CHAPTER 3

Inside an Apple

Apple initiated its transition from the 68K hardware platform to the PowerPC in 1994. Within the next two years, Apple's entire line of computers moved to the PowerPC. The various PowerPC-based Apple computer families available at any given time have often differed in system architecture,[1] the specific processor used, and the processor vendor. For example, before the G4 iBook was introduced in October 2003, Apple's then current systems included three generations of the PowerPC: the G3, the G4, and the G5. Whereas the G4 processor line is supplied by Motorola, the G3 and the G5 are from IBM. Table 3–1 lists the various PowerPC processors[2] used by Apple.

On June 6, 2005, at the Worldwide Developers Conference in San Francisco, Apple announced its plans to base future models of Macintosh computers on Intel processors. The move was presented as a two-year transition: Apple stated that

1. System architecture refers to the type and interconnection of a system's hardware components, including—but not limited to—the processor type.

2. The list does not account for minor differences between processor models—for example, differences based solely on processor clock frequencies.

TABLE 3–1 Processors Used in PowerPC-Based Apple Systems

Processor	Introduced	Discontinued
PowerPC 601	March 1994	June 1996
PowerPC 603	April 1995	May 1996
PowerPC 603e	April 1996	August 1998
PowerPC 604	August 1995	April 1998
PowerPC 604e	August 1996	September 1998
PowerPC G3	November 1997	October 2003
PowerPC G4	October 1999	—
PowerPC G5	June 2003	—
PowerPC G5 (dual-core)	October 2005	—

although x86-based Macintosh models would become available by mid-2006, all Apple computers would transition to the x86 platform only by the end of 2007. The transition was faster than expected, with the first x86 Macintosh computers appearing in January 2006. These systems—the iMac and the MacBook Pro—were based on the Intel Core Duo[3] dual-core processor line, which is built on 65 nm process technology.

In this chapter, we will look at the system architecture of a specific type of Apple computer: a G5-based dual-processor Power Mac. Moreover, we will discuss a specific PowerPC processor used in these systems: the 970FX. We focus on a G5-based system because the 970FX is more advanced, more powerful, and more interesting in general than its predecessors. It is also the basis for the first 64-bit dual-core PowerPC processor: the 970MP.

3.1 The Power Mac G5

Apple announced the Power Mac G5—its first 64-bit desktop system—in June 2003. Initial G5-based Apple computers used IBM's PowerPC 970 processors. These were followed by systems based on the 970FX processor. In late 2005, Apple revamped the Power Mac line by moving to the dual-core 970MP proces-

3. This processor was originally codenamed *Yonah*.

sor. The 970, 970FX, and 970MP are all derived from the execution core of the POWER4 processor family, which was designed for IBM's high-end servers. G5 is Apple's marketing term for the 970 and its variants.

IBM's Other G5

There was another G5 from IBM—the microprocessor used in the S/390 G5 system, which was announced in May 1998. The S/390 G5 was a member of IBM's CMOS[4] mainframe family. Unlike the 970 family processors, the S/390 G5 had a Complex Instruction-Set Computer (CISC) architecture.

Before we examine the architecture of any particular Power Mac G5, note that various Power Mac G5 models may have slightly different system architectures. In the following discussion, we will refer to the system shown in Figure 3–1.

3.1.1 The U3H System Controller

The U3H system controller combines the functionality of a memory controller[5] and a PCI bus bridge.[6] It is a custom integrated chip (IC) that is the meeting point of key system components: processors, the Double Data Rate (DDR) memory system, the Accelerated Graphics Port (AGP)[7] slot, and the HyperTransport bus that runs into a PCI-X bridge. The U3H provides bridging functionality by performing point-to-point routing between these components. It supports a Graphics Address Remapping Table (GART) that allows the AGP bridge to translate linear addresses used in AGP transactions into physical addresses. This improves the performance of direct memory access (DMA) transactions involving multiple pages that would typically be noncontiguous in virtual memory. Another table supported by the U3H is the Device Address Resolution Table (DART),[8] which translates linear addresses to physical addresses for devices attached to the

4. CMOS stands for Complementary Metal Oxide Semiconductor—a type of integrated circuit technology. CMOS chips use metal oxide semiconductor field effect transistors (MOSFETs), which differ greatly from the bipolar transistors that were prevalent before CMOS. Most modern processors are manufactured in CMOS technology.

5. A memory controller controls processor and I/O interactions with the memory system.

6. The G5 processors use the PCI bus bridge to execute operations on the PCI bus. The bridge also provides an interface through which PCI devices can access system memory.

7. AGP extends the PCI standard by adding functionality optimized for video devices.

8. DART is sometimes expanded as DMA Address Relocation Table.

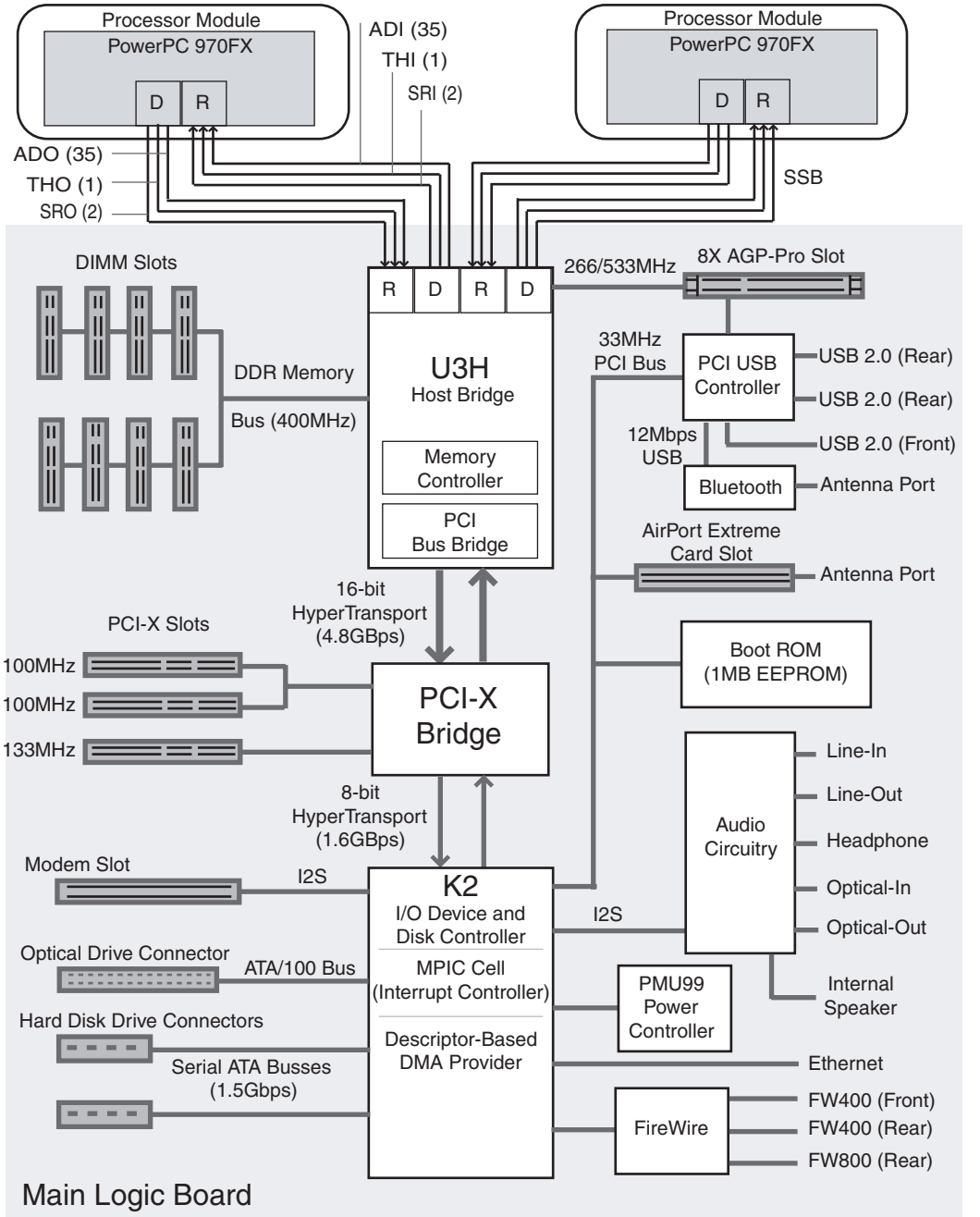

FIGURE 3–1 Architecture of a dual-processor Power Mac G5 system

HyperTransport bus. We will come across the DART in Chapter 10, when we discuss the I/O Kit.

3.1.2 The K2 I/O Device Controller

The U3H is connected to a PCI-X bridge via a 16-bit HyperTransport bus. The PCI-X bridge is further connected to the K2 custom IC via an 8-bit HyperTransport bus. The K2 is a custom integrated I/O device controller. In particular, it provides disk and multiprocessor interrupt controller (MPIC) functionality.

3.1.3 PCI-X and PCI Express

The Power Mac system shown in Figure 3–1 provides three PCI-X 1.0 slots. Power Mac G5 systems with dual-core processors use PCI Express.

3.1.3.1 PCI-X

PCI-X was developed to increase the bus speed and reduce the latency of PCI (see the sidebar "A Primer on Local Busses"). PCI-X 1.0 was based on the existing PCI architecture. In particular, it is also a shared bus. It solves many—but not all—of the problems with PCI. For example, its split-transaction protocol improves bus bandwidth utilization, resulting in far greater throughput rates than PCI. It is fully backward compatible in that PCI-X cards can be used in Conventional PCI slots, and conversely, Conventional PCI cards—both 33MHz and 66MHz—can be used in PCI-X slots. However, PCI-X is not electrically compatible with 5V-only cards or 5V-only slots.

PCI-X 1.0 uses 64-bit slots. It provides two speed grades: PCI-X 66 (66MHz signaling speed, up to 533MBps peak throughput) and PCI-X 133 (133MHz signaling speed, up to 1GBps peak throughput).

PCI-X 2.0 provides enhancements such as the following:

- An error correction code (ECC) mechanism for providing automatic 1-bit error recovery and 2-bit error detection
- New speed grades: PCI-X 266 (266MHz signaling speed, up to 2.13GBps peak throughput) and PCI-X 533 (533MHz signaling speed, up to 4.26GBps peak throughput)
- A new 16-bit interface for embedded or portable applications

Note how the slots are connected to the PCI-X bridge in Figure 3–1: Whereas one of them is "individually" connected (a *point-to-point* load), the other two "share" a connection (a *multidrop* load). A PCI-X speed limitation is that *its highest speed grades are supported only if the load is point-to-point*. Specifically, two PCI-X 133 loads will each operate at a maximum of 100MHz.[9] Correspondingly, two of this Power Mac's slots are 100MHz each, whereas the third is a 133MHz slot.

> The next revision of PCI-X—3.0—provides a 1066MHz data rate with a peak throughput of 8.5GBps.

3.1.3.2 PCI Express

An alternative to using a shared bus is to use point-to-point links to connect devices. PCI Express[10] uses a high-speed, point-to-point architecture. It provides PCI compatibility using established PCI driver programming models. Software-generated I/O requests are transported to I/O devices through a split-transaction, packet-based protocol. In other words, PCI Express essentially serializes and packetizes PCI. It supports multiple interconnect widths—a link's bandwidth can be linearly scaled by adding signal pairs to form *lanes*. There can be up to 32 separate lanes.

A Primer on Local Busses

As CPU speeds have increased greatly over the years, other computer subsystems have not managed to keep pace. Perhaps an exception is the main memory, which has fared better than I/O bandwidth. In 1991,[11] Intel introduced the *Peripheral Component Interconnect* (PCI) local bus standard. In the simplest terms, a bus is a shared communications link. In a computer system, a bus is implemented as a set of wires that connect some of the computer's subsystems. Multiple busses are typically used as building blocks to construct complex com-

9. Four PCI-X 133 loads in a multidrop configuration will operate at a maximum speed of 66MHz each.

10. The PCI Express standard was approved by the PCI-SIG Board of Directors in July 2002. PCI Express was formerly called 3GIO.

11. This was also the year that Macintosh System 7 was released, the Apple-IBM-Motorola (AIM) alliance was formed, and the Personal Computer Memory Card International Association (PCMCIA) was established, among other things.

puter systems. The "local" in local bus implies its proximity to the processor.[12] The PCI bus has proven to be an extremely popular interconnect mechanism (also called simply an interconnect), particularly in the so-called North Bridge/South Bridge implementation. A *North Bridge* typically takes care of communication between the processor, main memory, AGP, and the South Bridge. Note, however, that modern system designs are moving the memory controller to the processor die, thus making AGP obsolete and rendering the traditional North Bridge unnecessary.

A typical *South Bridge* controls various busses and devices, including the PCI bus. It is common to have the PCI bus work both as a plug-in bus for peripherals and as an interconnect allowing devices connected directly or indirectly to it to communicate with memory.

The PCI bus uses a shared, parallel multidrop architecture in which address, data, and control signals are multiplexed on the bus. When one PCI bus master[13] uses the bus, other connected devices must either wait for it to become free or use a contention protocol to request control of the bus. Several *sideband* signals[14] are required to keep track of communication directions, types of bus transactions, indications of bus-mastering requests, and so on. Moreover, a shared bus runs at limited clock speeds, and since the PCI bus can support a wide variety of devices with greatly varying requirements (in terms of bandwidth, transfer sizes, latency ranges, and so on), bus arbitration can be rather complicated. PCI has several other limitations that are beyond the scope of this chapter.

