted, with complement
	Count	Leading zeros, word
	Equivalent	—
	Extend sign	Byte, half word
	NAND	—
	NOR	—
	OR	Immediate, shifted, complement
	XOR	Immediate, shifted
Integer rotate and shift (**rlw***x*, **slw***x*, **srw***x*, **sraw***x*)	Rotate left word	Immediate, then AND with mask, then mask insert
	Shift	Left word, right word, algebraic word, immediate

Floating-Point Instructions

The IEEE-754 standard defines conventions for 64- and 32-bit floating-point arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision arithmetic instructions to have either (or both) single-precision or double-precision operands, but states that single-precision arithmetic instructions should not accept double-precision operands.

The instructions follow these guidelines:

- Double-precision arithmetic instructions can have single-precision operands but always produce double-precision results.
- Single-precision arithmetic instructions require all operands to be single-precision and always produce single-precision results.
- Conversion from double- to single-precision must be done explicitly by software; conversion from single- to double-precision is done implicitly by the processor.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic uses two additional bit positions to avoid potential transient overflow conditions. An extra bit is required when denormalized double-precision numbers are prenormalized. A second bit is required to permit computation of the adjusted exponent value in the following examples when the corresponding interrupt enable bit is one:

- Underflow during multiplication using a denormalized operand
- Overflow during division using a denormalized divisor

The EIS defines embedded vector single-precision scalar and vector floating-point APUs and the embedded scalar double-precision APU. The vector scalar and double-precision floating-point APUs require a GPR file with thirty-two 64-bit registers because vector floating-point instructions operate on the entire 64 bits of the GPR, but contain two 32-bit data items that are operated on independently of each other in a SIMD fashion. The scalar floating-point APU requires a GPR register file with thirty-two 32- or 64-bit registers. When implemented with a 64-bit register file, the scalar single-precision instructions only operate on 32 bits of a 64-bit register.

Floating-point instructions are user-level instructions and are shown in Table 3.

Table 4. Floating-Point Instructions

Instruction Category	Instruction Name	Options
Floating-point arithmetic (**fadd***x*, **fsub***x*, **fdiv***x*, **fmul***x*, **fmadd***x*, **fmsub***x*, **fnmadd***x*, **fnmsub***x*)	Add	Single, double
	Subtract	Single, double
	Divide	Single, double
	Multiply	Single, double
	Multiply-add	Single, double
	Multiply-subtract	Single, double
	Negative multiply-add	Single, double
	Negative multiply-subtract	Single, double
Floating-point rounding and conversion (**fcti***x*, **fr***x*)	Convert to integer	Word, double word, round to zero
	Round to single-precision	—
Floating-point compare and select (**fcmp***x*)	Compare	Ordered, unordered
	Select	—
Floating-point status and control register (**mtf***x*, **mff***x*)	Move from FPSCR	—
	Move to FPSCR	Bit 0, Bit 1, fields, immediate

Table 4. Floating-Point Instructions (continued)

Instruction Category	Instruction Name	Options
Floating-point move (**fmr***x*, **fneg***x*, **fabs***x*, **fnabs***x*)	Move register	—
	Negate	—
	Absolute value	—
	Negative absolute value	—
Floating-point load/store (see Table 5)	—	—

Load and Store Instructions

Load and store instructions are issued and translated in program order; however, the accesses can occur out of order. Synchronizing instructions are provided to enforce strict ordering. Most implementations support load and store instructions as follows:

- Integer load instructions
- Integer store instructions
- Integer load and store with byte-reverse instructions
- Integer load and store multiple instructions
- Memory synchronization instructions

Some implementations do not implement Book E floating-point load and store instructions. Load and store instructions are user-level instructions and are shown in Table 5.

Table 5. Load/Store Instructions

Instruction Category	Instruction Name	Options
Integer load (**lb***x*, **lh***x*, **lw***x*, **lmw**, **lswi**)	Load	Byte, word, half word, algebraic (half word), byte reverse, and zero, with update, indexed
	Load multiple word	—
	Load string word [1]	Immediate, indexed
Integer store (**stb***x*, **sth***x*, **stw***x*)	Store	Byte, word, half word, byte-reverse, with update, indexed
	Store multiple word	—
	Store string word [1]	Immediate, indexed
Floating-point load (**lf***x*)	Load floating-point	Double, single, with update, extended, indexed
Floating-point store (**stf***x*)	Store floating-point	Double, single, with update, extended, indexed, as integer word

[1] Not supported on all implementations

Branch and Flow Control Instructions

The specific methods of branch prediction vary from implementation to implementation, but most high-performance PowerPC implementations from Freescale provide some form of branch prediction. The branch predictor is used to predict the direction and target of a branch, and then to redirect the fetch unit so that it provides instructions from the predicted target of the branch. When the branch is able to execute (any pending CR bits, CTR values, or LR values are available), the branch execution unit compares the actual direction and target with the predicted direction and target. In the case of a correct prediction, no further action is taken. If the prediction is incorrect, the fetched instructions are purged, and instruction fetching continues along the correct path.

For branch conditional instructions, the BO operand specifies the conditions under which the branch is taken. The BO operand has five bits, which provides for a total of 32 different meanings, 17 of which are typically used in practice (and only 9 if static branch prediction hints are ignored).

Simplified mnemonics help the programmer use branch instructions without having to refer to all the options for the BO operand and similar multi-bit operands.

Branch instruction functions

- Branch instructions redirect instruction execution conditionally based on the value of bits in the CR. For branch conditional instructions, the BO operand specifies the conditions under which the branch is taken.
- CR logical instructions perform logical operations on CR contents that help determine branching conditions.
- Branch instructions can decrement and examine the CTR to determine the direction.
- The trap instruction tests for a specified set of conditions. If any of the conditions tested by a trap instruction are met, the system trap type program interrupt is taken. If the tested conditions are not met, instruction execution continues normally.
- The System Call (**sc**) instruction lets a user program call on the system to perform a service and causes the processor to take a system call interrupt. Executing this instruction causes the system call interrupt handler to be invoked. System Call instructions can be either user- or supervisor-level.

The branch target buffer (BTB) locking APU (not defined by the EIS) gives e500 users the ability to lock, unlock, and invalidate BTB entries for improved deterministic branch behavior.

The supervisor-level **rfi** instruction is used for returning from an interrupt handler. The **rfci** instruction is used for critical interrupts; **rfmci** is used for machine check interrupts, and **rfdi** is used for debug interrupts in the machine check APU and debug APU, respectively. See "Interrupt Model," on page 40.

Branch and flow control instructions are shown in Table 6.

Table 6. Branch and Flow Control Instruction

Instruction Category	Instruction Name	Options
Branch (**b***x*, **bc***x*)	Branch	Address, and link, conditional, conditional to link register, conditional to count register, if less than, if not less than, if less than or equal to, if equal to, if not greater than, if greater than, if greater than or equal to, if summary overflow, if not summary overflow, if unordered, if not unordered
Condition register logical (**cr***x*, **mcr***x*)	Condition register	AND, OR, XOR, NAND, NOR, Equivalent, AND with complement, OR with complement
Trap (**t***x*, **tw***x*)	Trap	Word, immediate, less than, not less than, equal, less than or equal, not equal, not greater than, greater than, greater than or equal, logically less than, logically not less than, logically less than or equal, logically not greater than, logically greater than, logically greater than or equal
System call (**sc**)	System call	—
Return (**rf***x*)	Return from	Interrupt, critical interrupt

Processor Control Instructions

Processor control instructions are used to read from and write to the CR and XER at the user level, as well as the machine state register (MSR) and most special-purpose registers (SPRs) at the supervisor level. The time base register and some SPRs are accessible at both the user level and supervisor level, in which case separate SPR numbers are used for each.

Processor control instructions are shown in Table 7.

Table 7. Processor Control Instructions

Instruction Category	Instruction Name	Options
Move (**mt***x*, **mf***x)*	Move to	SPR, condition register fields, condition register from XER, time base, MSR, PMR (in performance monitor APU)
	Move from	Special-purpose register, condition register, time base, MSR
Write (**wrtee***x*)	Write MSR external enable	Immediate (Book E only)

Memory Synchronization Instructions

Memory synchronization instructions control the order in which memory operations execute with respect to asynchronous events and the order in which operations are seen by other mechanisms that access memory. Memory synchronization instructions are user-level instructions and are shown in Table 8.

Table 8. Memory Synchronization Instructions

Instruction Category	Instruction Name	Options
Load word and reserve index	Load word	**lwarx**
Store word conditional index	Store word	**stwcx.**
Synchronize (**sync**, **eieio**, **isync**, **msync**, **mbar**)	Synchronize	(Classic PowerPC only)
	Enforce In-Order Execution of I/O	(Classic PowerPC only)
	Memory Synchronize	(Book E only)
	Memory Barrier	(Book E only)
	Instruction Synchronize	—

Memory Control Instructions

Memory control instructions include instructions for cache management, segment register management, and TLB management.

- Cache instructions—Help programs manage on-chip caches if they are implemented. The effects of the cache management instructions on memory are weakly ordered. If the programmer needs to ensure that cache or other instructions have been performed with respect to all other processors and system mechanisms, a **sync** (classic) or **msync** (Book E) must be placed in the program following those instructions.
- Segment register management instructions—Provide access to the segment registers.
- TLB management instructions—The address translation mechanism is defined in terms of segment descriptors and page table entries (PTEs) used to locate the logical-to-physical address mapping for a particular access. These segment descriptors and PTEs reside in segment tables and page tables in memory, respectively.

> For performance reasons, many processors implement one or more TLBs on-chip. These are caches of portions of the page table. As changes are made to the address translation tables, it is necessary to maintain coherency between the TLB and the updated tables. This is done by invalidating TLB entries, or occasionally by invalidating the entire TLB, and allowing the translation caching mechanism to refetch from the tables.

Memory control instructions are listed in Table 9.

Table 9. Memory Control Instructions

Instruction Category	Instruction Name	Options
User-level cache (**dcb***x*, **icb***x*)	Data cache block	Touch, touch for store, allocate, clear to zero, store, flush
	Instruction cache block	Invalidate, touch (Book E only)
Supervisor-level cache (**dcbi**)	Data cache block	Invalidate
Segment register (**mtsr***x*)	Move to segment register	Indirect (classic PowerPC only)
	Move from segment register	Indirect (classic PowerPC only)
TLB management (**tlb***x*)	TLB invalidate	Entry, all, virtual address indexed
	TLB synchronize	—
	TLB read entry	(Book E only)
	TLB search indexed	(Book E only)
	TLB write entry	(Book E only)

ISA Extensions—Auxiliary Processing Units (APUs)

The EIS defines many APUs that extend the programming and interrupt models. APUs may consist of any combination of instructions, optional behavior of Book E–defined instructions, registers, register files, fields within Book E–defined registers, interrupts, or exception conditions within Book E–defined interrupts. This section briefly describes EIS-defined APUs that substantially extend the ISA.

AltiVec APU

Although it precedes Book E and its definition of APUs, the AltiVec technology provides a prototype for the current assortment of APUs in that it extends the instruction set, register, and interrupt models. It not only makes use of architecture-defined resources, it creates a 32-entry vector register (VR) file similar to the GPRs and FPRs, but widened to 128 bits to accommodate the multiple-element vector operands. For more information about the AltiVec APU, see "Architectural Extensibility—AltiVec Technology," on page 12.

Integer Select APU

The Integer Select instruction, **isel**, can be used to eliminate short conditional branch segments. **isel** has two source registers and one destination register. Under the control of a specified condition code bit, it copies one or the other source operand to the destination. This APU consists of a single instruction.

Signal Processing Engine (SPE) APU

The SPE APU is designed to accelerate signal processing applications normally suited to DSP operation. This is accomplished using short (two-element) vectors within 64-bit GPRs and using single instruction multiple data (SIMD) operations to perform the requisite computations. SPE also architects an accumulator register to allow for back-to-back operations without loop unrolling.

The SPE APU supports a number of forms of multiply and multiply-accumulate operations, and of add and subtract to accumulator operations. The SPE supports signed and unsigned forms and optional fractional forms. For these instructions, the fractional form does not apply to unsigned forms because integer and fractional forms are identical for unsigned operands.

Mnemonics for SPE instructions generally begin with the letters ‘ev’ (embedded vector).

Table 10 shows how SPE APU vector multiply instruction mnemonics are structured.

Table 10. SPE APU Vector Multiply Instruction Mnemonic Structure

Prefix	Multiply Element		Data Type Element		Accumulate Element (Optional)	
evm	**ho**	half odd (16x16->32)	**usi**	unsigned saturate integer	**a**	write to ACC
	he	half even (16x16->32)	**umi**	unsigned modulo integer	**aa**	write to ACC & added ACC
	hog	half odd guarded (16x16->32)	**ssi**	signed saturate integer	**an**	write to ACC & negate ACC
	heg	half even guarded (16x16->32)	**ssf**[1]	signed saturate fractional	**aaw**	write to ACC & ACC in words
	wh	word high (32x32->32)	**smi**	signed modulo integer	**anw**	write to ACC & negate ACC in words
	wl	word low (32x32->32)	**smf**[1]	signed modulo fractional		
	whg	word high guarded (32x32->32)				
	wlg	word low guarded (32x32->32)				
	w	word (32x32->64)				

1 Low word versions of signed saturate and signed modulo fractional instructions are not supported. Attempting to execute an opcode corresponding to these instructions causes boundedly undefined results.

Embedded Vector and Scalar Single-Precision Floating-Point APUs (SPFP APUs)

The EIS defines the following three closely related embedded floating-point APUs:

- Single-precision scalar
- Single-precision vector
- Double-precision scalar

These APUs provide IEEE-compatible floating-point operations to power- and space-sensitive embedded applications. Rather than implementing the FPRs defined by the PowerPC architecture, these embedded floating-point instructions share the GPRs, extending them to 64-bits in the case of the vector single-precision and scalar double-precision APUs. These extended GPRs are described in “Register Files,” on page 29. Because the single-precision vector and double-precision scalar APUs require 64-bit operands, they use the double-word load and store instructions defined by the SPE APU.

Register Model

The PowerPC architecture defines register-to-register operations for all computational instructions. Source data for these instructions is accessed from the on-chip registers or is provided as immediate values embedded in the opcode. The PowerPC architecture allows specification of a target register distinct from the two source registers, preserving the original data for use by other instructions and reducing the number of instructions for some operations. Data is transferred between memory and registers with explicit load and store instructions only.

Many of the registers implemented in the classic and Book E versions of the PowerPC architectures are identical in name and function. Registers hold the source or destination of an instruction, or they are accessed as a by-product of execution.

- Register files. General-purpose registers (GPRs) and floating-point registers (FPRs) are accessed as either the source or destination of an instruction.
- Instruction-accessible registers. Registers such as the condition register (CR), the floating-point status and control register (FPSCR), and some SPRs are accessed as the by-products of executing certain instructions.
- Special-purpose registers (SPRs). SPRs are on-chip registers that are part of the processor core. They control the use of the debug facilities, timers, interrupts, memory management unit, and other processor resources. They include the hardware implementation-dependent registers (HIDs), not defined by the architecture, that are used for configuration and control. SPRs are accessed with the Move to SPR (**mtspr**) and Move from SPR (**mfspr**) instructions.

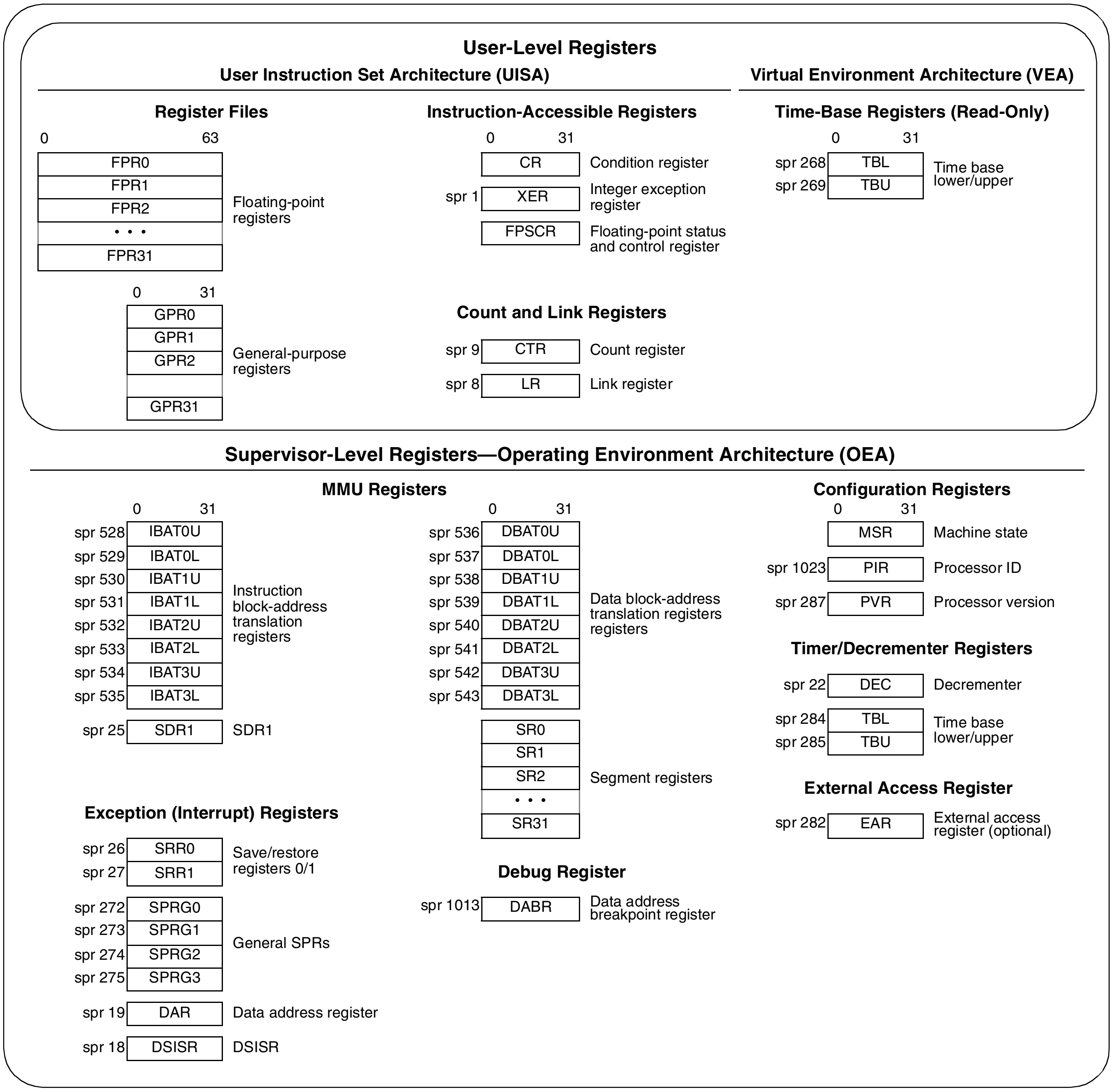
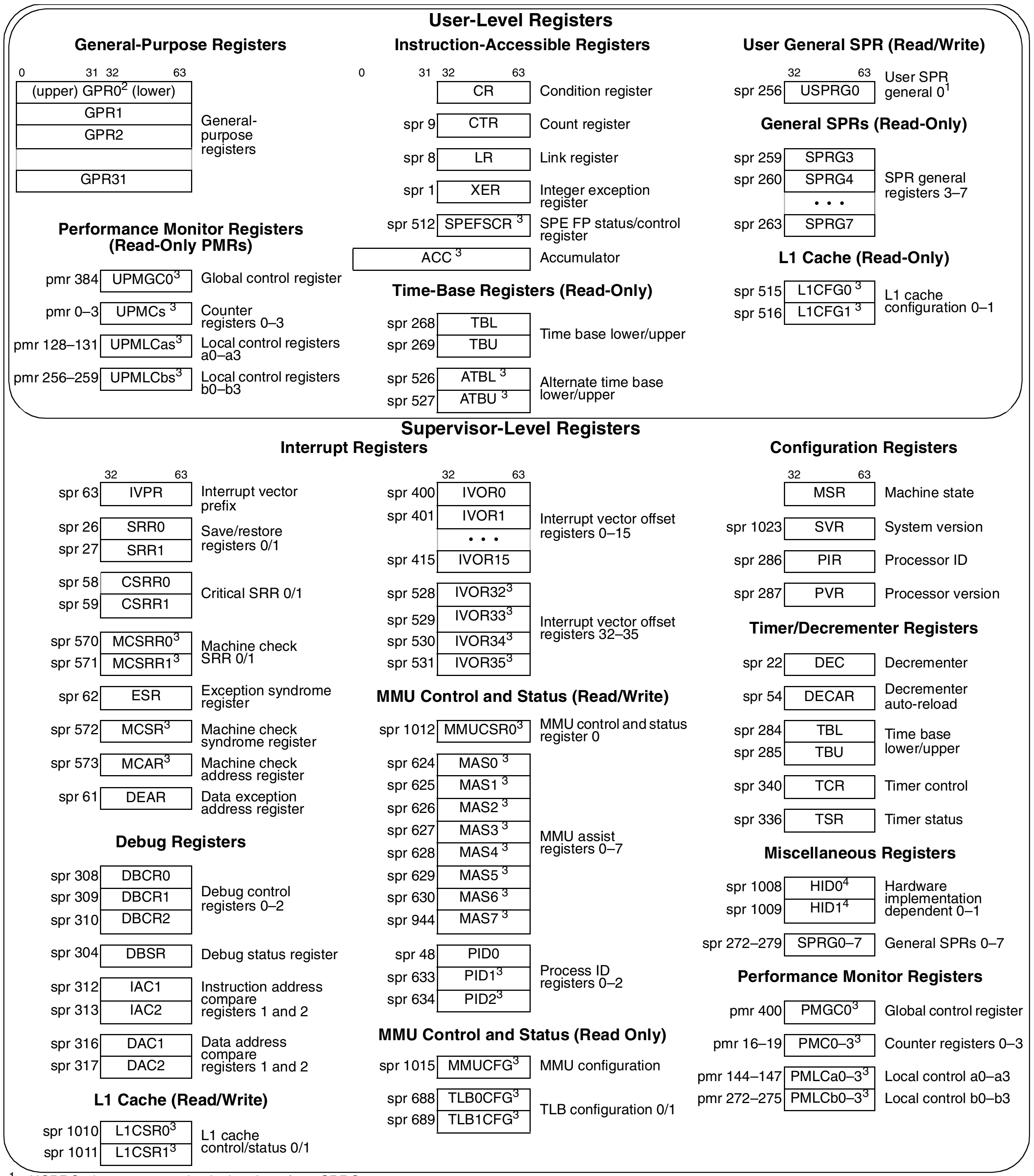