PCI has evolved into multiple variants that differ in backward compatibility, forward planning, bandwidth supported, and so on.

- **Conventional PCI**—The original PCI Local Bus Specification has evolved into what is now called *Conventional PCI*. The PCI Special Interest Group (PCI-SIG) introduced PCI 2.01 in 1993, followed by revisions 2.1 (1995), 2.2 (1998), and 2.3 (2002). Depending on the revision, PCI bus characteristics include the following: 5V or 3.3V signaling, 32-bit or 64-bit bus width, operation at 33MHz or 66MHz, and a peak throughput of 133MBps, 266MBps, or 533MBps. Conventional PCI 3.0—the current standard—finishes the migration of the PCI bus from being a 5.0V signaling bus to a 3.3V signaling bus.
- **MiniPCI**—MiniPCI defines a smaller form factor PCI card based on PCI 2.2. It is meant for use in products where space is a premium—such as notebook computers, docking stations, and set-top boxes. Apple's AirPort Extreme wireless card is based on MiniPCI.

12. The first local bus was the VESA local bus (VLB).

13. A bus master is a device that can initiate a read or write transaction—for example, a processor.

14. In the context of PCI, a sideband signal is any signal that is not part of the PCI specification but is used to connect two or more PCI-compliant devices. Sideband signals can be used for product-specific extensions to the bus, provided they do not interfere with the specification's implementation.

- **CardBus**—CardBus is a member of the PC Card family that provides a 32-bit, 33MHz PCI-like interface that operates at 3.3V. The PC Card Standard is maintained by the PCMCIA.[15]

PCI-X (Section 3.1.3.1) and PCI Express (Section 3.1.3.2) represent further advancements in I/O bus architecture.

3.1.4 HyperTransport

HyperTransport (HT) is a high-speed, point-to-point, chip interconnect technology. Formerly known as Lightning Data Transport (LDT), it was developed in the late 1990s at Advanced Micro Devices (AMD) in collaboration with industry partners. The technology was formally introduced in July 2001. Apple Computer was one of the founding members of the HyperTransport Technology Consortium. The HyperTransport architecture is open and nonproprietary.

HyperTransport aims to simplify complex chip-to-chip and board-to-board interconnections in a system by replacing multilevel busses. Each connection in the HyperTransport protocol is between two devices. Instead of using a single bidirectional bus, each connection consists of *two unidirectional links*. HyperTransport point-to-point interconnects (Figure 3–2 shows an example) can be extended to support a variety of devices, including tunnels, bridges, and end-point devices. HyperTransport connections are especially well suited for devices on the main logic board—that is, those devices that require the lowest latency and the highest performance. Chains of HyperTransport links can also be used as I/O channels, connecting I/O devices and bridges to a host system.

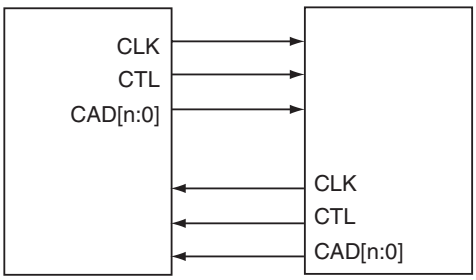

FIGURE 3–2 HyperTransport I/O link

15. The PCMCIA was established to standardize certain types of add-in memory cards for mobile computers.

Some important HyperTransport features include the following.

- HyperTransport uses a packet-based data protocol in which narrow and fast unidirectional point-to-point links carry *command*, *address*, and *data* (CAD) information encoded as packets.
- The electrical characteristics of the links help in cleaner signal transmission, higher clock rates, and lower power consumption. Consequently, considerably fewer sideband signals are required.
- Widths of various links do not need to be equal. An 8-bit-wide link can easily connect to a 32-bit-wide link. Links can scale from 2 bits to 4, 8, 16, or 32 bits in width. As shown in Figure 3–1, the HyperTransport bus between the U3H and the PCI-X bridge is 16 bits wide, whereas the PCI-X bridge and the K2 are connected by an 8-bit-wide HyperTransport bus.
- Clock speeds of various links do not need to be equal and can scale across a wide spectrum. Thus, it is possible to scale links in both *width* and *speed* to suit specific needs.
- HyperTransport supports *split transactions*, eliminating the need for inefficient retries, disconnects by targets, and insertion of wait states.
- HyperTransport combines many benefits of serial and parallel bus architectures.
- HyperTransport has comprehensive legacy support for PCI.

Split Transactions

When split transactions are used, a *request* (which requires a response) and *completion* of that request—the *response*[16]—are separate transactions on the bus. From the standpoint of operations that are performed as split transactions, the link is free after the request is sent and before the response is received. Moreover, depending on a chipset's implementation, multiple transactions could be pending[17] at the same time. It is also easier to route such transactions across larger fabrics.

HyperTransport was designed to work with the widely used PCI bus standard—it is software compatible with PCI, PCI-X, and PCI Express. In fact, it could be viewed as a superset of PCI, since it can offer complete PCI transparency by preserving

16. The response may also have data associated with it, as in the case of a read operation.

17. This is analogous to tagged queuing in the SCSI protocol.

PCI definitions and register formats. It can conform to PCI ordering and configuration specifications. It can also use Plug-and-Play so that compliant operating systems can recognize and configure HyperTransport-enabled devices. It is designed to support both CPU-to-CPU communications and CPU-to-I/O transfers, while emphasizing low latency.

A HyperTransport tunnel device can be used to provide connection to other busses such as PCI-X. A system can use additional HyperTransport busses by using an HT-to-HT bridge.

Apple uses HyperTransport in G5-based systems to connect PCI, PCI-X, USB, FireWire, Audio, and Video links. The U3H acts as a North Bridge in this scenario.

System Architecture and Platform

From the standpoint of Mac OS X, we can define a system's *architecture* to be primarily a combination of its processor type, the North Bridge (including the memory controller), and the I/O controller. For example, the AppleMacRISC4PE system architecture consists of one or more G5-based processors, a U3-based North Bridge, and a K2-based I/O controller. The combination of a G3- or G4-based processor, a UniNorth-based host bridge, and a KeyLargo-based I/O controller is referred to as the AppleMacRISC2PE system architecture.

A more model-specific concept is that of a *platform*, which usually depends on the specific motherboard and is likely to change more frequently than system architecture. An example of a platform is PowerMac11,2, which corresponds to a 2.5GHz quad-processor (dual dual-core) Power Mac G5.

3.1.5 Elastic I/O Interconnect

The PowerPC 970 was introduced along with Elastic I/O, a high-bandwidth and high-frequency processor-interconnect (PI) mechanism that requires no bus-level arbitration.[18] Elastic I/O consists of two 32-bit logical busses, each a high-speed source-synchronous bus (SSB) that represents a unidirectional point-to-point connection. As shown in Figure 3–1, one travels from the processor to the U3H companion chip, and the other travels from the U3H to the processor. In a dual-processor system, each processor gets its own dual-SSB bus. Note that the SSBs also support cache-coherency "snooping" protocols for use in multiprocessor systems.

18. In colloquial terms, arbitration is the mechanism that answers the question, "Who gets the bus?"

A *synchronous* bus is one that includes a clock signal in its control lines. Its communication protocol functions with respect to the clock. A *source-synchronous* bus uses a timing scheme in which a clock signal is forwarded along with the data, allowing data to be sampled precisely when the clock signal goes high or low.

Whereas the logical width of each SSB is 32 bits, the physical width is greater. Each SSB consists of 50 signal lines that are used as follows:

- 2 signals for the differential bus clock lines
- 44 signals for data, to transmit 35 bits of address and data or control information (AD), along with 1 bit for transfer-handshake (TH) packets for acknowledging such command or data packets received on the bus
- 4 signals for the differential snoop response (SR) bus to carry snoop-coherency responses, allowing global snooping activities to maintain cache coherency

Using 44 physical bits to transmit 36 logical bits of information allows 8 bits to be used for parity. Another supported format for redundant data transmission uses a balanced coding method (BCM) in which there are exactly 22 high signals and 22 low signals if the bus state is valid.

The overall processor interconnect is shown in Figure 3–1 as logically consisting of three inbound segments (ADI, THI, SRI) and three outbound segments (ADO, THO, SRO). The direction of transmission is from a driver side (D), or master, to a receive side (R), or slave. The unit of data transmission is a packet.

Each SSB runs at a frequency that is an integer fraction of the processor frequency. The 970FX design allows several such ratios. For example, Apple's dual-processor 2.7GHz system has an SSB frequency of 1.35GHz (a PI bus ratio of 2:1), whereas one of the single-processor 1.8GHz models has an SSB frequency of 600MHz (a PI bus ratio of 3:1).

The bidirectional nature of the channel between a 970FX processor and the U3H means there are dedicated data paths for reading and writing. Consequently, throughput will be highest in a workload containing an equal number of reads and writes. Conventional bus architectures that are shared and unidirectional-at-a-time will offer higher peak throughput for workloads that are mostly reads or mostly writes. In other words, Elastic I/O leads to higher bus utilization for balanced workloads.

The Bus Interface Unit (BIU) is capable of self-tuning during startup to ensure optimal signal quality.

3.2 The G5: Lineage and Roadmap

As we saw earlier, the G5 is a derivative of IBM's POWER4 processor. In this section, we will briefly look at how the G5 is similar to and different from the POWER4 and some of the POWER4's successors. This will help us understand the position of the G5 in the POWER/PowerPC roadmap. Table 3–2 provides a high-level summary of some key features of the POWER4 and POWER5 lines.

TABLE 3–2 POWER4 and Newer Processors

	POWER4	POWER4+	POWER5	POWER5+
Year introduced	2001	2002	2004	2005
Lithography	180 nm	130 nm	130 nm	90 nm
Cores/chip	2	2	2	2
Transistors	174 million	184 million	276 million/chip[a]	276 million/chip
Die size	415 mm^2	267 mm^2	389 mm^2/chip	243 mm^2/chip
LPAR[b]	Yes	Yes	Yes	Yes
SMT[c]	No	No	Yes	Yes
Memory controller	Off-chip	Off-chip	On-chip	On-chip
Fast Path	No	No	Yes	Yes
L1 I-cache	2×64KB	2×64KB	2×64KB	2×64KB
L1 D-cache	2×32KB	2×32KB	2×32KB	2×32KB
L2 cache	1.41MB	1.5MB	1.875MB	1.875MB
L3 cache	32MB+	32MB+	36MB+	36MB+

a. A chip includes two processor cores and L2 cache. A multichip module (MCM) contains multiple chips and usually L3 cache. A four-chip POWER5 MCM with four L3 cache modules is 95 mm^2.

b. LPAR stands for (processor-level) Logical Partitioning.

c. SMT stands for simultaneous multithreading.

Transcribing Transistors

In light of the technical specifications of modern processors, it is interesting to see how they compare with some of the most important processors in the history of personal computing.

- **Intel 4004**—1971, 750kHz clock frequency, 2,300 transistors, 4-bit accumulator architecture, 8 μm pMOS, 3×4 mm^2, 8–16 cycles/instruction, designed for a desktop printing calculator
- **Intel 8086**—1978, 8MHz clock frequency, 29,000 transistors, 16-bit extended accumulator architecture, assembly-compatible with the 8080, 20-bit addressing through a segmented addressing scheme
- **Intel 8088**—1979 (prototyped), 8-bit bus version of the 8086, used in the IBM PC in 1981
- **Motorola 68000**—1979, 8MHz clock frequency, 68,000 transistors, 32-bit general-purpose register architecture (with 24 address pins), heavily microcoded (even nanocoded), eight address registers, eight data registers, used in the original Macintosh in 1984

3.2.1 Fundamental Aspects of the G5

All POWER processors listed in Table 3–2, as well as the G5 derivatives, share some fundamental architectural features. They are all *64-bit* and *superscalar*, and they perform *speculative*, *out-of-order* execution. Let us briefly discuss each of these terms.

3.2.1.1 64-bit Processor

Although there is no formal definition of what constitutes a *64-bit* processor, the following attributes are shared by all 64-bit processors:

- 64-bit-wide general-purpose registers
- Support for 64-bit virtual addressing, although the physical or virtual address spaces may not use all 64 bits
- Integer arithmetic and logical operations performed on all 64 bits of a 64-bit operand—without being broken down into, say, two operations on two 32-bit quantities

The PowerPC architecture was designed to support both 32-bit and 64-bit computation modes—an implementation is free to implement only the 32-bit subset.

The G5 supports both computation modes. In fact, the POWER4 supports multiple processor architectures: the 32-bit and 64-bit POWER; the 32-bit and 64-bit PowerPC; and the 64-bit Amazon architecture. We will use the term PowerPC to refer to both the *processor* and the *processor architecture*. We will discuss the 64-bit capabilities of the 970FX in Section 3.3.12.1.

Amazon

The Amazon architecture was defined in 1991 by a group of IBM researchers and developers as they collaborated to create an architecture that could be used for both the RS/6000 and the AS/400. Amazon is a 64-bit-only architecture.