Figure 3. PowerPC Classic Register Model


User-Level Registers
General-Purpose Registers
0 31 32 63
(upper) GPR0[2] (lower)
GPR1
GPR2
GPR31
General-purpose registers
Instruction-Accessible Registers
0 31 32 63
CR Condition register
spr 9 CTR Count register
spr 8 LR Link register
spr 1 XER Integer exception register
spr 512 SPEFSCR[3] SPE FP status/control register
ACC[3] Accumulator
User General SPR (Read/Write)
32 63
spr 256 USPRG0 User SPR general 0[1]
General SPRs (Read-Only)
spr 259 SPRG3
spr 260 SPRG4
• • •
spr 263 SPRG7
SPR general registers 3–7
L1 Cache (Read-Only)
spr 515 L1CFG0[3]
spr 516 L1CFG1[3]
L1 cache configuration 0–1
Performance Monitor Registers (Read-Only PMRs)
pmr 384 UPMGC0[3] Global control register
pmr 0–3 UPMCs[3] Counter registers 0–3
pmr 128–131 UPMLCas[3] Local control registers a0–a3
pmr 256–259 UPMLCbs[3] Local control registers b0–b3
Time-Base Registers (Read-Only)
spr 268 TBL
spr 269 TBU
Time base lower/upper
spr 526 ATBL[3]
spr 527 ATBU[3]
Alternate time base lower/upper
Supervisor-Level Registers
Interrupt Registers
32 63
spr 63 IVPR Interrupt vector prefix
spr 26 SRR0
spr 27 SRR1
Save/restore registers 0/1
spr 58 CSRR0
spr 59 CSRR1
Critical SRR 0/1
spr 570 MCSRR0[3]
spr 571 MCSRR1[3]
Machine check SRR 0/1
spr 62 ESR Exception syndrome register
spr 572 MCSR[3] Machine check syndrome register
spr 573 MCAR[3] Machine check address register
spr 61 DEAR Data exception address register
32 63
spr 400 IVOR0
spr 401 IVOR1
• • •
spr 415 IVOR15
Interrupt vector offset registers 0–15
spr 528 IVOR32[3]
spr 529 IVOR33[3]
spr 530 IVOR34[3]
spr 531 IVOR35[3]
Interrupt vector offset registers 32–35
Debug Registers
spr 308 DBCR0
spr 309 DBCR1
spr 310 DBCR2
Debug control registers 0–2
spr 304 DBSR Debug status register
spr 312 IAC1
spr 313 IAC2
Instruction address compare registers 1 and 2
spr 316 DAC1
spr 317 DAC2
Data address compare registers 1 and 2
L1 Cache (Read/Write)
spr 1010 L1CSR0[3]
spr 1011 L1CSR1[3]
L1 cache control/status 0/1
MMU Control and Status (Read/Write)
spr 1012 MMUCSR0[3] MMU control and status register 0
spr 624 MAS0[3]
spr 625 MAS1[3]
spr 626 MAS2[3]
spr 627 MAS3[3]
spr 628 MAS4[3]
spr 629 MAS5[3]
spr 630 MAS6[3]
spr 944 MAS7[3]
MMU assist registers 0–7
spr 48 PID0
spr 633 PID1[3]
spr 634 PID2[3]
Process ID registers 0–2
MMU Control and Status (Read Only)
spr 1015 MMUCFG[3] MMU configuration
spr 688 TLB0CFG[3]
spr 689 TLB1CFG[3]
TLB configuration 0/1
Configuration Registers
32 63
MSR Machine state
spr 1023 SVR System version
spr 286 PIR Processor ID
spr 287 PVR Processor version
Timer/Decrementer Registers
spr 22 DEC Decrementer
spr 54 DECAR Decrementer auto-reload
spr 284 TBL
spr 285 TBU
Time base lower/upper
spr 340 TCR Timer control
spr 336 TSR Timer status
Miscellaneous Registers
spr 1008 HID0[4]
spr 1009 HID1[4]
Hardware implementation dependent 0–1
spr 272–279 SPRG0–7 General SPRs 0–7
Performance Monitor Registers
pmr 400 PMGC0[3] Global control register
pmr 16–19 PMC0–3[3] Counter registers 0–3
pmr 144–147 PMLCa0–3[3] Local control a0–a3
pmr 272–275 PMLCb0–3[3] Local control b0–b3

1 USPRG0 is a separate physical register from SPRG0.
2 Only SPE, double-precision and vector single-precision APU instructions can access the upper word of the 64-bit GPRs.
3 These registers are defined by the EIS and are not part of the Book E architecture.
4 Commonly implemented processor-specific registers

Figure 4. Book E Register Model

The PowerPC Book E user instruction and register models are fully binary-compatible with those of the classic PowerPC UISA. The Book E register model is shown in Figure 4. Book E defines all registers as 64-bit registers, although for 32-bit implementations, only the lower 32 bits are required to be implemented. For example, 32-bit accesses to GPRs use only bits 32–63.

The UISA registers, shown in Figure 3, can be accessed by user- or supervisor-level instructions; the VEA introduces the time base facility as user-level registers, also shown in Figure 3. The OEA defines the registers an operating system uses for memory management, configuration, and interrupt handling. The OEA register model includes only supervisor-level registers. The following describes specific registers for both classic and Book E architectures.

Register Files

Figure 5 shows a comparison of classic PowerPC and Book E PowerPC architecture register files.

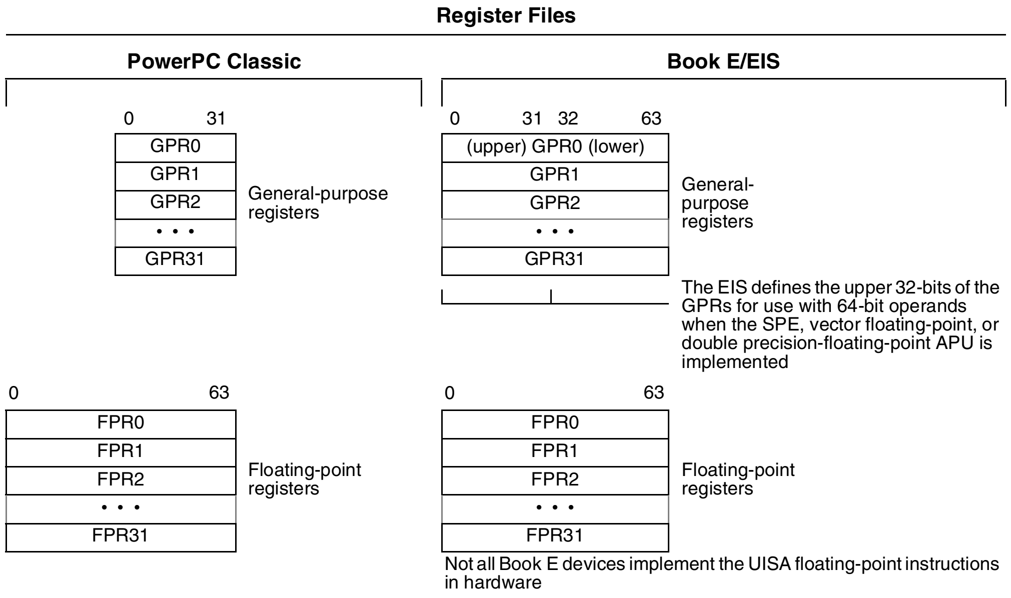

Register Files
PowerPC Classic
Book E/EIS
0
31
GPR0
GPR1
GPR2
• • •
GPR31
General-purpose registers
0
31
32
63
(upper) GPR0 (lower)
GPR1
GPR2
• • •
GPR31
General-purpose registers
The EIS defines the upper 32-bits of the GPRs for use with 64-bit operands when the SPE, vector floating-point, or double precision-floating-point APU is implemented
0
63
FPR0
FPR1
FPR2
• • •
FPR31
Floating-point registers
0
63
FPR0
FPR1
FPR2
• • •
FPR31
Floating-point registers
Not all Book E devices implement the UISA floating-point instructions in hardware

Figure 5. Register File Comparison

PowerPC classic and Book E both include GPR and FPR register files necessary for instruction computation:

- General-purpose registers (GPRs). GPRs are accessed to serve as the data source or destination for all integer and non-floating-point load/store instructions and provide data for generating addresses. The classic PowerPC architecture and Book E define a GPR file that consists of thirty-two 32-bit GPRs designated as GPR0–GPR31. The EIS defines a set of thirty-two 64-bit GPRs for use with certain APUs, such as the SPE APU and the vector single-precision floating-point APU.

  To provide room for expansion, the classic PowerPC architecture defines 32- and 64-bit addressing modes, which provide additional instructions for loading and storing double-word operands. Because many registers have to be wide enough to hold an address, the sizes of some register resources, including the GPRs, save restore register 0 (SRR0), and the data address register (DAR), are defined to be 32 bits wide in 32-bit implementations and 64 bits wide in 64 bit implementations.

- Floating-point registers (FPRs). FPRs are accessed to act as either the source or destination of a floating-point instructions. The classic PowerPC architecture includes an FPR file that consists of thirty-two 64-bit FPRs, FPR0–FPR31. These registers can contain data objects in either single- or double-precision floating-point format. Although Book E retains the floating-point instructions defined in the UISA, Book E implementations need not implement the floating-point ISA in hardware. Instead, they may implement an alternative floating-point mechanism or provide software emulation for floating-point instructions. This is significant in those processors targeted for the embedded markets because the absence of UISA-defined floating-point hardware saves the extra core space for FPRs and FPUs. When not implementing floating-point hardware, Book E implementations can emulate classic PowerPC floating-point or use embedded floating-point APUs.

  The FPRs use double-precision operand format for both single and double-precision data. See "Floating-Point Instructions," on page 19 for more information.

Instruction-Accessible Registers

Figure 6 shows a comparison of classic PowerPC and Book E PowerPC architecture instruction-accessible registers.

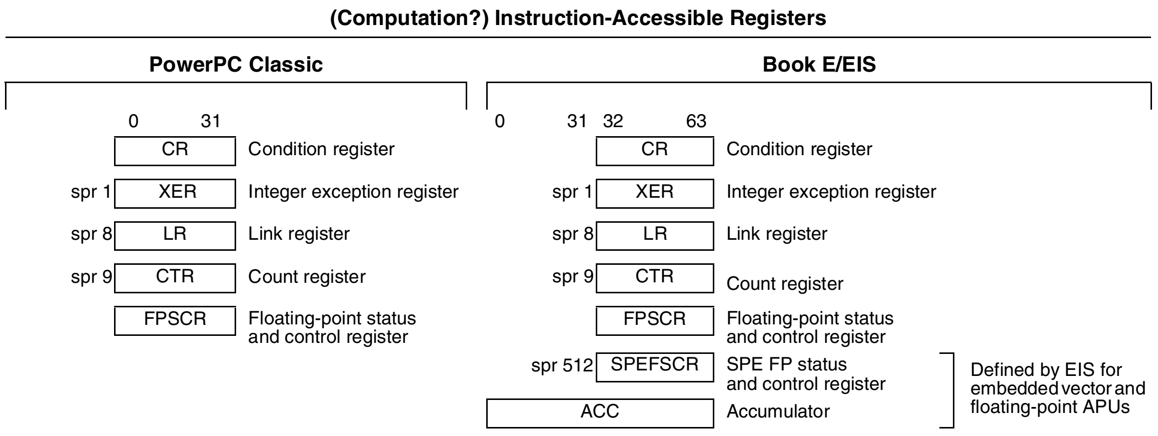

(Computation?) Instruction-Accessible Registers
PowerPC Classic
Book E/EIS
0
31
CR
Condition register
spr 1
XER
Integer exception register
spr 8
LR
Link register
spr 9
CTR
Count register
FPSCR
Floating-point status and control register
0
31
32
63
CR
Condition register
spr 1
XER
Integer exception register
spr 8
LR
Link register
spr 9
CTR
Count register
FPSCR
Floating-point status and control register
spr 512
SPEFSCR
SPE FP status and control register
ACC
Accumulator
Defined by EIS for embedded vector and floating-point APUs

Figure 6. Instruction-Accessible Registers Comparison

PowerPC classic and Book E registers contain instruction-accessible registers that can be accessed as the by-product of executing certain instructions:

- Condition register (CR). The CR reflects the results of testing and branching. It is used to record conditions such as overflows and carries. A specified CR field can be set as the result of either an integer or a floating-point compare instruction.
- Integer exception register (XER). The XER indicates overflow and carries for integer operations. XER status bits overflow, summary overflow, and carry are set based on the operation of an instruction considered as a whole, not on intermediate results. For example, the subtract from carrying instruction, which produces a result specified as the sum of three values, sets XER bits based on the entire operation, not on an intermediate sum.
- Link register (LR). The LR provides the branch target address for the branch conditional to link register (**bclr***x*) instructions and can optionally be used to hold the logical address (also called the

effective address) of the instruction that follows a branch and link instruction, typically used for linking to subroutines.

- Count register (CTR). The CTR can be used to hold a loop count, which can be decremented during execution of branch instructions. The entire count register can also be used to provide the branch target address for the branch conditional to count register (**bcctr***x*) instruction.
- Floating-point status and control register (FPSCR). The FPSCR controls the handling of floating-point exceptions and records status resulting from the floating-point operations. The register includes status bits and control bits needed for compliance with the IEEE 754 floating-point standard.

The EIS definition includes registers that support SPE and embedded floating-point instructions:

- SPEFSCR is used for status and control of SPE and embedded floating-point instructions. It controls the handling of floating-point exceptions and records status resulting from the floating-point operations.
- Accumulator register (ACC). The accumulator holds the results of the multiply accumulate (MAC) forms of SPE integer instructions. The accumulator allows back-to-back execution of dependent MAC instructions, something that is found in the inner loops of DSP code such as finite impulse response (FIR) filters. The accumulator is partially visible to the programmer in that its results do not have to be explicitly read to use them. Instead, they are always copied into a 64-bit destination GPR specified as part of the instruction. Based upon the type of instruction, this register can hold either a single 64-bit value or a vector of two 32-bit elements.

Time Base Registers

The VEA introduces the time base facility as user-level registers. Figure 7 shows a comparison of classic PowerPC and Book E PowerPC architecture time base registers.

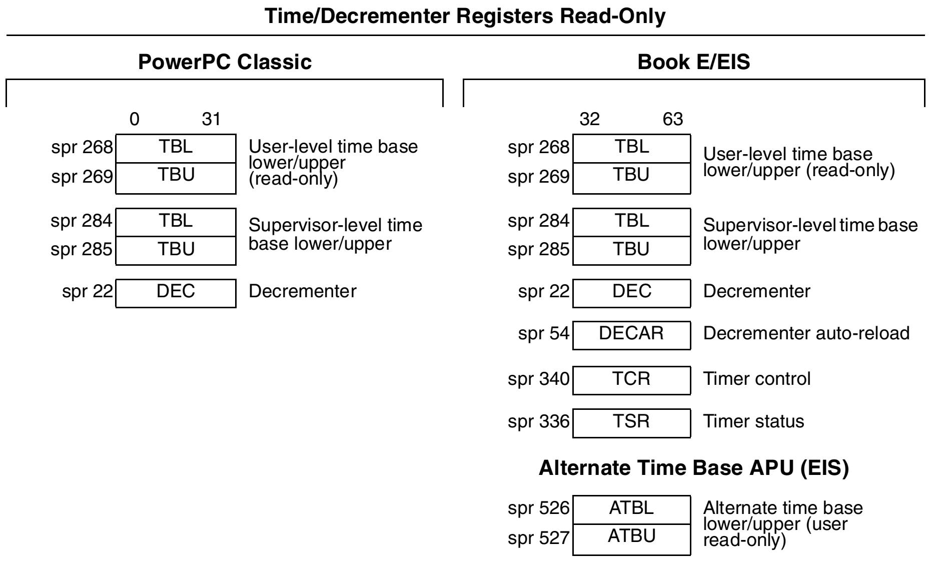

Time/Decrementer Registers Read-Only
PowerPC Classic
0
31
spr 268
TBL
spr 269
TBU
User-level time base lower/upper (read-only)
spr 284
TBL
spr 285
TBU
Supervisor-level time base lower/upper
spr 22
DEC
Decrementer
Book E/EIS
32
63
spr 268
TBL
spr 269
TBU
User-level time base lower/upper (read-only)
spr 284
TBL
spr 285
TBU
Supervisor-level time base lower/upper
spr 22
DEC
Decrementer
spr 54
DECAR
Decrementer auto-reload
spr 340
TCR
Timer control
spr 336
TSR
Timer status
Alternate Time Base APU (EIS)
spr 526
ATBL
spr 527
ATBU
Alternate time base lower/upper (user read-only)

Figure 7. Time/Decrementer Registers Comparison

Classic PowerPC and Book E registers include hardware and software timers:

- Time base (TBU and TBL). The time base (TB) provides timing functions for the system. TB is composed of two 32-bit registers, the time base upper (TBU) concatenated on the right with the time base lower (TBL). The two 32-bit TB registers count at an implementation-specific rate like a 64-bit counter. User-level applications have read-only access to the TB while supervisor-level applications have read/write access. The time base count is used, among other functions, to trigger interrupts.
- Decrementer register (DEC). The decrementer is typically used as a general-purpose software timer. It is updated at the same rate as the TB and provides a way to signal a decrementer interrupt after a specified period.

The Book E definition provides registers that incorporate timing mechanisms for the fixed-interval and watchdog timer interrupts defined in Book E:

- Decrementer auto-reload register (DECAR). The DECAR is used to automatically reload a programmed value into DEC. In the classic PowerPC architecture, a value has to be explicitly programmed into DEC.
- Timer control register (TCR). The TCR provides control information for the decrementer. It controls features such as auto-reload enable and decrementer interrupt enable. EIS adds bits to the TCR to allow the selection of any TB bit to be used to trigger a fixed-interval or watchdog timer interrupt.
- Timer status register (TSR). The TSR contains status on timer events and the most recent watchdog timer-initiated processor reset. It controls features such as watchdog timer and fixed-interval interrupt enable and watchdog timer interrupt status.

The EIS defines an alternate time base APU, which is implemented on the e500v2. It is intended for measuring time in implementation-defined intervals. It differs from the PowerPC-defined time base in that it is not writable, it counts at a different, typically higher, frequency, and it always counts up, wrapping when the 64-bit count overflows.

MMU Control and Status Registers

Because the classic PowerPC MMU specification was cumbersome for embedded applications, many PowerPC processors intended for the embedded environment freely broke from the MMU model to support such features as variable-sized pages and software-managed page tables. These features were added while preserving the integrity of the UISA and setting a model for the PowerPC Book E architecture that provides a standard for embedded controllers that is general enough to support the greater variety of needs of special-purpose embedded controllers.

The complexity of the modal 32-/64-bit MMU model in the classic PowerPC architecture provided impediments to portability and in Book E is replaced by additional registers and instructions in the UISA and slight modifications to existing instructions to extend the addressibility. The Book E definition results in a more embedded-friendly MMU architecture that is simpler and more flexible while implementing software-driven TLBs and per-page properties. Translation look-aside buffers (TLBs) keep recently-used page address translations on-chip. See "Memory Management Unit (MMU) Model," on page 44 for more information on the MMU.

Figure 8 shows a comparison of classic PowerPC and Book E PowerPC architecture MMU registers.

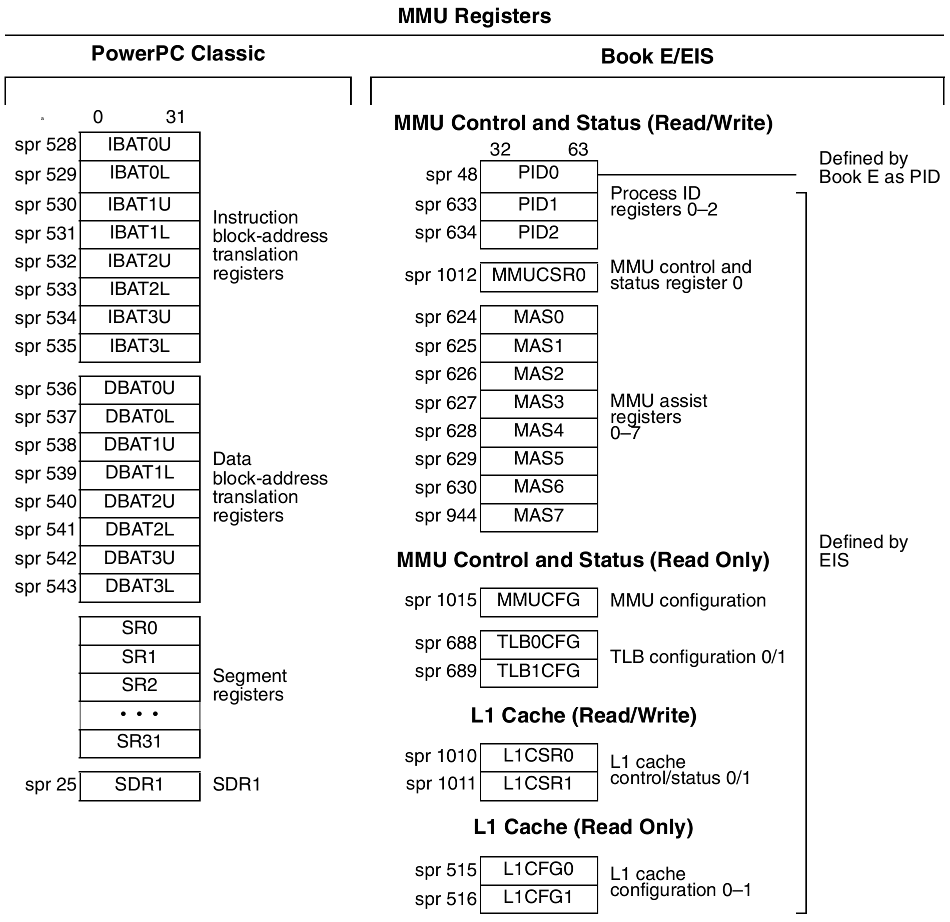


Figure 8. MMU Registers Comparison

The PowerPC classic definition includes SPRs that are used for address translation:

- Block address translation registers (IBATs and DBATs). The classic PowerPC MMU defines BAT registers (BATs) to maintain the address translation information for four blocks of instruction-related memory and four blocks of data-related memory. BATs are supported through the use of two independent instruction and data arrays called IBAT and DBAT arrays. These BAT registers define the base addresses and sizes of BAT areas as well as the value of the WIMG bits, which describe cache coherency attributes. Effective addresses are compared simultaneously with all entries in the BAT array during block translation.