3.2.1.2 Superscalar

If we define *scalar* to be a processor design in which one instruction is issued per clock cycle, then a *superscalar* processor would be one that issues a variable number of instructions per clock cycle, allowing a clock-cycle-per-instruction (CPI) ratio of less than 1. It is important to note that even though a superscalar processor can issue multiple instructions in a clock cycle, it can do so only with several caveats, such as whether the instructions depend on each other and which specific functional units they use. Superscalar processors typically have multiple functional units, including multiple units of the same type.

VLIW

Another type of multiple-issue processor is a *very-large instruction-word* (VLIW) processor, which packages multiple operations into one very long instruction. The compiler—rather than the processor's instruction dispatcher—plays a critical role in selecting which instructions are to be issued simultaneously in a VLIW processor. It may schedule operations by using heuristics, traces, and profiles to guess branch directions.

3.2.1.3 Speculative Execution

A *speculative* processor can execute instructions before it is determined whether those instructions will need to be executed (instructions may not need to be executed because of a branch that bypasses them, for example). Therefore, instruction execution does not wait for *control* dependencies to resolve—it waits only

for the instruction's operands (*data*) to become available. Such speculation can be done by the compiler, the processor, or both. The processors in Table 3–2 employ in-hardware dynamic branch prediction (with multiple branches "in flight"), speculation, and dynamic scheduling of instruction groups to achieve substantial instruction-level parallelism.

3.2.1.4 Out-of-Order Execution

A processor that performs *out-of-order execution* includes additional hardware that can bypass instructions whose operands are not available—say, due to a cache miss that occurred during register loading. Thus, rather than always executing instructions in the order they appear in the programs being run, the processor may execute instructions whose operands are ready, deferring the bypassed instructions for execution at a more appropriate time.

3.2.2 New POWER Generations

The POWER4 contains two processor cores in a single chip. Moreover, the POWER4 architecture has features that help in virtualization. Examples include a special hypervisor mode in the processor, the ability to include an address offset when using nonvirtual memory addressing, and support for multiple global interrupt queues in the interrupt controller. IBM's Logical Partitioning (LPAR) allows multiple independent operating system images (such as AIX and Linux) to be run on a single POWER4-based system simultaneously. Dynamic LPAR (DLPAR), introduced in AIX 5L Version 5.2, allows dynamic addition and removal of resources from active partitions.

The POWER4+ improves upon the POWER4 by reducing its size, consuming less power, providing a larger L2 cache, and allowing more DLPAR partitions.

The POWER5 introduces simultaneous multithreading (SMT), wherein a single processor supports multiple instruction streams—in this case, two—simultaneously.

Many Processors . . . Simultaneously

IBM's RS 64 IV, a 64-bit member of the PowerPC family, was the first mainstream processor to support processor-level multithreading (the *processor* holds the states of multiple threads). The RS 64 IV implemented coarse-grained two-way multithreading—a single thread (the foreground thread) executed until a high-latency event, such as a cache miss, occurred. Thereafter, execution switched to

the background thread. This was essentially a very fast hardware-based context-switching implementation. Additional hardware resources allowed two threads to have their state in hardware at the same time. Switching between the two states was extremely fast, consuming only three "dead" cycles.

The POWER5 implements two-way SMT, which is far more fine-grained. The processor fetches instructions from two active instruction streams. Each instruction includes a thread indicator. The processor can issue instructions from both streams simultaneously to the various functional units. In fact, an instruction pipeline can simultaneously contain instructions from both streams in its various stages.

A two-way SMT implementation does not provide a factor-of-two performance improvement—the processor effectively behaves as if it were more than one processor, but not quite two processors. Nevertheless, the operating system sees a symmetric-multiprocessing (SMP) programming mode. Typical improvement factors range between 1.2 and 1.3, with a best case of about 1.6. In some pathological cases, the performance could even degrade.

A single POWER5 chip contains two cores, each of which is capable of two-way SMT. A multichip module (MCM) can contain multiple such chips. For example, a four-chip POWER5 module has eight cores. When each core is running in SMT mode, the operating system will see *sixteen* processors. Note that the operating system will be able to utilize the "real" processors first before resorting to SMT.

The POWER5 supports other important features such as the following:

- 64-way multiprocessing.
- Subprocessor partitioning (or *micropartitioning*), wherein multiple LPAR partitions can share a single processor.[19] Micropartitioned LPARs support automatic CPU load balancing.
- *Virtual Inter-partition Ethernet*, which enables a VLAN connection between LPARs—at gigabit or even higher speeds—without requiring physical network interface cards. Virtual Ethernet devices can be defined through the management console. Multiple virtual adapters are supported per partition, depending on the operating system.
- *Virtual I/O Server Partition*,[20] which provides virtual disk storage and Ethernet adapter sharing. Ethernet sharing connects virtual Ethernet to external networks.

19. A single processor may be shared by up to 10 partitions, with support for up to 160 partitions total.

20. The Virtual I/O Server Partition must run in either a dedicated partition or a micropartition.

- An on-chip memory controller.
- Dynamic firmware updates.
- Detection and correction of errors in transmitting data courtesy of specialized circuitry.
- *Fast Path*, the ability to execute some common software operations directly within the processor. For example, certain parts of TCP/IP processing that are traditionally handled within the operating system using a sequence of processor instructions could be performed via a single instruction. Such silicon acceleration could be applied to other operating system areas such as message passing and virtual memory.

Besides using 90-nm technology, the POWER5+ adds several features to the POWER5's feature set, for example: 16GB page sizes, 1TB segments, multiple page sizes per segment, a larger (2048-entry) translation lookaside buffer (TLB), and a larger number of memory controller read queues.

The POWER6 is expected to add evolutionary improvements and to extend the Fast Path concept even further, allowing functions of higher-level software—for example, databases and application servers—to be performed in silicon.[21] It is likely to be based on a 65-nm process and is expected to have multiple ultra-high-frequency cores and multiple L2 caches.

3.2.3 The PowerPC 970, 970FX, and 970MP

The PowerPC 970 was introduced in October 2002 as a 64-bit high-performance processor for desktops, entry-level servers, and embedded systems. The 970 can be thought of as a stripped-down POWER4+. Apple used the 970—followed by the 970FX and the 970MP—in its G5-based systems. Table 3–3 contains a brief comparison of the specifications of these processors. Figure 3–3 shows a pictorial comparison. Note that unlike the POWER4+, whose L2 cache is shared between cores, each core in the 970MP has its own L2 cache, which is twice as large as the L2 cache in the 970 or the 970FX.

Another noteworthy point about the 970MP is that both its cores share the same input and output busses. In particular, the output bus is shared "fairly" between cores using a simple round-robin algorithm.

21. The "reduced" in RISC becomes not quite reduced!

TABLE 3–3 POWER4+ and the PowerPC 9xx

	POWER4+	PowerPC 970	PowerPC 970FX	PowerPC 970MP
Year introduced	2002	2002	2004	2005
Lithography	130 nm	130 nm	90 nm[a]	90 nm
Cores/chip	2	1	1	2
Transistors	184 million	55 million	58 million	183 million
Die size	267 mm^2	121 mm^2	66 mm^2	154 mm^2
LPAR	Yes	No	No	No
SMT	No	No	No	No
Memory controller	Off-chip	Off-chip	Off-chip	Off-chip
Fast Path	No	No	No	No
L1 I-cache	2×64KB	64KB	64KB	2×64KB
L1 D-cache	2×32KB	32KB	32KB	2×32KB
L2 cache	1.41MB shared[b]	512KB	512KB	2×1MB
L3 cache	32MB+	None	None	None
VMX (AltiVec[c])	No	Yes	Yes	Yes
PowerTune[d]	No	No	Yes	Yes

a. The 970FX and 970MP use 90 nm lithography, in which copper wiring, strained silicon, and silicon-on-insulator (SOI) are fused into the same manufacturing process. This technique accelerates electron flow through transistors and provides an insulating layer in silicon. The result is increased performance, transistor isolation, and lower power consumption. Controlling power dissipation is particularly critical for chips with low process geometries, where subthreshold leakage current can cause problems.

b. The L2 cache is shared between the two processor cores.

c. Although jointly developed by Motorola, Apple, and IBM, AltiVec is a trademark of Motorola, or more precisely, Freescale. In early 2004, Motorola spun out its semiconductor products sector as Freescale Semiconductor, Inc.

d. PowerTune is a clock-frequency and voltage-scaling technology.

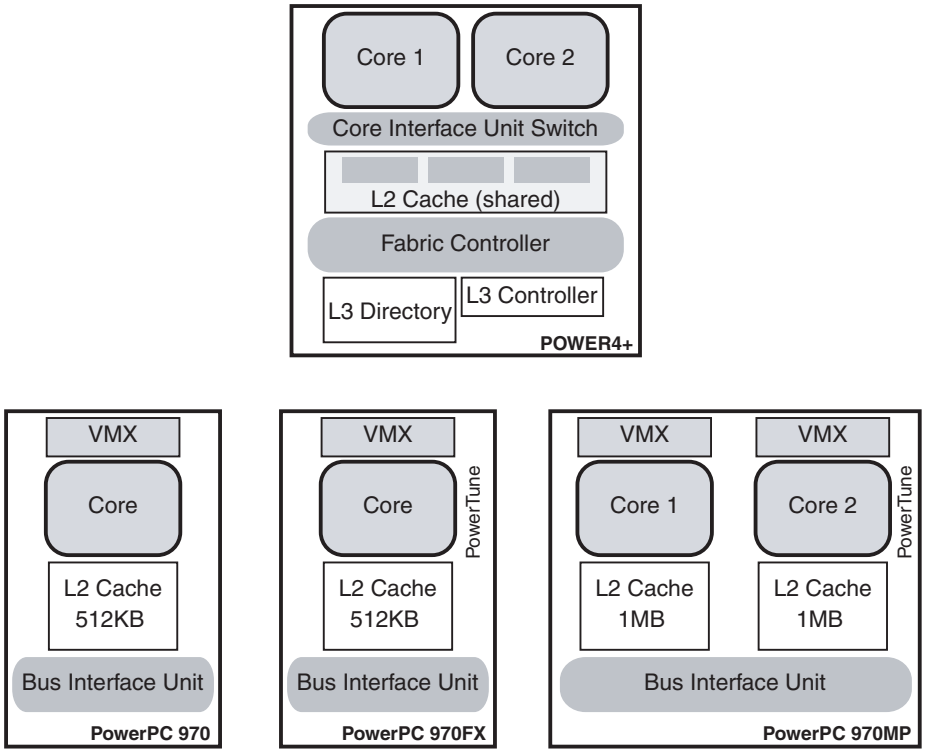

FIGURE 3–3 The PowerPC 9xx family and the POWER4+

3.2.4 The Intel Core Duo

In contrast, the Intel Core Duo processor line used in the first x86-based Macintosh computers (the iMac and the MacBook Pro) has the following key characteristics:

- Two cores per chip
- Manufactured using 65-nm process technology
- 90.3 mm^2 die size
- 151.6 million transistors
- Up to 2.16GHz frequency (along with a 667MHz processor system bus)
- 32KB on-die I-cache and 32KB on-die D-cache (write-back)
- 2MB on-die L2 cache (shared between the two cores)
- Data prefetch logic

- Streaming SIMD[22] Extensions 2 (SSE2) and Streaming SIMD Extensions 3 (SSE3)
- Sophisticated power and thermal management features

3.3 The PowerPC 970FX

3.3.1 At a Glance

In this section, we will look at details of the PowerPC 970FX. Although several parts of the discussion could apply to other PowerPC processors, we will not attempt to identify such cases. Table 3–4 lists the important technical specifications of the 970FX.

TABLE 3–4 The PowerPC 970FX at a Glance

Feature	Details
Architecture	64-bit PowerPC AS,[a] with support for 32-bit operating system bridge facility
Extensions	Vector/SIMD Multimedia extension (VMX[b])
Processor clock frequency	Up to 2.7GHz[c]
Front-side bus frequency	Integer fraction of processor clock frequency
Data-bus width	128 bits
Address-bus width	42 bits
Maximum addressable physical memory	4TB (2^{42} bytes)
Address translation	65-bit virtual addresses, 42-bit real addresses, support for large (16MB) virtual memory pages, a 1024-entry translation lookaside buffer (TLB), and a 64-entry segment lookaside buffer (SLB)
Endianness	Big-endian; optional little-endian facility *not* implemented
L1 I-cache	64KB, direct-mapped, with parity
L1 D-cache	32KB, two-way set-associative, with parity

(continues)

22. Section 3.3.10.1 defines SIMD.

TABLE 3–4 The PowerPC 970FX at a Glance *(Continued)*

Feature	Details
L2 cache	512KB, eight-way set-associative, with ECC, fully inclusive of L1 D-cache
L3 cache	None
Cache line width	128 bytes for all caches
Instruction buffer	32 entries
Instructions/cycle	Up to five (up to four nonbranch + up to one branch)
General-purpose registers	32×64-bit
Floating-point registers	32×64-bit
Vector registers	32×128-bit
Load/Store Units	Two units, with 64-bit data paths
Fixed-Point Units	Two asymmetrical[d] 64-bit units
Floating-Point Units	Two 64-bit units, with support for IEEE-754 double-precision floating-point, hardware fused multiply-add, and square root
Vector units	A 128-bit unit
Condition Register Unit	For performing logical operations on the Condition Register (CR)
Execution pipeline	Ten execution pipelines, with up to 25 stages in a pipeline, and up to 215 instructions in various stages of execution at a time
Power management	Multiple software-initialized power-saving modes, PowerTune frequency and voltage scaling

a. AS stands for Advanced Series.

b. VMX is interchangeable with AltiVec. Apple markets the PowerPC's vector functionality as Velocity Engine.

c. As of 2005.

d. The two fixed-point (integer) units of the 970FX are not symmetrical. Only one of them can perform division, and only one can be used for special-purpose register (SPR) operations.