  The BATs are maintained by the system software and are implemented as eight pairs of SPRs. Each block is defined by a pair of BATs. For example, IBAT0U and IBAT0L provide translation and protection for one block. BAT registers are not part of the PowerPC Book E specification. A more detailed discussion of how BATs function can be found in the section "Memory Management Unit (MMU) Model," on page 44.
- SDR1 register. SDR1 is a classic PowerPC register that specifies the base address and the size of the page tables in memory. When a table search operation commences, a primary hashing function is performed on the virtual address. The output of the hashing function is then concatenated with

bits programmed into SDR1 by the operating system to create the physical address of the primary page table entry group.

The Book E MMU implements a register to track effective address generation:

- Process ID register (PID). The Book E architecture specifies that a process ID (PID) value be associated with each effective address (instruction or data) generated by the processor. Book E defines one PID register that holds the PID value for the current process.

The EIS definition includes MMU assist (MAS) registers, among others, to provide MMU control:

- Process ID registers (PID1–PID2). PID1 and PID2 are part of EIS. PID values are used to construct virtual addresses for accessing memory. Note that in EIS, the PID register defined in Book E is known as PID0.
- MMU control and status register 0 (MMUCSR0). Used for general control of L1 and L2 MMUs.
- MMU assist registers. MMU assist registers (MAS0–MAS7). Used to configure and manage pages through translation look-aside buffers (TLBs). These registers are used to configure and control MMU read/write and replacement, descriptor configuration, effective page number and page attributes, real page number and access, and hardware replacement assist configuration. MAS7 is provided to support implementations with virtual addresses greater than 32 bits. The e500v2 uses MAS7 to support 36-bit addressing.
- MMU configuration register (MMUCFG). Provides configuration information for the particular MMU supplied with a version of the core. It is a read-only register that provides information on PID register size and the number of TLBs.
- TLB configuration registers (TLB0CFG–TLB1CFG). These read-only registers provide information about each TLB that is visible to the programming model. They provide configuration information for TLBs and describe aspects such as the associativity, minimum and maximum page sizes of the TLBs, and the number of entries in the TLBs.

L1 Cache Registers

The EIS defines L1 cache configuration and status registers, shown in Figure 9. Neither Book E nor the classic PowerPC architecture defines L1 cache registers.

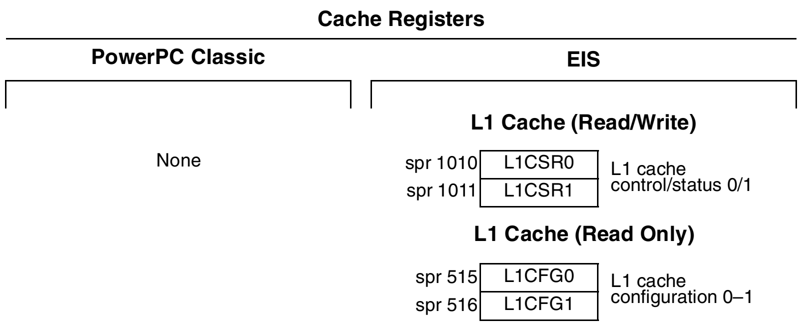


Figure 9. Cache Registers Comparison

The EIS defines the following registers for L1 cache configuration, status, and control:

- L1 cache configuration registers (L1CFG0–L1CFG1). Read-only registers that provide configuration information for the particular L1 data and instruction caches supplied with a version of the core. They include a description of the cache block size, the number of ways, the cache size, and the cache replacement policy, among other features.
- L1 cache control and status registers (L1CSR0–L1CSR1). L1CSRs are used for general control and status of the L1 data and instruction caches and are read/write accessible by supervisor-level programs. They allow the programmer to enable features such as cache parity and the cache itself. They provide status on information such as cache locking and cache locking overflow.

Interrupt Registers

When interrupts occur, information about the state of the processor is saved to certain registers and the processor begins execution at an address (interrupt vector) predetermined for each interrupt. In the classic PowerPC architecture, this interrupt vector consists of a fixed offset prepended with a value as determined by MSR[IP]. Processing of interrupts begins in supervisor mode.

The Book E specification includes features that provide a more agile interrupt model that is required by embedded applications. Certain interrupt types demand immediate attention, even if other interrupt types are currently being processed and have not yet had the opportunity to save the state of the machine. This type of interrupt is new to Book E and is called a critical interrupt. The Book E specification supports faster context switches and the potential for better real-time performance than the classic PowerPC architecture. See the section “Interrupt Model” on page 40 for more information.

Figure 10 shows a comparison of classic PowerPC, Book E, and EIS interrupt registers.

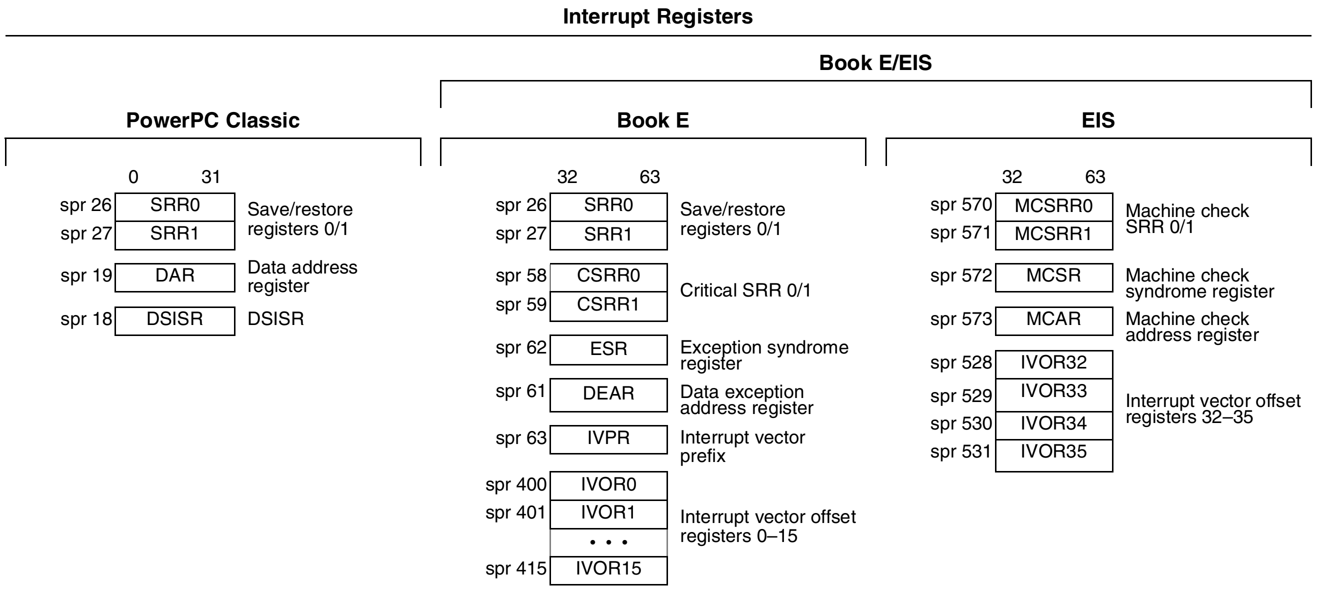


Figure 10. Interrupt Register Comparison

Save/restore registers are automatically updated with machine state information and the return address when an interrupt is taken. The classic PowerPC architecture defines only SRR0 and SRR1. Book E defines critical interrupts and the EIS defines additional interrupt types that use similar resources. These registers are described below.

Both classic PowerPC and Book E define save/restore registers:

- Save/restore registers (SRR0 and SRR1)
  - SRR0 holds the address of the instruction where an interrupted process should resume. For instruction-caused interrupts, it is typically the address of the instruction that caused the interrupt. When **rfi** executes, instruction execution continues at the address in SRR0. In Book E, SRR0 is used for non-critical interrupts.
  - SRR1 holds machine state information. When an interrupt is taken, MSR contents are placed in SRR1. When **rfi** executes, SRR1 contents are placed into MSR. In Book E, SRR1 is used for non-critical interrupts.

The PowerPC classic definition accounts for DSI (data storage interrupt) and alignment exceptions in its register model:

- Data address register (DAR). The effective address generated by a memory access instruction is placed in the DAR if the access causes an exception (for example, an alignment exception). This register is not supported in Book E implementations.
- DSISR. The DSISR identifies the cause of DSI and alignment exceptions. It is not supported in Book E implementations, although much of its functionality is supported in the ESR.

Book E provides greater flexibility in specifying vectors through the implementation of the interrupt vector prefix register (IVPR) and interrupt-specific interrupt vector offset registers (IVORs):

- Critical save/restore registers (CSRR0 and CSRR1). Defined to save and restore machine state on Book E–defined critical interrupts (critical input, machine check, watchdog timer, and debug) and are analogous to SRR0 and SRR1.
- Exception syndrome register (ESR). The ESR provides a way to differentiate between exceptions that can generate an interrupt type.
- Data exception address register (DEAR). Loaded with the effective address of a data access (caused by a load, store, or cache management instruction) that results in an alignment, data TLB miss, or DSI exception.
- Interrupt vector prefix register (IVPR). Used with IVORs to determine the vector address. The 16-bit vector offsets are concatenated to the right of IVPR to form the address of the interrupt routine.
- Interrupt vector offset registers (IVOR0–IVOR15). IVORs provide the index from the base address provided by the IVPR for its respective interrupt type. IVORs provide storage for specific interrupts. Book E allows implementations to define additional IVORs to support implementation-specific interrupts. For example, the SPE defines IVOR32–IVOR35. EIS-defined IVORs are listed at the bottom of Table 11.

Although Book E defines the machine check interrupt as a critical interrupt, the EIS defines the machine check APU, which, if implemented, treats the machine check interrupt as its own interrupt type. The machine check APU defines the following registers:

- Machine check save/restore registers (MCSRR0 and MCSRR1). Defined to save machine state on EIS-defined machine check interrupts and are analogous to SRR0 and SRR1.
- Machine check syndrome register (MCSR). When the core complex takes a machine check interrupt, it updates MCSR to differentiate between machine check conditions. The MCSR indicates whether a machine check condition is recoverable.
- Machine check address register (MCAR). When the core complex takes a machine check interrupt, it updates MCAR to indicate the address of the data associated with the machine check.

Configuration Registers

The PowerPC architecture define registers that provide control and configuration and status information of the machine state and process IDs. Figure 11 shows a comparison of classic PowerPC and Book E PowerPC architecture configuration registers.

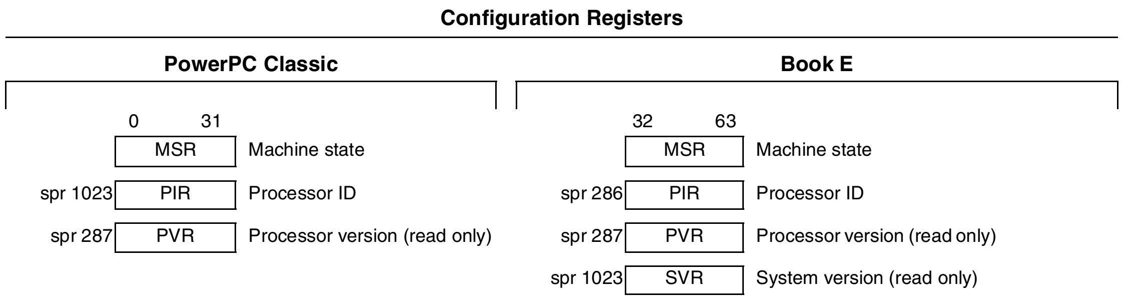


Figure 11. Configuration Registers Comparison

PowerPC classic and Book E both define the following registers:

- Machine state register (MSR). Defines the state of the processor (that is, enabling and disabling of interrupts and debugging exceptions, enabling and disabling some APUs, and specifying whether the processor is in supervisor or user mode).