3.3.2 Caches

A multilevel cache hierarchy is a common aspect of modern processors. A cache can be defined as a small chunk of very fast memory that stores recently used data, instructions, or both. Information is typically added and removed from a cache in aligned quanta called *cache lines*. The 970FX contains several caches and other special-purpose buffers to improve memory performance. Figure 3–4 shows a conceptual diagram of these caches and buffers.

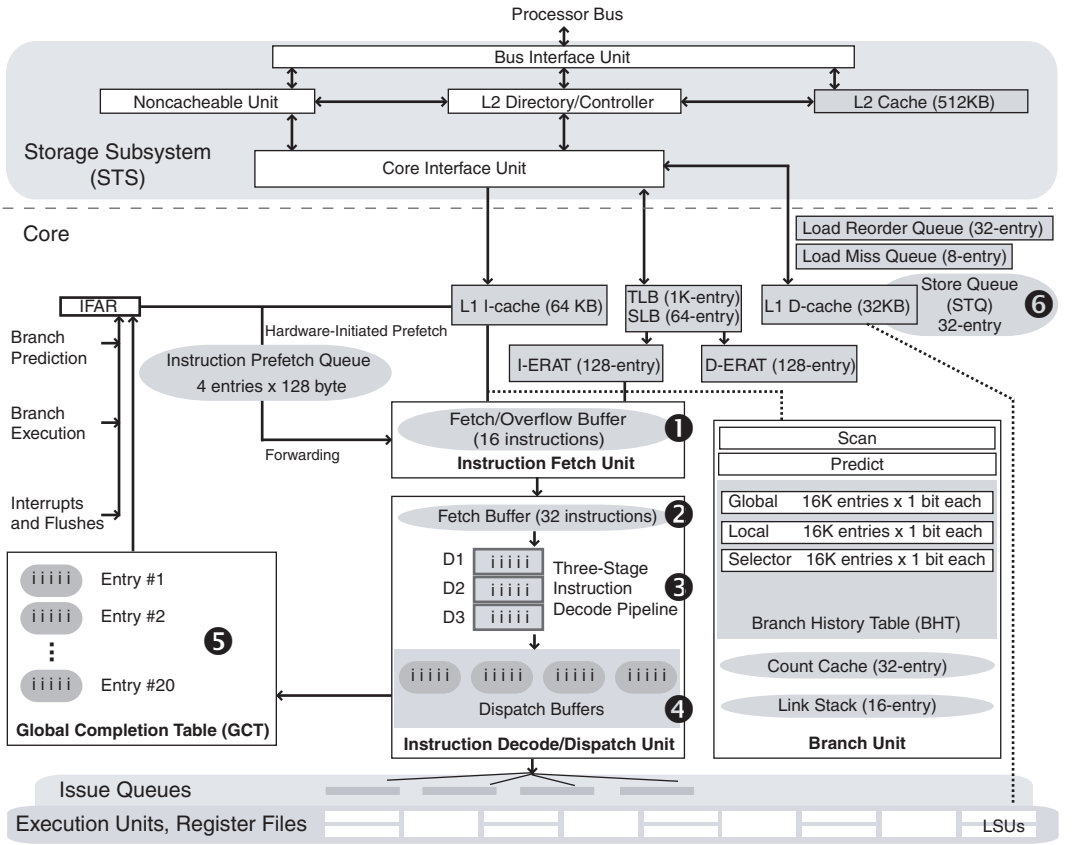

FIGURE 3–4 Caches and buffers in the 970FX

3.3.2.1 L1 and L2 Caches

The level 1 (L1) cache is closest to the processor. Memory-resident information must be loaded into this cache before the processor can use it, unless that portion of memory is marked noncacheable. For example, when a load instruction is being executed, the processor refers to the L1 cache to see if the data in question is already held by a currently resident cache line. If so, the data is simply loaded from the L1 cache—an L1 cache *hit*. This operation takes only a few processor cycles as compared to a few hundred cycles for accessing main memory.[23] If there is an L1 *miss*, the processor checks the next level in the cache hierarchy: the level 2 (L2) cache. An L2 hit would cause the cache line containing the data to be loaded into the

23. Main memory refers to the system's installed and available dynamic memory (DRAM).

L1 cache and then into the appropriate register. The 970FX does not have level 3 (L3) caches, but if it did, similar steps would be repeated for the L3 cache. If none of the caches contains the requested data, the processor must access main memory.

As a cache line's worth of data is loaded into L1, a resident cache line must be flushed to make room for the new cache line. The 970FX uses a pseudo-least-recently-used (LRU) algorithm[24] to determine which cache line to evict. Unless instructed otherwise, the evicted cache line is sent to the L2 cache, which makes L2 a *victim* cache. Table 3–5 shows the important properties of the 970FX's caches.

TABLE 3–5 970FX Caches

Property	L1 I-cache	L1 D-cache	L2 Cache
Size	64KB	32KB	512KB
Type	Instructions	Data	Data and instructions
Associativity	Direct-mapped	Two-way set-associative	Eight-way set-associative
Line size	128 bytes	128 bytes	128 bytes
Sector size	32 bytes	—	—
Number of cache lines	512	256	4096
Number of sets	512	128	512
Granularity	1 cache line	1 cache line	1 cache line
Replacement policy	—	LRU	LRU
Store policy	—	Write-through, with no allocate-on-store-miss	Write-back, with allocate-on-store-miss
Index	Effective address	Effective address	Physical address
Tags	Physical address	Physical address	Physical address
Inclusivity	—	—	Inclusive of L1 D-cache
Hardware coherency	No	Yes	Yes, standard MERSI cache-coherency protocol
Enable bit	Yes	Yes	No

(continues)

24. The 970FX allows the data-cache replacement algorithm to be changed from LRU to FIFO through a bit in a hardware-dependent register.

TABLE 3–5 970FX Caches *(Continued)*

Property	L1 I-cache	L1 D-cache	L2 Cache
Reliability, availability, and serviceability (RAS)	Parity, with invalidate-on-error for data and tags	Parity, with invalidate-on-error for data and tags	ECC on data, parity on tags
Cache locking	No	No	No
Demand load latencies (typical)	—	3, 5, 4, 5 cycles for GPRs, FPRs, VPERM, and VALU, respectively[a]	11, 12, 11, 11 cycles for GPRs, FPRs, VPERM, and VALU, respectively[a]

a. Section 3.3.6.1 discusses GPRs and FPRs. Section 3.3.10.2 discusses VPERM and VALU.

Harvard Architecture

The 970FX's L1 cache is split into separate caches for instructions and data. This design aspect is referred to as the *Harvard Architecture*, alluding to the separate memories for instructions and data in the Mark-III and Mark-IV vacuum tube machines that originated at Harvard University in the 1940s.

You can retrieve processor cache information using the `sysctl` command on Mac OS X as shown in Figure 3–5. Note that the `hwprefs` command is part of Apple's CHUD Tools package.

FIGURE 3–5 Retrieving processor cache information using the `sysctl` command

```
$ sudo hwprefs machine_type # Power Mac G5 Dual 2.5GHz
PowerMac7,3
$ sysctl -a hw
...
hw.cachelinesize: 128
hw.l1icachesize: 65536
hw.l1dcachesize: 32768
hw.l2settings = 2147483648
hw.l2cachesize: 524288
...
$ sudo hwprefs machine_type # Power Mac G5 Quad 2.5GHz
PowerMac11,2
$ sysctl -a hw
...
hw.cachelinesize = 128
hw.l1icachesize = 65536
hw.l1dcachesize = 32768
hw.l2settings = 2147483648
hw.l2cachesize = 1048576
...
```

3.3.2.2 Cache Properties

Let us look more closely at some of the cache-related terminology used in Table 3–5.

Associativity

As we saw earlier, the granularity of operation for a cache—that is, the unit of memory transfers in and out of a cache—is a cache line (also called a *block*). The cache line size on the 970FX is 128 bytes for both the L1 and L2 caches. The *associativity* of a cache is used to determine where to place a cache line's worth of memory in the cache.

If a cache is *m-way set-associative*, then the total space in the cache is conceptually divided into *sets*, with each set containing *m* cache lines. In a set-associative cache, a block of memory can be placed only in certain locations in the cache: It is first mapped to a set in the cache, after which it can be stored in any of the cache lines within that set. Typically, given a memory block with address B, the target set is calculated using the following modulo operation:

```
target set = B MOD {number of sets in cache}
```

A *direct-mapped* cache is equivalent to a one-way set-associative cache. It has the same number of sets as cache lines. This means a memory block with address B can exist only in one cache line, which is calculated as the following:

```
target cache line = B MOD {number of cache lines in cache}
```

Store Policy

A cache's store policy defines what happens when an instruction writes to memory. In a *write-through* design, such as the 970FX L1 D-cache, information is written to both the cache line and to the corresponding block in memory. There is no L1 D-cache allocation on write misses—the affected block is modified only in the lower level of the cache hierarchy and is not loaded into L1. In a *write-back* design, such as the 970FX L2 cache, information is written only to the cache line—the affected block is written to memory only when the cache line is replaced.

> Memory pages that are contiguous in virtual memory will normally not be contiguous in physical memory. Similarly, given a set of virtual addresses, it is not possible to predict how they will fit in the cache. A related point is that if you take a block of contiguous virtual memory the same size as a cache, say, a 512KB block (the size of the entire L2 cache), there is little chance that it will fit in the L2 cache.

MERSI

Only the L2 cache is physically mapped, although all caches use physical address tags. Stores are always sent to the L2 cache in addition to the L1 cache, as the L2 cache is the data *coherency* point. Coherent memory systems aim to provide the same view of all devices accessing the memory. For example, it must be ensured that processors in a multiprocessor system access the correct data—whether the most up-to-date data resides in main memory or in another processor's cache. Maintaining such coherency in hardware introduces a protocol that requires the processor to "remember" the state of the sharing of cache lines.[25] The L2 cache implements the MERSI cache-coherency protocol, which has the following five states.

1. **M**odified—This cache line is modified with respect to the rest of the memory subsystem.
2. **E**xclusive—This cache line is not cached in any other cache.
3. **R**ecent—The current processor is the most recent reader of this shared cache line.
4. **S**hared—This cache line was cached by multiple processors.
5. **I**nvalid—This cache line is invalid.

RAS

The caches incorporate parity-based error detection and correction mechanisms. *Parity bits* are additional bits used along with normal information to detect and correct errors in the transmission of that information. In the simplest case, a single parity bit is used to detect an error. The basic idea in such parity checking is to add an extra bit to each unit of information—say, to make the number of 1s in each unit either odd or even. Now, if a single error (actually, an odd number of errors) occurs during information transfer, the parity-protected information unit would be invalid. In the 970FX's L1 cache, parity errors are reported as cache misses and therefore are implicitly handled by refetching the cache line from the L2 cache. Besides parity, the L2 cache implements an error detection and correction scheme that can detect double errors and correct single errors by using a *Hamming code*.[26] When a single error is detected during an L2 fetch request, the

25. Cache-coherency protocols are primarily either directory-based or snooping-based.

26. A Hamming code is an *error-correcting code*. It is an algorithm in which a sequence of numbers can be expressed such that any errors that appear in certain numbers (say, on the receiving side after the sequence was transmitted by one party to another) can be detected, and corrected, subject to certain limits, based on the remaining numbers.

bad data is corrected and actually written back to the L2 cache. Thereafter, the good data is refetched from the L2 cache.

3.3.3 Memory Management Unit (MMU)

During virtual memory operation, software-visible memory addresses must be *translated* to real (or physical) addresses, both for instruction accesses and for data accesses generated by load/store instructions. The 970FX uses a two-step address translation mechanism[27] based on *segments* and *pages*. In the first step, a software-generated 64-bit *effective address* (EA) is translated to a 65-bit *virtual address* (VA) using the segment table, which lives in memory. *Segment table entries* (STEs) contain *segment descriptors* that define virtual addresses of segments. In the second step, the virtual address is translated to a 42-bit *real address* (RA) using the hashed *page table*, which also lives in memory.

The 32-bit PowerPC architecture provides 16 segment registers through which the 4GB virtual address space can be divided into 16 segments of 256MB each. The 32-bit PowerPC implementations use these segment registers to generate VAs from EAs. The 970FX includes a transitional bridge facility that allows a 32-bit operating system to continue using the 32-bit PowerPC implementation's segment register manipulation instructions. Specifically, the 970FX allows software to associate segments 0 through 15 with any of the 2^{37} available virtual segments. In this case, the first 16 entries of the segment lookaside buffer (SLB), which is discussed next, act as the 16 segment registers.

3.3.3.1 SLB and TLB

We saw that the segment table and the page table are memory-resident. It would be prohibitively expensive if the processor were to go to main memory not only for data fetching but also for address translation. Caching exploits the principle of locality of memory. If caching is effective, then address translations will also have the same locality as memory. The 970FX includes two on-chip buffers for caching recently used segment table entries and page address translations: the *segment lookaside buffer* (SLB) and the *translation lookaside buffer* (TLB), respectively. The SLB is a 64-entry, fully associative cache. The TLB is a 1024-entry,

27. The 970FX also supports a real addressing mode, in which physical translation can be effectively disabled.

four-way set-associative cache with parity protection. It also supports large pages (see Section 3.3.3.4).

3.3.3.2 Address Translation

Figure 3–6 depicts address translation in the 970FX MMU, including the roles of the SLB and the TLB. The 970FX MMU uses 64-bit or 32-bit effective addresses, 65-bit virtual addresses, and 42-bit physical addresses. The presence of the DART introduces another address flavor, the I/O address, which is an address in a 32-bit address space that maps to a larger physical address space.

> Technically, a computer architecture has three (and perhaps more) types of memory addresses: the processor-visible *physical address*, the software-visible *virtual address*, and the *bus address*, which is visible to an I/O device. In most cases (especially on 32-bit hardware), the physical and bus addresses are identical and therefore not differentiated.

The 65-bit extended address space is divided into pages. Each page is mapped to a physical page. A 970FX page table can be as large as 2^{31} bytes (2GB), containing up to 2^{24} (16 million) page table entry groups (PTEGs), where each PTEG is 128 bytes.

As Figure 3–6 shows, during address translation, the MMU converts program-visible effective addresses to real addresses in physical memory. It uses a part of the effective address (the effective segment ID) to locate an entry in the segment table. It first checks the SLB to see if it contains the desired STE. If there is an SLB miss, the MMU searches for the STE in the memory-resident segment table. If the STE is still not found, a memory access fault occurs. If the STE is found, a new SLB entry is allocated for it. The STE represents a segment descriptor, which is used to generate the 65-bit virtual address. The virtual address has a 37-bit virtual segment ID (VSID). Note that the page index and the byte offset in the virtual address are the same as in the effective address. The concatenation of the VSID and the page index forms the virtual page number (VPN), which is used for looking up in the TLB. If there is a TLB miss, the memory-resident page table is looked up to retrieve a page table entry (PTE), which contains a real page number (RPN). The RPN, along with the byte offset carried over from the effective address, forms the physical address.

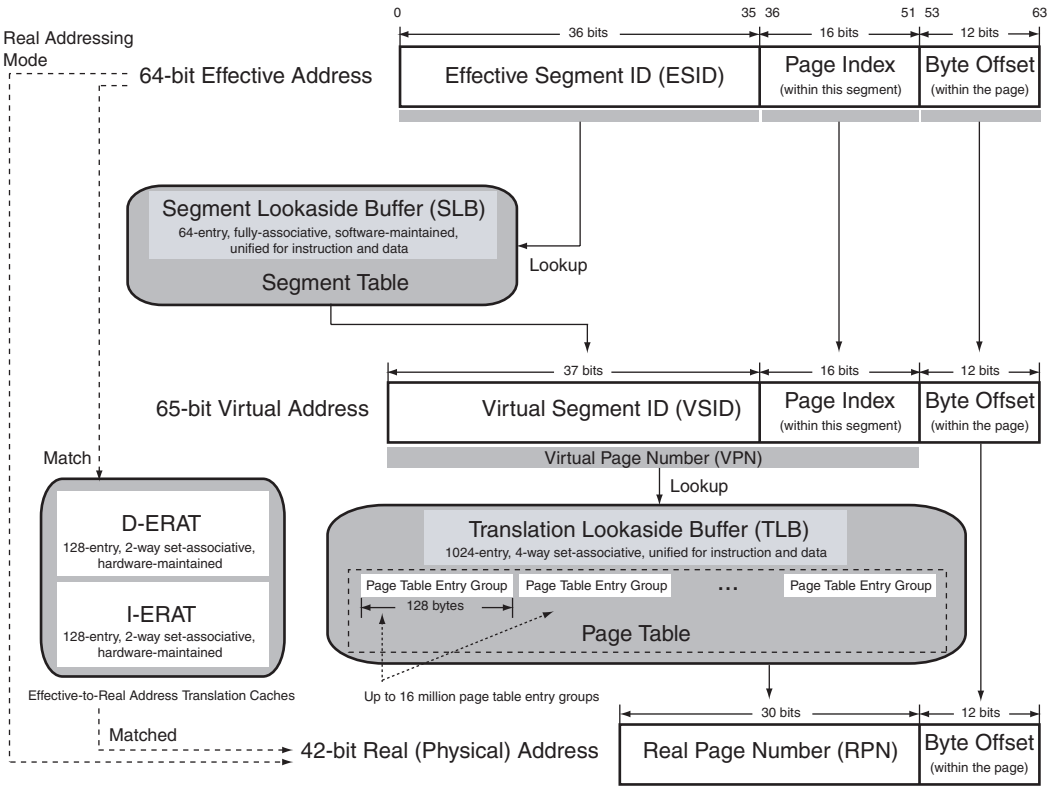

FIGURE 3–6 Address translation in the 970FX MMU

The 970FX allows setting up the TLB to be direct-mapped by setting a particular bit of a hardware-implementation-dependent register.

3.3.3.3 Caching the Caches: ERATs

Information from the SLB and the TLB may be cached in two *effective-to-real address translation* caches (ERATs)—one for instructions (I-ERAT) and another for data (D-ERAT). Both ERATs are 128-entry, two-way set-associative caches. Each ERAT entry contains effective-to-real address translation information for a 4KB block of storage. Both ERATs contain invalid information upon power-on. As shown in Figure 3–6, the ERATs represent a shortcut path to the physical address when there is a match for the effective address in the ERATs.

3.3.3.4 Large Pages

Large pages are meant for use by high-performance computing (HPC) applications. The typical page size of 4KB could be detrimental to memory performance in certain circumstances. If an application's locality of reference is too wide, 4KB pages may not capture the locality effectively enough. If too many TLB misses occur, the consequent TLB entry allocations and the associated delays would be undesirable. Since a large page represents a much larger memory range, the number of TLB hits should increase, as the TLB would now cache translations for larger virtual memory ranges.

It is an interesting problem for the operating system to make large pages available to applications. Linux provides large-page support through a pseudo file system (*hugetlbfs*) that is backed by large pages. The superuser must explicitly configure some number of large pages in the system by preallocating physically contiguous memory. Thereafter, the hugetlbfs instance can be mounted on a directory, which is required if applications intend to use the `mmap()` system call to access large pages. An alternative is to use shared memory calls—`shmat()` and `shmget()`. Files may be created, deleted, `mmap()`'ed, and `munmap()`'ed on hugetlbfs. It does not support reads or writes, however. AIX also requires separate, dedicated physical memory for large-page use. An AIX application can use large pages either via shared memory, as on Linux, or by requesting that the application's data and heap segments be backed by large pages.

Note that whereas the 970FX TLB supports large pages, the ERATs do not; large pages require multiple entries—corresponding to each referenced 4KB block of a large page—in the ERATs. Cache-inhibited accesses to addresses in large pages are not permitted.

3.3.3.5 No Support for Block Address Translation Mechanism

The 970FX does not support the Block Address Translation (BAT) mechanism that is supported in earlier PowerPC processors such as the G4. BAT is a software-controlled array used for mapping large—often much larger than a page—virtual address ranges into contiguous areas of physical memory. The entire map will have the same attributes, including access protection. Thus, the BAT mechanism is meant to reduce address translation overhead for large, contiguous regions of special-purpose virtual address spaces. Since BAT does not use pages, such memory cannot be paged normally. A good example of a scenario where BAT is useful is that of a region of framebuffer memory, which could be memory-mapped effectively via BAT. Software can select block sizes ranging from 128KB to 256MB.

On PowerPC processors that implement BAT, there are four BAT registers each for data (DBATs) and instructions (IBATs). A BAT register is actually a pair of *upper* and *lower* registers, which are accessible from supervisor mode. The eight pairs are named DBAT0U-DBAT3U, DBAT0L-DBAT3L, IBAT0U-IBAT3U, and IBAT0L-IBAT3L. The contents of a BAT register include a block effective page index (BEPI), a block length (BL), and a block real page number (BRPN). During BAT translation, a certain number of high-order bits of the EA—as specified by BL—are matched against each BAT register. If there is a match, the BRPN value is used to yield the RA from the EA. Note that BAT translation is used over page table translation for storage locations that have mappings in both a BAT register and the page table.

3.3.4 Miscellaneous Internal Buffers and Queues

The 970FX contains several miscellaneous buffers and queues internal to the processor, most of which are not visible to software. Examples include the following:

- A 4-entry (128 bytes per entry) Instruction Prefetch Queue logically above the L1 I-cache
- Fetch buffers in the Instruction Fetch Unit and the Instruction Decode Unit
- An 8-entry Load Miss Queue (LMQ) that tracks loads that missed the L1 cache and are waiting to receive data from the processor's storage subsystem
- A 32-entry Store Queue (STQ)[28] for holding stores that can be written to cache or memory later
- A 32-entry Load Reorder Queue (LRQ) in the Load/Store Unit (LSU) that holds physical addresses for tracking the order of loads and watching for hazards
- A 32-entry Store Reorder Queue (SRQ) in the LSU that holds physical addresses and tracks all active stores
- A 32-entry Store Data Queue (SDQ) in the LSU that holds a double word of data
- A 12-entry Prefetch Filter Queue (PFQ) for detecting data streams for prefetching
- An 8-entry (64 bytes per entry) fully associative Store Queue for the L2 cache controller

28. The STQ supports forwarding.

3.3.5 Prefetching

Cache miss rates can be reduced through a technique called *prefetching*—that is, fetching information before the processor requests it. The 970FX prefetches instructions and data to hide memory latency. It also supports software-initiated prefetching of up to eight data streams called *hardware streams*, four of which can optionally be vector streams. A stream is defined as a sequence of loads that reference more than one contiguous cache line.

The prefetch engine is a functionality of the Load/Store Unit. It can detect sequential access patterns in ascending or descending order by monitoring loads and recording cache line addresses when there are cache misses. The 970FX does not prefetch store misses.

Let us look at an example of the prefetch engine's operation. Assuming no prefetch streams are active, the prefetch engine will act when there is an L1 D-cache miss. Suppose the miss was for a cache line with address A; then the engine will create an entry in the Prefetch Filter Queue (PFQ)[29] with the address of either the next or the previous cache line—that is, either A + 1 or A – 1. It guesses the direction (up or down) based on whether the memory access was located in the top 25% of the cache line (guesses down) or the bottom 75% of the cache line (guesses up). If there is another L1 D-cache miss, the engine will compare the line address with the entries in the PFQ. If the access is indeed sequential, the line address now being compared must be either A + 1 or A – 1. Alternatively, the engine could have incorrectly guessed the direction, in which case it would create another filter entry for the opposite direction. If the guessed direction was correct (say, up), the engine deems it a sequential access and allocates a stream entry in the Prefetch Request Queue (PRQ)[30] using the next available stream identifier. Moreover, the engine will initiate prefetching for cache line A + 2 to L1 and cache line A + 3 to L2. If A + 2 is read, the engine will cause A + 3 to be fetched to L1 from L2, and A + 4, A + 5, and A + 6 to be fetched to L2. If further sequential demand-reads occur (for A + 3 next), this pattern will continue until all streams are assigned. The PFQ is updated using an LRU algorithm.

The 970FX allows software to manipulate the prefetch mechanism. This is useful if the programmer knows data access patterns ahead of time. A version of the *data-cache-block-touch* (`dcbt`) instruction, which is one of the storage con-

29. The PFQ is a 12-entry queue for detecting data streams for prefetching.

30. The PRQ is a queue of eight streams that will be prefetched.

trol instructions, can be used by a program to provide hints that it intends to read from a specified address or data stream in the near future. Consequently, the processor would initiate a data stream prefetch from a particular address.

Note that if you attempt to access unmapped or protected memory via software-initiated prefetching, no page faults will occur. Moreover, these instructions are not guaranteed to succeed and can fail silently for a variety of reasons. In the case of success, no result is returned in any register—only the cache block is fetched. In the case of failure, no cache block is fetched, and again, no result is returned in any register. In particular, failure does not affect program correctness; it simply means that the program will not benefit from prefetching.

Prefetching continues until a page boundary is reached, at which point the stream will have to be reinitialized. This is so because the prefetch engine does not know about the effective-to-real address mapping and can prefetch only within a real page. This is an example of a situation in which large pages—with page boundaries that are 16MB apart—will fare better than 4KB pages.

On a Mac OS X system with AltiVec hardware, you can use the `vec_dst()` AltiVec function to initiate data read of a line into cache, as shown in the pseudocode in Figure 3–7.

FIGURE 3–7 Data prefetching in AltiVec

```
while (/* data processing loop */) {

    /* prefetch */
    vec_dst(address + prefetch_lead, control, stream_id);

    /* do some processing */

    /* advance address pointer */
}

/* stop the stream */
vec_dss(stream_id);
```

The `address` argument to `vec_dst()` is a pointer to a byte that lies within the first cache line to be fetched; the `control` argument is a word whose bits specify the block size, the block count, and the distance between the blocks; and the `stream_id` specifies the stream to use.