  The classic PowerPC MSR supports bits that enable data address translation (IR and DR) and modal big/little endian byte ordering (LE and ILE). The Book E and EIS definitions do not support modal big/little endian byte ordering or real mode (IR=0 and DR=0).

  The MSR provides enable bits for machine check, external, and critical interrupts. MSR contents are automatically saved, altered, and restored by the interrupt-handling mechanism. If an interrupt is taken (a non-critical interrupt in Book E), MSR contents are automatically copied into SRR1. In Book E, if a critical interrupt is taken, MSR contents are automatically copied into CSRR1. When an **rfi** or **rfci** is executed, MSR contents are restored from SRR1 or CSRR1.
- Processor ID register (PIR). Contains a value that can be used to distinguish the processor from other processors in the system. Note that the classic PowerPC and Book E PIR SPR numbers differ.
- Processor version register (PVR). Contains a value identifying the version and revision level of the processor. The PVR distinguishes between processors whose attributes may affect software.

The Book E definition defines the SVR to support embedded cores as part of a system-on-a-chip (SoC):

- System version register (SVR). Contains a read-only SoC-dependent value that identifies the version of the SoC that incorporates the core.

Performance Monitor Registers (PMRs)

The EIS defines a set of register resources, shown in Figure 12, used exclusively by the performance monitor APU. PMRs are similar to the SPRs defined in the Book E architecture and are accessed by **mtpmr** and **mfpmr**, which are also part of the performance monitor APU. User-level PMRs are read-only.

Although the PowerPC architecture does not define performance monitor registers, most PowerPC processors implement a performance monitor that uses implementation-specific SPRs that are very similar to the EIS-defined PMRs.

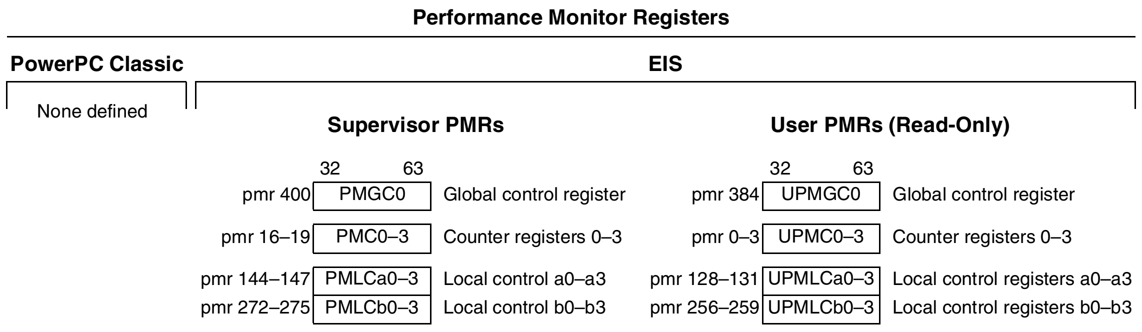


Figure 12. Performance Monitor Registers Comparison

All PMRs are defined under the EIS specification:

- Global control register (PMGC0/UPMGC0). PMGC0 controls all performance monitor counters and is a supervisor-level register. The contents of PMGC0 are reflected to UPMGC0, which is read by user-level software.
- Performance monitor counter registers (PMC0–PMC3/UPMC0–UPMC3). PMC0–PMC3 are 32-bit counters that can be programmed to generate interrupt signals when they overflow. Each counter is enabled to count 128 events. The contents of PMC0–PMC3 are reflected to UPMC0–UPMC3, which are read by user-level software.
- Local control registers are included in the EIS definition to facilitate software control of the PMRs:
  - PMLCa0–PMLCa3/UPMLCa0–UPMLCa3. PMLCa registers function as event selectors and give local control for the corresponding performance monitor counters. Each PMLCa works with the corresponding PMLCb register.

    The contents of PMLCa0–PMLCa3 are reflected to UPMLCa0–UPMLCa3, which are read by user-level software and are read-only.
  - PMLCb0–PMLCb3/UPMLCb0–UPMLCb3. PMLCb registers specify a threshold value and a multiple to apply to a threshold event selected for the corresponding performance monitor counter. Each PMLCb works with the corresponding PMLCa.

    The contents of PMLCb0–PMLCb3 are reflected to UPMLCb0–UPMLCb3, which are read by user-level software.

Debug Registers

Debug registers are accessible to software running on the processor. These registers are intended for use by special debug tools and debug software, and not by general application or operating system code. Figure 13 shows a comparison of classic PowerPC and Book E PowerPC architecture debug registers.

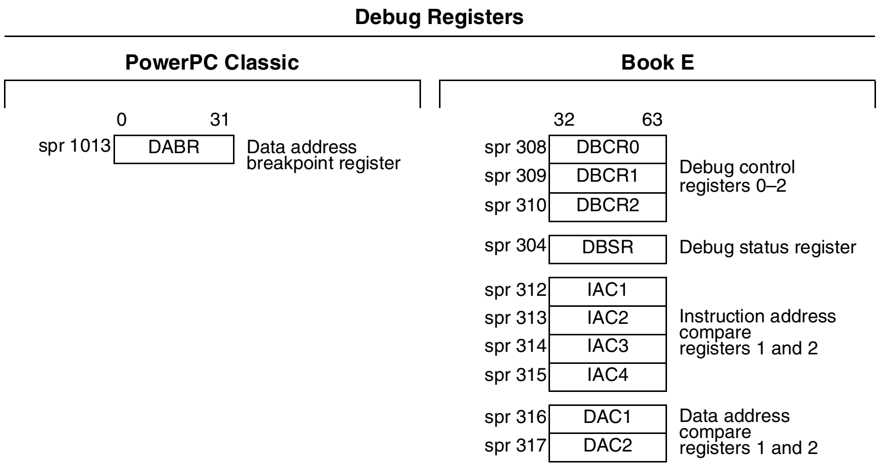


Figure 13. Debug Registers Comparison

The classic PowerPC definition provides one register to facilitate debugging:

- Data address breakpoint register (DABR). The data address breakpoint facility provides a means to detect accesses to a designated word. A data address breakpoint match is detected for a load or store instruction and a match generates a DSI exception. The address comparison is done on an effective address, and it applies to data accesses only.

The Book E definition provides debugging support at data and instruction addresses:

- Debug control registers (DBCR0–DBCR1). Enable debug events, reset the processor, control timer operation during debug events, and set the debug mode of the processor.
- Debug status register (DBSR). Provides status information for debug events and for the most recent processor reset. The DBSR is set through hardware but is read and cleared through software.
- Instruction and data address compare registers (IAC1–IAC4, DAC1–DAC2). A debug event may be enabled to occur upon an attempt to execute an instruction or access a data location from an address specified in an IAC/DAC, inside or outside a range specified by the IACs/DACs, or to blocks of addresses specified by the combination of the IACs/DACs.

Implementation-Specific Registers

Implementations typically have additional SPRs not defined by the architecture to handle special functions, and some of these registers may appear on multiple implementations with similar functionality. In particular, implementations define hardware implementation-dependent registers (HIDs) that typically control hardware-related functionality.

Interrupt Model

Most of the features of the PowerPC interrupt model are common to both the Book E and classic definitions. The PowerPC interrupt mechanism allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When interrupts occur, information about the state of the processor is saved to certain registers and the processor begins execution at an address (interrupt vector) predetermined for each interrupt. Processing of interrupts begins in supervisor mode.

Exception conditions may be defined at non-supervisor levels of the architecture. For example, the user instruction set architecture defines conditions that may cause floating-point exceptions; the OEA defines, at the supervisor level, the mechanism by which the interrupt is taken.

The PowerPC architecture differentiates between the terms 'exception' and 'interrupt.' Use of these terms in this document are as follows:

- An exception is the event that, if enabled, causes the processor to take an interrupt. The PowerPC architecture describes exceptions as being generated by signals from internal and external peripherals, instructions, the internal timer facility, debug events, or error conditions.
- An interrupt is the action a processor takes in response to an exception. The processor saves its context (typically the machine state register (MSR) settings and return instruction address) and begins execution at a predetermined interrupt handler address with a modified MSR.

The PowerPC architecture requires that interrupts be taken in program order; therefore, although a particular implementation may recognize exception conditions out of order, they are handled strictly in order with respect to the instruction stream. When an instruction-caused interrupt is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that have not yet entered the execute state, are required to complete before the interrupt is taken.

Interrupts can occur while an interrupt handler routine is executing, and multiple interrupts can become nested. It is up to the interrupt handler to save the appropriate machine state if it is desired to allow control to ultimately return to the interrupting program.

In many cases, after the interrupt handler handles an interrupt, there is an attempt to execute the instruction that caused the interrupt. Instruction execution continues until the next exception condition is encountered. This method of recognizing and handling interrupts sequentially guarantees that the machine state is recoverable and processing can resume without losing instruction results.

To prevent the loss of state information, interrupt handlers must save the information stored in save/restore registers (SRR0 and SRR1) (the return address and the MSR settings), soon after the interrupt is taken to prevent this information from being lost due to another interrupt being taken.

All interrupts except some machine check interrupts are recoverable. The conditions that cause a machine check may prohibit recovery.

Because multiple exception conditions can map to a single interrupt, a more specific condition may be determined by examining a register associated with the exception—for example, in classic PowerPC, the DSISR and the floating-point status and control register (FPSCR). Additionally, certain exception conditions can be explicitly enabled or disabled by software.

Invocation of an interrupt is precise, unless one of the imprecise modes for invoking the floating-point enabled exception-type program interrupt is in effect. When the interrupt is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts (because the invocation of the interrupt required to complete execution has not occurred).

Interrupts, their offsets, and conditions that cause them for the classic and Book E implementations of the PowerPC architecture, are summarized in Table 11. The Book E implementation of the PowerPC architecture uses interrupt vector offset registers (IVORs) to handle each interrupt type whereas the classic PowerPC implementation uses fixed-location vector offsets. Unless otherwise specified, MSR settings and the return address for every interrupt are stored in SRR0 and SRR1.

- Interrupts in the classic PowerPC definition—The classic PowerPC interrupt model differs from that of PowerPC Book E in that the classic PowerPC definition uses fixed addresses as vector offsets to map to physical memory locations. In the classic PowerPC definition, the base address is determined by the MSR[IP] bit. If IP is cleared, exception vector offsets are added to the physical address 0x000n_nnnn. If IP is set, exception vector offsets are added to the physical address 0xFFFn_nnnn. Table 11 shows the vector offsets associated with each interrupt type. Finally, classic PowerPC includes the system reset, trace, and floating-point assist interrupts as part of its definition.
- Interrupts in the Book E definition. The PowerPC Book E definition includes interrupt vector offset registers (IVORs), interrupt vector prefix registers (IVPRs), and critical interrupts as part of its definition. An IVOR is assigned to each interrupt type. The IVPR provides the base address location to which the offset in the IVORs is added. Table 11 shows the IVORs associated with each interrupt type.