3.3.6 Registers

The 970FX has two *privilege modes* of operation: a user mode (*problem state*) and a supervisor mode (*privileged state*). The former is used by user-space applications, whereas the latter is used by the Mac OS X kernel. When the processor is first initialized, it comes up in supervisor mode, after which it can be switched to user mode via the Machine State Register (MSR).

The set of architected registers can be divided into three levels (or models) in the PowerPC architecture:

1. User Instruction Set Architecture (UISA)
2. Virtual Environment Architecture (VEA)
3. Operating Environment Architecture (OEA)

The UISA and VEA registers can be accessed by software through either user-level or supervisor-level privileges, although there are VEA registers that cannot be written to by user-level instructions. OEA registers can be accessed only by supervisor-level instructions.

3.3.6.1 UISA and VEA Registers

Figure 3–8 shows the UISA and VEA registers of the 970FX. Their purpose is summarized in Table 3–6. Note that whereas the general-purpose registers are all 64-bit wide, the set of supervisor-level registers contains both 32-bit and 64-bit registers.

Processor registers are used with all normal instructions that access memory. In fact, there are no computational instructions in the PowerPC architecture that modify storage. For a computational instruction to use a storage operand, it must first load the operand into a register. Similarly, if a computational instruction writes a value to a storage operand, the value must go to the target location via a register. The PowerPC architecture supports the following addressing modes for such instructions.

- *Register Indirect*—The effective address EA is given by `(rA | 0)`.
- *Register Indirect with Immediate Index*—EA is given by `(rA | 0) + offset`, where `offset` can be zero.
- *Register Indirect with Index*—EA is given by `(rA | 0) + rB`.

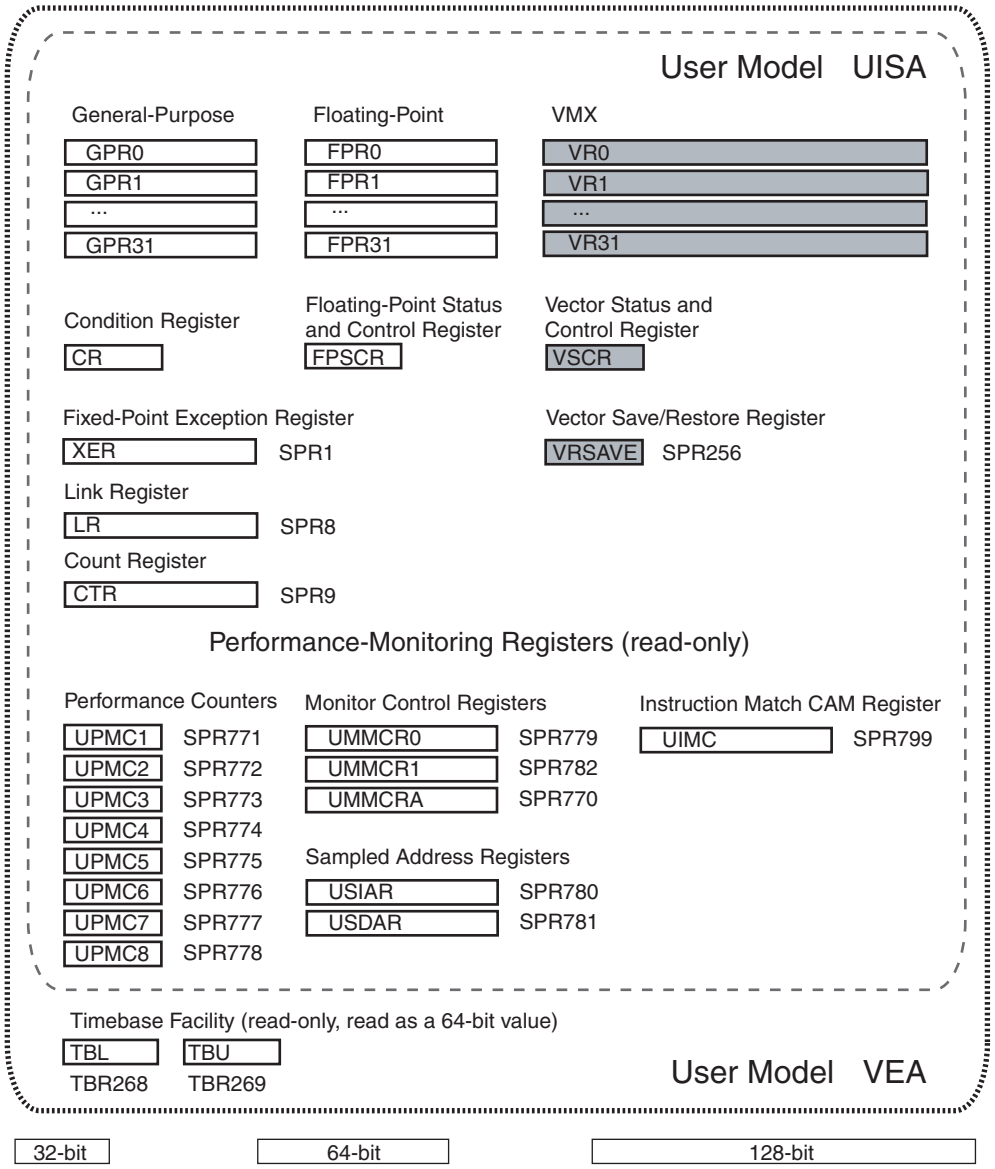

FIGURE 3–8 PowerPC UISA and VEA registers

TABLE 3–6 UISA and VEA Registers

Name	Width	Count	Notes
General-Purpose Registers (GPRs)	64-bit	32	GPRs are used as source or destination registers for fixed-point operations—e.g., by fixed-point load/store instructions. You also use GPRs while accessing special-purpose registers (SPRs). Note that GPR0 is not hardwired to the value 0, as is the case on several RISC architectures.
Floating-Point Registers (FPRs)	64-bit	32	FPRs are used as source or destination registers for floating-point instructions. You also use FPRs to access the Floating-Point Status and Control Register (FPSCR). An FPR can hold integer, single-precision floating-point, or double-precision floating-point values.
Vector Registers (VRs)	128-bit	32	VRs are used as vector source or destination registers for vector instructions.
Integer Exception Register (XER)	32-bit	1	The XER is used to indicate carry conditions and overflows for integer operations. It is also used to specify the number of bytes to be transferred by a *load-string-word-indexed* (`lswx`) or *store-string-word-indexed* (`stswx`) instruction.
Floating-Point Status and Control Register (FPSCR)	32-bit	1	The FPSCR is used to record floating-point exceptions and the result type of a floating-point operation. It is also used to toggle the reporting of floating-point exceptions and to control the floating-point rounding mode.
Vector Status and Control Register (VSCR)	32-bit	1	Only two bits of the VSCR are defined: the saturate (SAT) bit and the non-Java mode (NJ) bit. The SAT bit indicates that a vector saturating-type instruction generated a saturated result. The NJ bit, if cleared, enables a Java-IEEE-C9X-compliant mode for vector floating-point operations that handles denormalized values in accordance with these standards. When the NJ bit is set, a potentially faster mode is selected, in which the value 0 is used in place of denormalized values in source or result vectors.
Condition Register (CR)	32-bit	1	The CR is conceptually divided into eight 4-bit fields (CR0–CR7). These fields store results of certain fixed-point and floating-point operations. Some branch instructions can test individual CR bits.
Vector Save/Restore Register (VRSAVE)	32-bit	1	The VRSAVE is used by software while saving and restoring VRs across context-switching events. Each bit of the VRSAVE corresponds to a VR and specifies whether that VR is in use or not.

(continues)

TABLE 3–6 UISA and VEA Registers *(Continued)*

Name	Width	Count	Notes
Link Register (LR)	64-bit	1	The LR can be used to return from a subroutine—it holds the return address after a branch instruction if the link (LK) bit in that branch instruction's encoding is 1. It is also used to hold the target address for the *branch-conditional-to-Link-Register* (`bclrx`) instruction. Some instructions can automatically load the LR to the instruction following the branch.
Count Register (CTR)	64-bit	1	The CTR can be used to hold a loop count that is decremented during execution of branch instructions. The *branch-conditional-to-Count-Register* (`bcctrx`) instruction branches to the target address held in this register.
Timebase Registers (TBL, TBU)	32-bit	2	The Timebase (TB) Register, which is the concatenation of the 32-bit TBU and TBL registers, contains a periodically incrementing 64-bit unsigned integer.

`rA` and `rB` represent register contents. The notation `(rA | 0)` means the contents of register `rA` unless `rA` is GPR0, in which case `(rA | 0)` is taken to be the value `0`.

The UISA-level performance-monitoring registers provide user-level read access to the 970FX's performance-monitoring facility. They can be written only by a supervisor-level program such as the kernel or a kernel extension.

Apple's Computer Hardware Understanding Development (CHUD) is a suite of programs (the "CHUD Tools") for measuring and optimizing performance on Mac OS X. The software in the CHUD Tools package makes use of the processor's performance-monitoring counters.

The Timebase Register

The Timebase (TB) provides a long-period counter driven by an implementation-dependent frequency. The TB is a 64-bit register containing an unsigned 64-bit integer that is incremented periodically. Each increment adds 1 to bit 63 (the lowest-order bit) of the TB. The maximum value that the TB can hold is $2^{64} - 1$, after which it resets to zero without generating any exception. The TB can either be incremented at a frequency that is a function of the processor clock frequency, or it can

be driven by the rising edge of the signal on the TB enable (TBEN) input pin.[31] In the former case, the 970FX increments the TB once every eight full frequency processor clocks. It is the operating system's responsibility to initialize the TB. The TB can be read—but not written to—from user space. The program shown in Figure 3–9 retrieves and prints the TB.

FIGURE 3–9 Retrieving and displaying the Timebase Register

```
// timebase.c

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

u_int64_t mftb64(void);
void mftb32(u_int32_t *, u_int32_t *);

int
main(void)
{
    u_int64_t tb64;
    u_int32_t tb32u, tb32l;

    tb64 = mftb64();
    mftb32(&tb32u, &tb32l);

    printf("%llx %x%08x\n", tb64, tb32l, tb32u);
    exit(0);
}

// Requires a 64-bit processor
// The TBR can be read in a single instruction (TBU || TBL)
u_int64_t
mftb64(void)
{
    u_int64_t tb64;

    __asm("mftb %0\n\t"
          : "=r" (tb64)
          :
    );

    return tb64;
}
```

(continues)

31. In this case, the TB frequency may change at any time.

FIGURE 3–9 Retrieving and displaying the Timebase Register *(continued)*

```
// 32-bit or 64-bit
void
mftb32(u_int32_t *u, u_int32_t *l)
{
    u_int32_t tmp;

    __asm(
    "loop:              \n\t"
        "mftbu    %0    \n\t"
        "mftb     %1    \n\t"
        "mftbu    %2    \n\t"
        "cmpw     %2,%0 \n\t"
        "bne      loop  \n\t"
        : "=r"(*u), "=r"(*l), "=r"(tmp)
        :
    );
}


$ gcc -Wall -o timebase timebase.c
$ ./timebase; ./timebase; ./timebase; ./timebase; ./timebase
b6d10de300000001 b6d10de4000002d3
b6d4db7100000001 b6d4db72000002d3
b6d795f700000001 b6d795f8000002d3
b6da5a3000000001 b6da5a31000002d3
b6dd538c00000001 b6dd538d000002d3
```

Note in Figure 3–9 that we use inline assembly rather than create a separate assembly source file. The GNU assembler inline syntax is based on the template shown in Figure 3–10.

FIGURE 3–10 Code template for inline assembly in the GNU assembler

```
__asm__ volatile(
        "assembly statement 1\n"
        "assembly statement 2\n"
        ...
        "assembly statement N\n"
    :   outputs, if any
    :   inputs, if any
    :   clobbered registers, if any
);
```

We will come across other examples of inline assembly in this book.

Viewing Register Contents: The Mac OS X Way

The contents of the TBR, along with those of several configuration registers, memory management registers, performance-monitoring registers, and miscellaneous registers can be viewed using the Reggie SE graphical application (`Reggie SE.app`), which is part of the CHUD Tools package. Reggie SE can also display physical memory contents and details of PCI devices.

3.3.6.2 OEA Registers

The OEA registers are shown in Figure 3–11. Examples of their use include the following.

- The bit-fields of the *Machine State Register* (MSR) are used to define the processor's state. For example, MSR bits are used to specify the processor's computation mode (32-bit or 64-bit), to enable or disable power management, to determine whether the processor is in *privileged* (supervisor) or *nonprivileged* (user) mode, to enable single-step tracing, and to enable or disable address translation. The MSR can be explicitly accessed via the *move-to-MSR* (`mtmsr`), *move-to-MSR-double* (`mtmsrd`), and *move-from-MSR* (`mfmsr`) instructions. It is also modified by the *system-call* (`sc`) and *return-from-interrupt-double* (`rfid`) instructions.
- The *Hardware-Implementation-Dependent* (HID) registers allow very fine-grained control of the processor's features. Bit-fields in the various HID registers can be used to enable, disable, or alter the behavior of processor features such as branch prediction mode, data prefetching, instruction cache, and instruction prefetch mode and also to specify which data cache replacement algorithm to use (LRU or first-in first-out [FIFO]), whether the Timebase is externally clocked, and whether large pages are disabled.
- The *Storage Description Register* (SDR1) is used to hold the page table base address.

3.3.7 Rename Registers

The 970FX implements a substantial number of *rename registers*, which are used to handle register-name dependencies. Instructions can depend on one another from the point of view of *control*, *data*, or *name*. Consider two instructions, say, `I1` and `I2`, in a program, where `I2` comes after `I1`.

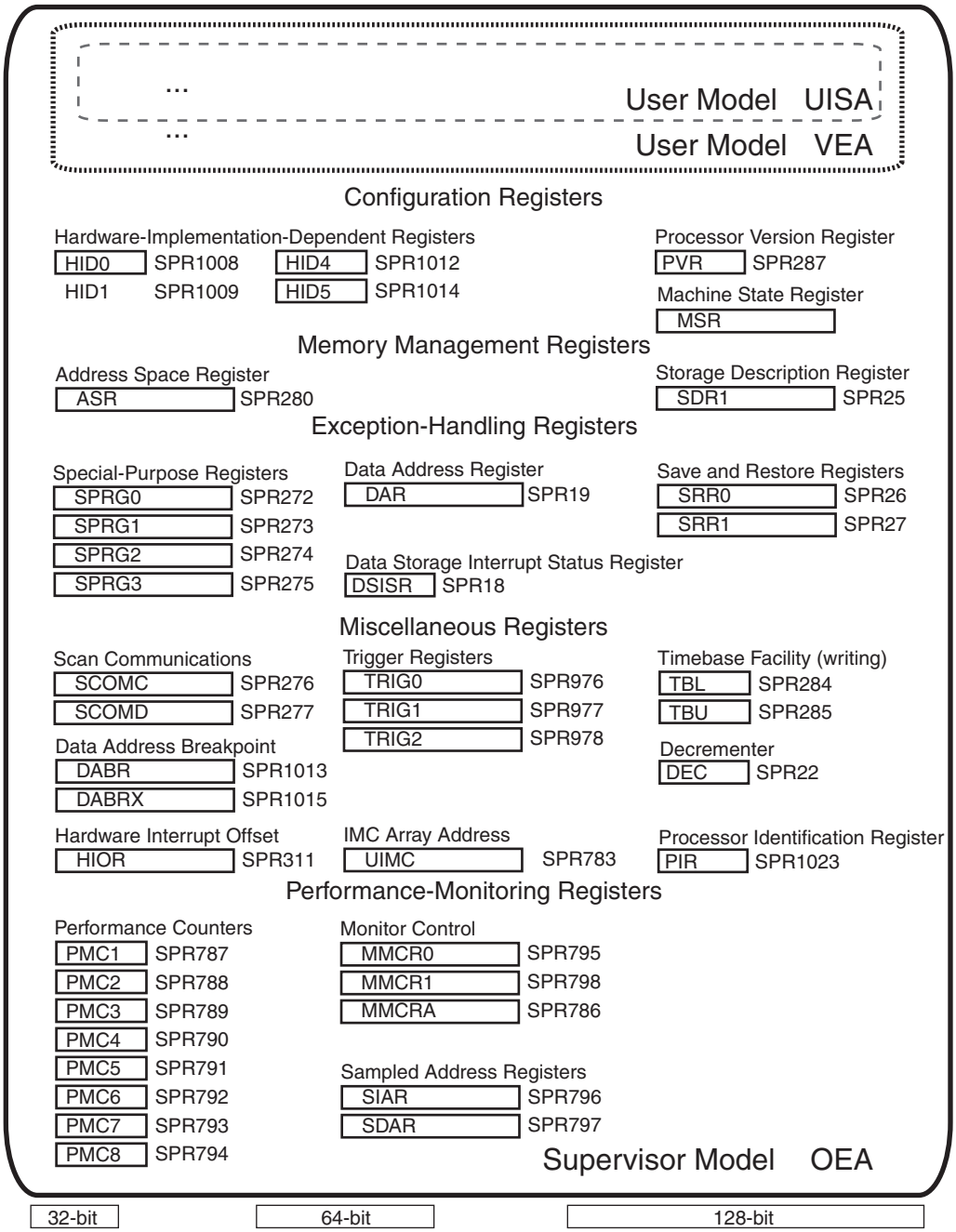

FIGURE 3–11 PowerPC OEA registers

```
I1
...
Ix
...
I2
```

In a data dependency, `I2` either uses a result produced by `I1`, or `I2` has a data dependency on an instruction `Ix`, which in turn has a data dependency on `I1`. In both cases, a value is effectively transmitted from `I1` to `I2`.

In a name dependency, `I1` and `I2` use the same logical resource or *name*, such as a register or a memory location. In particular, if `I2` writes to the same register that is either read from or written to by `I1`, then `I2` would have to wait for `I1` to execute before it can execute. These are known as *write-after-read* (WAR) and *write-after-write* (WAW) hazards.

```
I1 reads (or writes) <REGISTER X>
...
I2 writes <REGISTER X>
```

In this case, the dependency is not "real" in that `I2` does not *need* `I1`'s result. One solution to handle register-name dependencies is to *rename* the conflicting register used in the instructions so that they become independent. Such renaming could be done in software (statically, by the compiler) or in hardware (dynamically, by logic in the processor). The 970FX uses pools of physical *rename registers* that are assigned to instructions during the *mapping* stage in the processor pipeline and released when they are no longer needed. In other words, the processor internally renames *architected* registers used by instructions to *physical* registers. This makes sense only when the number of physical registers is (substantially) larger than the number of architected registers. For example, the PowerPC architecture has 32 GPRs, but the 970FX implementation has a pool of 80 physical GPRs, from which the 32 architected GPRs are assigned. Let us consider a specific example, say, of a WAW hazard, where renaming is helpful.

```
; before renaming
r20 ← r21 + r22 ; r20 is written to
...
r20 ← r23 + r24 ; r20 is written to... WAW hazard here
r25 ← r20 + r26 ; r20 is read from

; after renaming
r20 ← r21 + r22 ; r20 is written to
...
r64 ← r23 + 424 ; r20 is renamed to r64... no WAW hazard now
r25 ← r64 + r26 ; r20 is renamed to r64
```

Renaming is also beneficial to speculative execution, since the processor can use the extra physical registers to reduce the amount of architected register state it must save to recover from incorrectly speculated execution.

Table 3–7 lists the available renamed registers in the 970FX. The table also mentions *emulation* registers, which are available to cracked and microcoded instructions, which, as we will see in Section 3.3.9.1, are processes by which complex instructions are broken down into simpler instructions.

TABLE 3–7 Rename Register Resources

Resource	Architected (Logical Resource)	Emulation (Logical Resource)	Rename Pool (Physical Resource)
GPRs	32×64-bit	4×64-bit	80×64-bit.
VRSAVE	1×32-bit	—	Shared with the GPR rename pool.
FPRs	32×64-bit	1×64-bit	80×64-bit.
FPSCR	1×32-bit	—	One rename per active instruction group using a 20-entry buffer.
LR	1×64-bit	—	16×64-bit.
CTR	1×64-bit	—	LR and CTR share the same rename pool.
CR	8×4-bit	1×4-bit	32×4-bit.
XER	1×32-bit	—	24×2-bit. Only two bits—the overflow bit OV and the carry bit CA—are renamed from a pool of 24 2-bit registers.
VRs	32×128-bit	—	80×128-bit.
VSCR	1×32-bit	—	20×1-bit. Of the VSCR's two defined bits, only the SAT bit is renamed from a pool of 20 1-bit registers.

3.3.8 Instruction Set

All PowerPC instructions are 32 bits wide regardless of whether the processor is in 32-bit or 64-bit computation mode. All instructions are *word aligned*, which means that the two lowest-order bits of an instruction address are irrelevant from the processor's standpoint. There are several instruction formats, but bits 0 through 5 of an instruction word always specify the *major opcode*. PowerPC instructions typically have three operands: two source operands and one result.

One of the source operands may be a constant or a register, but the other operands are usually registers.

We can broadly divide the instruction set implemented by the 970FX into the following instruction categories: fixed-point, floating-point, vector, control flow, and everything else.

3.3.8.1 Fixed-Point Instructions

Operands of fixed-point instructions can be bytes (8-bit), half words (16-bit), words (32-bit), or double words (64-bit). This category includes the following instruction types:

- Fixed-point load and store instructions for moving values between the GPRs and storage
- Fixed-point *load-multiple-word* (`lmw`) and *store-multiple-word* (`stmw`), which can be used for restoring or saving up to 32 GPRs in a single instruction
- Fixed-point *load-string-word-immediate* (`lswi`), *load-string-word-indexed* (`lswx`), *store-string-word-immediate* (`stswi`), and *store-string-word-indexed* (`stswx`), which can be used to fetch and store fixed- and variable-length strings, with arbitrary alignments
- Fixed-point arithmetic instructions, such as *add*, *divide*, *multiply*, *negate*, and *subtract*
- Fixed-point compare instructions, such as *compare-algebraic*, *compare-algebraic-immediate*, *compare-algebraic-logical*, and *compare-algebraic-logical-immediate*
- Fixed-point logical instructions, such as *and*, *and-with-complement*, *equivalent*, *or*, *or-with-complement*, *nor*, *xor*, *sign-extend*, and *count-leading-zeros* (`cntlzw` and variants)
- Fixed-point rotate and shift instructions, such as *rotate*, *rotate-and-mask*, *shift-left*, and *shift-right*
- Fixed-point *move-to-system-register* (`mtspr`), *move-from-system-register* (`mfspr`), *move-to-MSR* (`mtmsr`), and *move-from-MSR* (`mfmsr`), which allow GPRs to be used to access system registers

> Most load/store instructions can optionally update the base register with the effective address of the data operated on by the instruction.

3.3.8.2 Floating-Point Instructions

Floating-point operands can be single-precision (32-bit) or double-precision (64-bit) floating-point quantities. However, floating-point data is always stored in the FPRs in double-precision format. Loading a single-precision value from storage converts it to double precision, and storing a single-precision value to storage actually rounds the FPR-resident double-precision value to single precision. The 970FX complies with the IEEE 754 standard[32] for floating-point arithmetic. This instruction category includes the following types:

- Floating-point load and store instructions for moving values between the FPRs and storage
- Floating-point comparison instructions
- Floating-point arithmetic instructions, such as *add*, *divide*, *multiply*, *multiply-add*, *multiply-subtract*, *negative-multiply-add*, *negative-multiply-subtract*, *negate*, *square-root*, and *subtract*
- Instructions for manipulating the FPSCR, such as *move-to-FPSCR*, *move-from-FPSCR*, *set-FPSCR-bit*, *clear-FPSCR-bit*, and *copy-FPSCR-field-to-CR*
- PowerPC optional floating-point instructions, namely: *floating-square-root* (`fsqrt`), *floating-square-root-single* (`fsqrts`), *floating-reciprocal-estimate-single* (`fres`), *floating-reciprocal-square-root-estimate* (`frsqrte`), and *floating-point-select* (`fsel`)

> The precision of floating-point-estimate instructions (`fres` and `frsqrte`) is less on the 970FX than on the G4. Although the 970FX is at least as accurate as the IEEE 754 standard requires, the G4 is more accurate than required. Figure 3–12 shows a program that can be executed on a G4 and a G5 to illustrate this difference.

3.3.8.3 Vector Instructions

Vector instructions execute in the 128-bit VMX execution unit. We will look at some of the VMX details in Section 3.3.10. The 970FX VMX implementation contains 162 vector instructions in various categories.

32. The IEEE 754 standard governs binary floating-point arithmetic. The standard's primary architect was William Velvel Kahan, who received the Turing Award in 1989 for his fundamental contributions to numerical analysis.

FIGURE 3–12 Precision of the floating-point-estimate instruction on the G4 and the G5

```
// frsqrte.c

#include <stdio.h>
#include <stdlib.h>

double
frsqrte(double n)
{
    double s;

    asm(
        "frsqrte %0, %1"
        : "=f" (s)  /* out */
        : "f"  (n)  /* in */
    );

    return s;
}

int
main(int argc, char **argv)
{
    printf("%8.8f\n", frsqrte(strtod(argv[1], NULL)));
    return 0;
}

$ machine
ppc7450
$ gcc -Wall -o frsqrte frsqrte.c
$ ./frsqrte 0.5
1.39062500

$ machine
ppc970
$ gcc -Wall -o frsqrte frsqrte.c
$ ./frsqrte 0.5
1.37500000
```

3.3.8.4 Control-Flow Instructions

A program's control flow is sequential—that is, its instructions logically execute in the order they appear—until a control-flow change occurs either explicitly (because of an instruction that modifies the control flow of a program) or as a side effect of another event. The following are examples of control-flow changes:

- An explicit *branch* instruction, after which execution continues at the target address specified by the branch
- An *exception*, which could represent an error, a signal external to the processor core, or an unusual condition that sets a status bit but may or may not cause an *interrupt*[33]
- A *trap*, which is an interrupt caused by a trap instruction
- A *system call*, which is a form of software-only interrupt caused by the *system-call* (`sc`) instruction

Each of these events could have *handlers*—pieces of code that handle them. For example, a trap handler may be executed when the conditions specified in the trap instruction are satisfied. When a user-space program executes an `sc` instruction with a valid system call identifier, a function in the operating system kernel is invoked to provide the service corresponding to that system call. Similarly, control flow also changes when the program is returning from such handlers. For example, after a system call finishes in the kernel, execution continues in user space—in a different piece of code.