  PowerPC Book E defines the following two interrupt types:

  — Noncritical interrupts—First-level interrupts typically encountered in routine operations. The implementation of these interrupts are largely identical to those defined by the OEA; they use save and restore registers (SRR0/SRR1) to save the return address and machine state when they are taken, and they use the **rfi** instruction to restore state.

  — Critical interrupts—Higher priority interrupts that use separate save/restore resources analogous to those defined for noncritical interrupts. These consist of the critical save and restore registers (CSRR0/CSRR1) and the Return from Critical Interrupt instruction **rfci** instruction to restore state. Critical interrupts are not part of the classic PowerPC interrupt definition, although some classic PowerPC implementations support critical interrupts and the CSRR registers.

  Book E defines the critical input, watchdog timer, debug, and machine check interrupts as critical interrupts.
- The EIS extends the notion introduced by the critical type interrupt of having a separate set of save/restore resources for allowing certain interrupts to be taken without having to encounter the latency of saving the machine state from the shared save/restore registers. The EIS defines two such interrupts:

  — The machine check interrupt APU in the e500 core implements save and restore registers (MCSRR0/MCSRR1) that are used to save the return address and machine state when machine check interrupts are taken, and they use the **rfmci** instruction to restore state.

— The debug interrupt APU in the e200 core implements save and restore registers (DSRR0/DSRR1) that are used to save the return address and machine state when debug interrupts are taken, and they use the **rfdi** instruction to restore state.

In addition, other APUs, such as the SPE and embedded floating-point define non-critical interrupts to handle APU-specific program interrupts.

Table 11 shows the interrupts as they are defined by the classic PowerPC architecture, Book E, and the EIS.

Table 11. Interrupts and Conditions—Overview

Interrupt Type	Vector Offset		Causing Conditions
	Classic	Book E	
Critical input	—	IVOR0	Typically caused by assertion of an asynchronous signal, is presented to the interrupt mechanism. Similar to external interrupts.
System reset	0x100	—	Caused by implementation-defined asynchronous conditions.
Machine check	0x200	IVOR1	Causes are implementation-dependent, but typically are related to conditions such as bus parity errors or attempts to access an invalid physical address. Typically, these interrupts are triggered by an input signal to the processor. Disabled when MSR[ME] = 0. If a machine check interrupt condition exists and ME is cleared, the processor goes into checkstop. **Classic PowerPC-specific features:** MSR settings and return address are stored in SRR0 and SRR1. **Book E–specific features:** Critical interrupt, uses CSRR0 and CSRR1 and Return from Critical Interrupt (**rfci**). The ESR reports the cause of the machine check interrupt. **EIS-specific features:** Defines an optional machine check APU that, when implemented, uses MCSRR0 and MCSRR1 and the Return from Machine Check Interrupt instruction (**rfmci**). An address related to the machine check may be stored in MCAR. MCSR reports the cause of the machine check.
DSI	0x300	IVOR2	A data memory access cannot be performed. Such accesses can be generated by load/store instructions and by certain memory control and cache control instructions. **Classic PowerPC-specific features:** DSISR reports the cause; DAR is set based on DSISR settings. **Book E–specific features:** ESR reports the cause; DEAR holds the effective address of the data access.
ISI	0x400	IVOR3	Instruction fetch cannot be performed. Causes include the following: • The effective address cannot be translated. For example, when there is a page fault for this portion of the translation, an ISI interrupt must be taken to retrieve the page (and possibly the translation), typically from a storage device. • An attempt is made to fetch an instruction from a no-execute segment or from guarded memory when MSR[IR] = 1. • The fetch access violates memory protection. **Book E–specific features:** Book E also uses ISI to assist implementations that cannot dynamically switch byte ordering between consecutive accesses, do not support the byte order for a class of accesses, or do not support misaligned accesses using a specific byte order. ESR reports the cause.
External interrupt	0x500	IVOR4	Generated only when an external interrupt is pending (typically signalled by a signal defined by the implementation) and the interrupt is enabled (MSR[EE]=1).

Table 11. Interrupts and Conditions—Overview (continued)

Interrupt Type	Vector Offset		Causing Conditions
	Classic	Book E	
Alignment	0x600	IVOR5	The processor cannot perform a memory access for one of these reasons: • The operand of a load or store is not aligned. • The instruction is a move assist, load multiple, or store multiple. • A **dcbz** operand is in write-through-required or caching-inhibited memory, or **dcbz** is executed in an implementation with no data cache or a write-through data cache. • The operand of a store, except store conditional, is in write-through required memory. **Classic PowerPC-specific features:** DSISR reports the interrupt cause; DAR is set based on DSISR settings. **Book E–specific features:** ESR reports the interrupt cause; DEAR holds the effective address of the data access. **EIS-specific features:** EIS defines additional exception conditions.
Program	0x700	IVOR6	One of the following conditions occurs during instruction execution: • Floating-point enabled exception—Generated when MSR[FE0,FE1] ≠ 00 and FPSCR[FEX] is set. • Illegal instruction— Generated when execution of an instruction is attempted with an illegal opcode or illegal combination of opcode and extended opcode fields or when execution of an optional instruction not provided in the specific implementation is attempted. (These do not include those optional instructions that are treated as no-ops.) • Privileged instruction—User-level code attempts execution of a supervisor instruction. • Trap—Any of the conditions specified in a trap instruction is met. **Classic PowerPC-specific features:** Caused when a floating-point instruction causes an enabled exception or by the execution of a Move to FPSCR instruction that sets both an exception condition bit and its corresponding FPSCR enable bit.DSISR reports the cause of the program interrupt; DAR is set based on DSISR settings. **Book E–specific features:** An unimplemented operation exception may occur if an unimplemented defined or allocated instruction is encountered. Otherwise an illegal instruction interrupt occurs. ESR reports the cause.
Floating-point unavailable	0x800	IVOR7	Caused by an attempt to execute a floating-point instruction (including floating-point load, store, and move instructions) when the floating-point available bit is cleared, MSR[FP] = 0.
Decrementer	0x900	IVOR10	The most-significant DEC bit changes from 0 to 1 and the interrupt is enabled (MSR[EE] = 1). If it is not enabled, the interrupt remains pending until it is taken. **Book E–specific features:** The TSR records status on timer events. An auto-reload value in the DECAR is written to DEC when it decrements from 0x0000_0001 to 0x0000_0000.
System call	0xC00	IVOR8	Occurs when a System Call (**sc**) instruction is executed.
Trace	0xD00	—	Optional. Either MSR[SE] = 1, almost any instruction successfully completed, or MSR[BE] = 1, and a branch instruction is completed. See the debug interrupt (IVOR 15) for a description of Book E trace support.
Floating-point assist	0xE00	—	Optional. Can be used to provide software assistance for infrequent and complex floating-point operations such as denormalization.
Auxiliary processor unavailable	—	IVOR9	An attempt is made to execute an APU instruction (including loads, stores, and moves), the target APU is implemented, but the APU is configured as unavailable.
Fixed interval timer	—	IVOR11	A fixed-interval timer exception exists (TSR[FIS] = 1), and the interrupt is enabled (TCR[FIE] = 1 and MSR[EE] = 1).

Table 11. Interrupts and Conditions—Overview (continued)

Interrupt Type	Vector Offset		Causing Conditions
	Classic	Book E	
Watchdog timer	—	IVOR12	Critical interrupt. Occurs when a watchdog timer exception exists (TSR[WIS] = 1), and the interrupt is enabled (TCR[WIE] = 1 and MSR[CE] = 1).
Data TLB error	—	IVOR13	A virtual addresses associated with an instruction fetch does not match any valid TLB entry.
Instruction TLB error	—	IVOR14	A virtual addresses associated with a fetch does not match any valid TLB entry.
Debug	—	IVOR15	Critical interrupt. A debug event causes a corresponding DBSR bit to be set and debug interrupts are enabled (DBCR0[IDM] = 1 and MSR[DE] = 1).
Reserved	—	IVOR16–IVOR31	Reserved for future architectural use.
EIS-Defined Interrupts			
SPE/APU unavailable	—	IVOR32	MSR[SPE] is cleared and an SPE or embedded floating-point instruction is executed.
Embedded floating-point data	—	IVOR33	Embedded floating-point invalid operation, underflow or overflow exception
Embedded floating-point round	—	IVOR34	Embedded floating-point inexact or rounding error

Memory Management Unit (MMU) Model

The MMU, together with the exception processing mechanism, provides the necessary support for the operating system to implement a paged virtual memory environment and for enforcing protection of designated memory areas. The MSR controls some of the critical functionality of the MMU.

The MMU and exception models support demand-paged virtual memory. Virtual memory management permits execution of programs larger than the size of physical memory; the term demand paged implies that individual pages are loaded into physical memory from backing storage only as they are accessed by an executing program.

The memory management model includes the concept of a virtual address that is not only larger than that of the maximum physical memory allowed but a virtual address space that is also larger than the effective address space. Effective addresses are 32 bits wide. In the address translation process, the processor converts an effective address to a virtual address of up to 52 bits in size.

Two general types of processor-generated memory accesses require address translation—instruction accesses and data accesses generated by load and store instructions. In addition, the addresses specified by cache instructions also require translation. Generally, the address translation mechanism is defined in terms of mapping an effective-to-physical address for memory accesses. The effective address is converted to an interim virtual address and a page table is used to translate the virtual address to a physical address.

Translation lookaside buffers (TLBs) are commonly implemented to keep recently-used page address translations on-chip. Although their exact characteristics are not specified in the architecture, the general concepts that are pertinent to the system software are described.

MMU Features in the Classic PowerPC Definition

In the classic PowerPC definition, the address translation mechanism is further defined in terms of segment descriptors. The segment information translates the effective address to an interim virtual address, and, as previously mentioned, the page table information translates the virtual address to a physical address.

Effective address spaces are divided into 256-Mbyte segments or into other large regions called blocks (128 Kbyte–256 Mbyte). Segments that correspond to memory-mapped areas can be further subdivided into 4-Kbyte pages.

The definition of the segment and page table data structures provides significant flexibility for the implementation of performance enhancement features in a wide range of processors. Therefore, the performance enhancements used to store the segment or page table information on-chip vary from implementation to implementation.

The segment information, used to generate the interim virtual addresses, is stored as segment descriptors. These descriptors may reside in on-chip segment registers (32-bit implementations) or as segment table entries (STEs) in memory (64-bit implementations).

The classic PowerPC architecture also defines the block address translation (BAT) mechanism, which is a software-controlled array that stores the available block address translations on-chip. BAT array entries are implemented as pairs of BAT registers that are accessible as supervisor special-purpose registers (SPRs).

For each block or page, the operating system creates an address descriptor (page table entry (PTE) or BAT array entry); the MMU then uses these descriptors to generate the physical address, the protection information, and other access control information each time an address within the block or page is accessed. Address descriptors for pages reside in tables (as PTEs) in physical memory; for faster accesses, the MMU often caches on-chip copies of recently-used PTEs in an on-chip TLB. The MMU keeps the block information on-chip in the BAT array (comprised of the BAT registers).

MMU Features in the Book E PowerPC Definition

Book E also provides alternatives to the MMU and interrupt models defined in the classic PowerPC definition. Where the classic architecture supports hardware-based page address translation with fixed 4-Kbyte pages, the Book E MMU is strictly software-managed and supports fixed- and variable-sized pages. Where the classic architecture defined block address translation (BAT) SPRs that could provide a single translation for large blocks of memory space, in Book E processors, this is done with variable-sized pages. For more information, see “Interrupt Model” on page 40.

Differences between Book E and the classic architecture MMU models are outlined in Table 12.

Table 12. Classic and Book E MMU Models

Classic PowerPC	Book E
Support for block address translation, page address translation, and real mode	Enhanced page address translation, no block address translation or real mode
Fixed 4-Kbyte pages	Support for both fixed and variable-sized page address translation mechanisms
Segmented memory model	No segments defined
Hardware page address translation definition with little architected support for software management	No hardware table hashing, and additional features that support management of page translation and protection in TLBs in software. Two instructions, TLB Read Entry (**tlbre**) and TLB Write Entry (**tlbwe**), are defined that provide direct software access to page translation and configuration.
Byte ordering. Modal big- and (munged) little-endian support provided through the MSR	Support for big- and true little-endian byte ordering provided on a per-page basis, programmed through the TLBs
DSI and ISI interrupts taken when an address cannot be translated or a protection violation occurs	In addition to the DSI and ISI interrupts, data and instruction TLB error interrupts are taken if there is a TLB miss.

Book E differs from the classic PowerPC architecture in that it defines some features more loosely. For example, the Book E MMU model defines the TLB Read Entry and TLB Write Entry instructions (**tlbre** and **tlbwe**) for reading and writing the TLBs in software. However, it does not specify how this is to be accomplished. Freescale processors execute these instructions by reading or writing the contents of a set of MMU assist (MAS) SPRs into the TLBs. These MAS registers, which provide the translation, protection, byte-ordering, and cache characteristics for the relevant pages, and the exact behavior of the **tlbre** and **tlbwe** instructions are defined by the Freescale EIS. The EIS provides additional optional features as shown in Table 13.

Table 13. Book E and Freescale EIS Layers of the MMU and Cache Models

Book E	EIS
TLB Read Entry (**tlbre**) and TLB Write Entry (**tlbwe**) give software direct access to page translation, protection, and configuration.	MMU assist registers (MAS*n*) defined as SPRs that hold translation, configuration, and protection information that is copied to and from the TLBs by executing **tlbwe** and **tlbre**.
A single process ID register (PID) used by system software to identify TLB entries used by the processor to accomplish address translation for loads, stores, and instruction fetches.	The EIS defines 14 additional PID registers, PID1–PID14. (the Book E–defined PID is treated as PID0). A implementation may choose to provide any number of PIDs up to a maximum of 15. The number of PIDs implemented is indicated by the value of MMUCFG[NPIDS].

Table 13. Book E and Freescale EIS Layers of the MMU and Cache Models (continued)

Book E	EIS
	Additional MMU registers: • MMU configuration register (MMUCFG). Identifies the number of bits in a real address supported by the implementation, the number and size of PID registers, and the number of TLBs. • TLB*n*CFG registers (one for each implemented TLB array). Indicates the number of ways of associativity, minimum and maximum page size, page size availability, number of entries, and invalidate protection capability in each TLB array. • MMUCFG0 used for general control of the MMU including flash invalidation of the TLB arrays and page sizes for programmable fixed size arrays
	The EIS defines the following cache-related APUs • L1 cache way partitioning APU (e200 cores), which allows partitioning of the unified cache implemented on e200 cores. • L1 cache locking APU (e500 cores) defines instructions and methods for locking cache lines for frequently used instructions and data. Cache locking allows software to instruct a cache to keep latency-sensitive data readily available for fast access.

Address Translation Mechanisms

The following types of address translation are supported:

- Page address translation—translates the page frame address for a 4-Kbyte page size. Book E provides enhanced page address translation that eliminates the need for block address translation and real addressing mode address translation.
- Block address translation—translates the block number for blocks that range in size from 128 Kbyte to 256 Mbyte (classic PowerPC only)
- Real addressing mode address translation—when address translation is disabled, the physical address is identical to the effective address. (classic PowerPC only)

Figure 14 shows the address translation mechanisms provided by the MMU.

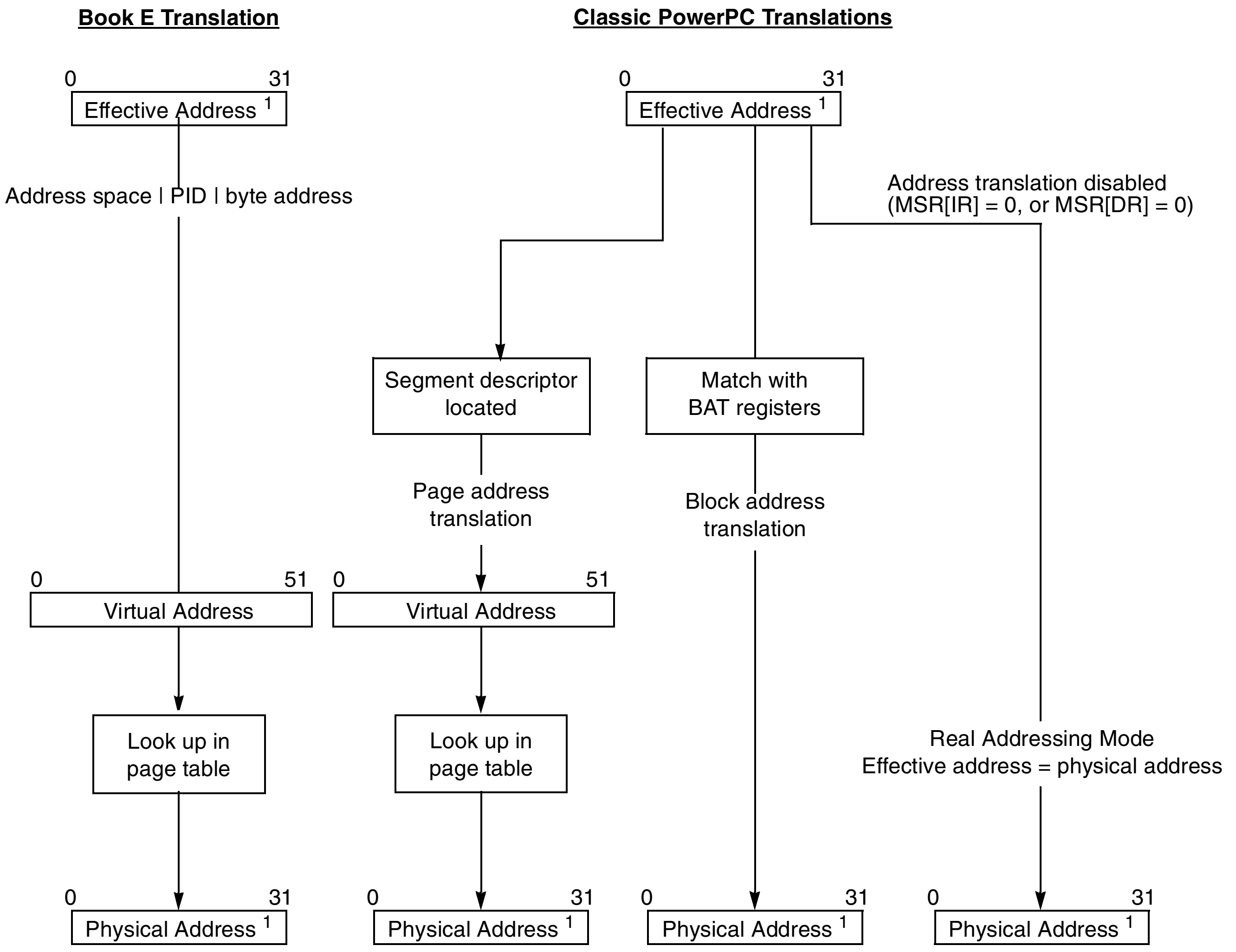
The segment descriptors shown in the figure control the page address translation mechanisms in classic PowerPC processors. When an access uses the page address translation, the appropriate segment descriptor is required. One of the 16 on-chip segment registers (which contain the segment descriptors) is selected by the highest-order effective address bits.

In classic PowerPC processors, for memory accesses translated by a segment descriptor, the interim virtual address is generated using the information in the segment descriptor. In Freescale Book E processors, the virtual address is determined by concatenating the address space bit and the process ID (PID) to the effective address.

In both classic and Book E processors, page address translation typically corresponds to the conversion of this virtual address into the 32-bit physical address (or in the case of the e600 family and the e500v2, to a 36-bit physical addresses). In some cases, the physical address for the page resides in an on-chip TLB and is available for quick access. However, if the page address translation misses in a TLB, the MMU searches the page table in memory (using the virtual address information and a hashing function) to locate the required physical address. Some classic PowerPC implementations may have dedicated hardware to perform the page table search automatically, while all Book E and some classic PowerPC implementations perform the page table searches with software.

Because blocks are larger than pages, there are fewer upper-order effective address bits to be translated into physical address bits (more low-order address bits (at least 17) are untranslated to form the offset into a block) for block address translation. Also, instead of segment descriptors and a page table, block address translations use the on-chip BAT registers as a BAT array. If an effective address matches the corresponding field of a BAT register, the information in the BAT register is used to generate the physical address; in this case, the results of the page translation (occurring in parallel) are ignored. Note that a matching BAT array entry takes precedence over a translation provided by the segment descriptor in all cases.

Also, Book E implementation used enhanced page address translation that eliminates the block address translation. Figure 14 shows a high-level address translation mechanism for classic and Book E PowerPC processors.


Book E Translation
Classic PowerPC Translations
0
31
Effective Address [1]
Address space | PID | byte address
Address translation disabled
(MSR[IR] = 0, or MSR[DR] = 0)
Segment descriptor located
Match with BAT registers
Page address translation
Block address translation
0
51
Virtual Address
Look up in page table
Real Addressing Mode
Effective address = physical address
Physical Address [1]

[1] The e600 family generates a 36-bit physical address; the e500v2 generates a 36-bit physical address as supported by the EIS definition of MAS7.

Figure 14. Address Translation Types

Chapter 3 Freescale Documentation

This chapter lists Freescale documentation that can be accessed at www.freescale.com/powerpc, Document order numbers are included in parentheses for ease in ordering:

- *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture* (MPCFPE32B/AD)—Describes resources defined by the PowerPC architecture.
- *EREF: A Reference for Freescale Book E and the e500 Core* (EREF)—This book provides a higher-level view of the programming model as it is defined by Book E, the Freescale Book E implementation standards, and the e500 microprocessor.
- Reference manuals—These books provide details about individual implementations and are intended for use with the *Programming Environments Manual*.
- Addenda/errata to reference manuals—Because some processors have follow-on parts, an addendum is provided that describes additional features and functionality changes. These addenda are intended for use with the corresponding reference manuals.
- Hardware specifications—Hardware specifications provide specific data regarding bus timing, signal behavior, and AC, DC, and thermal characteristics, as well as other design considerations. Separate hardware specifications are provided for each part.
- Product briefs—Each device has a product brief that provides an overview of its features. This document is roughly the equivalent to the overview (Chapter 1) of an implementation's reference manual.
- Application notes—These short documents address specific design issues useful to programmers and engineers working with Freescale processors.

Additional literature is published as new processors become available. For a current list of documentation, refer to www.freescale.com.