The 970FX supports *absolute* and *relative* branching. A branch could be *conditional* or *unconditional*. A conditional branch can be based on any of the bits in the CR being 1 or 0. We earlier came across the special-purpose registers LR and CTR. LR can hold the return address on a procedure call. A leaf procedure—one that does not call another procedure—does not need to save LR and therefore can return faster. CTR is used for loops with a fixed iteration limit. It can be used to branch based on its contents—the *loop counter*—being zero or nonzero, while decrementing the counter automatically. LR and CTR are also used to hold target addresses of conditional branches for use with the `bclr` and `bcctr` instructions, respectively.

> Besides performing aggressive dynamic branch prediction, the 970FX allows *hints* to be provided along with many types of branch instructions to improve branch prediction accuracy.

3.3.8.5 Miscellaneous Instructions

The 970FX includes various other types of instructions, many of which are used by the operating system for low-level manipulation of the processor. Examples include the following types:

33. When machine state changes in response to an exception, an interrupt is said to have occurred.

- Instructions for processor management, including direct manipulation of some SPRs
- Instructions for controlling caches, such as for touching, zeroing, and flushing a cache; requesting a store; and requesting a prefetch stream to be initiated—for example: *instruction-cache-block-invalidate* (`icbi`), *data-cache-block-touch* (`dcbt`), *data-cache-block-touch-for-store* (`dcbtst`), *data-cache-block-set-to-zero* (`dcbz`), *data-cache-block-store* (`dcbst`), and *data-cache-block-flush* (`dcbf`)
- Instructions for loading and storing conditionally, such as *load-word-and-reserve-indexed* (`lwarx`), *load-double-word-and-reserve-indexed* (`ldarx`), *store-word-conditional-indexed* (`stwcx.`), and *store-double-word-conditional-indexed* (`stdcx.`)

> The `lwarx` (or `ldarx`) instruction performs a load and sets a reservation bit internal to the processor. This bit is hidden from the programming model. The corresponding store instruction—`stwcx.` (or `stdcx.`)—performs a conditional store if the reservation bit is set and clears the reservation bit.

- Instructions for memory synchronization,[34] such as *enforce-in-order-execution-of-i/o* (`eieio`), *synchronize* (`sync`), and special forms of sync (`lwsync` and `ptesync`)
- Instructions for manipulating SLB and TLB entries, such as *slb-invalidate-all* (`slbia`), *slb-invalidate-entry* (`slbie`), *tlb-invalidate-entry* (`tlbie`), and *tlb-synchronize* (`tlbsync`)

3.3.9 The 970FX Core

The 970FX core is depicted in Figure 3–13. We have come across several of the core's major components earlier in this chapter, such as the L1 caches, the ERATs, the TLB, the SLB, register files, and register-renaming resources.

The 970FX core is designed to achieve a high degree of instruction parallelism. Some of its noteworthy features include the following.

34. During memory synchronization, bit 2 of the CR—the EQ bit—is set to record the successful completion of a store operation.

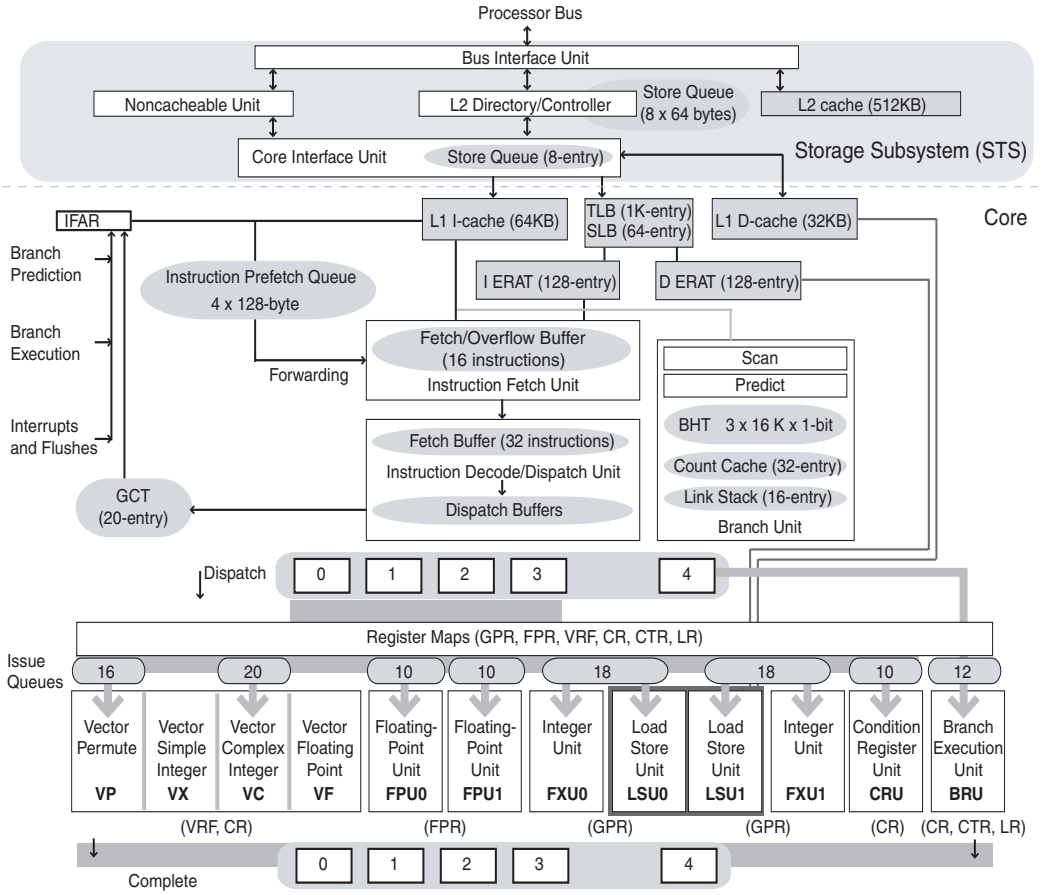

FIGURE 3–13 The core of the 970FX

- It has a highly superscalar 64-bit design, with support for the 32-bit operating-system-bridge[35] facility.
- It performs dynamic "cracking" of certain instructions into two or more simpler instructions.
- It performs highly speculative execution of instructions along with aggressive branch prediction and dynamic instruction scheduling.
- It has twelve logically separate functional units and ten execution pipelines.

35. The "bridge" refers to a set of optional features defined to simplify the migration of 32-bit operating systems to 64-bit implementations.

- It has two Fixed-Point Units (FXU1 and FXU2). Both units are capable of basic arithmetic, logical, shifting, and multiplicative operations on integers. However, only FXU1 is capable of executing divide instructions, whereas only FXU2 can be used in operations involving special purpose registers.
- It has two Floating-Point Units (FPU1 and FPU2). Both units are capable of performing the full supported set of floating-point operations.
- It has two Load/Store Units (LSU1 and LSU2).
- It has a Condition Register Unit (CRU) that executes CR logical instructions.
- It has a Branch Execution Unit (BRU) that computes branch address and branch direction. The latter is compared with the predicted direction. If the prediction was incorrect, the BRU redirects instruction fetching.
- It has a Vector Processing Unit (VPU) with two subunits: a Vector Arithmetic and Logical Unit (VALU) and a Vector Permute Unit (VPERM). The VALU has three subunits of its own: a Vector Simple-Integer[36] Unit (VX), a Vector Complex-Integer Unit (VC), and a Vector Floating-Point Unit (VF).
- It can perform 64-bit integer or floating-point operations in one clock cycle.
- It has deeply pipelined execution units, with pipeline depths of up to 25 stages.
- It has reordering issue queues that allow for out-of-order execution.
- Up to 8 instructions can be fetched in each cycle from the L1 instruction cache.
- Up to 8 instructions can be issued in each cycle.
- Up to 5 instructions can complete in each cycle.
- Up to 215 instructions can be *in flight*—that is, in various stages of execution (partially executed)—at any time.

The processor uses a large number of its resources such as reorder queues, rename register pools, and other logic to track in-flight instructions and their dependencies.

36. Simple integers (non-floating-point) are also referred to as fixed-point. The "X" in "VX" indicates "fixed."

3.3.9.1 Instruction Pipeline

In this section, we will discuss how the 970FX processes instructions. The overall instruction pipeline is shown in Figure 3–14. Let us look at the important stages of this pipeline.

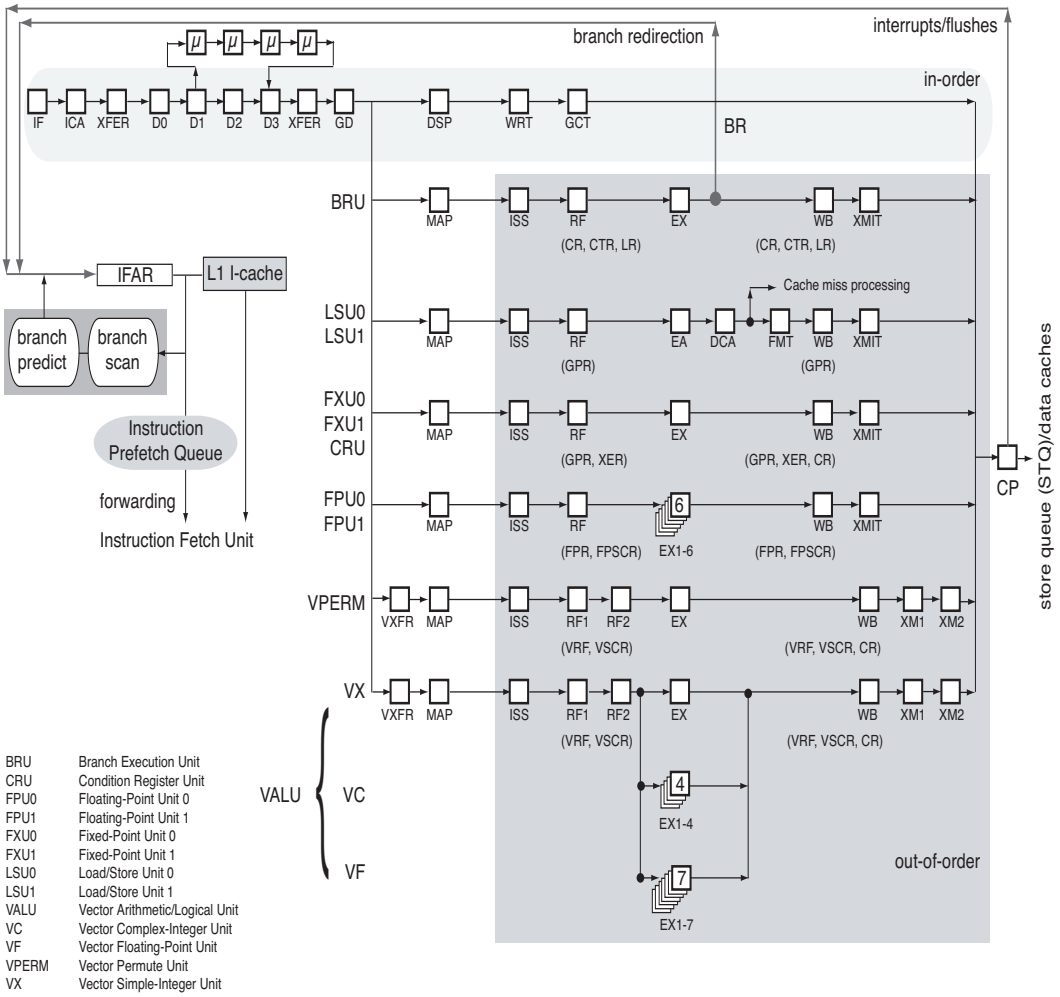

FIGURE 3–14 The 970FX instruction pipeline

IFAR, ICA[37]

Based on the address in the Instruction Fetch Address Register (IFAR), the instruction-fetch-logic fetches eight instructions every cycle from the L1 I-cache into a 32-entry instruction buffer. The eight-instruction block, so fetched, is 32-byte aligned. Besides performing IFAR-based demand fetching, the 970FX prefetches cache lines into a 4×128-byte Instruction Prefetch Queue. If a demand fetch results in an I-cache miss, the 970FX checks whether the instructions are in the prefetch queue. If the instructions are found, they are inserted into the pipeline as if no I-cache miss had occurred. The cache line's critical sector (eight words) is written into the I-cache.

D0

There is logic to partially decode (predecode) instructions after they leave the L2 cache and before they enter the I-cache or the prefetch queue. This process adds five extra bits to each instruction to yield a 37-bit instruction. An instruction's predecode bits mark it as illegal, microcoded, conditional or unconditional branch, and so on. In particular, the bits also specify how the instruction is to be grouped for dispatching.

D1, D2, D3

The 970FX splits complex instructions into two or more internal operations, or *iops*. The iops are more RISC-like than the instructions they are part of. Instructions that are broken into exactly two iops are called *cracked* instructions, whereas those that are broken into three or more iops are called *microcoded* instructions because the processor emulates them using microcode.

> An instruction may not be atomic because the atomicity of cracked or microcoded instructions is at the iop level. Moreover, it is the iops, and not programmer-visible instructions, that are executed out-of-order. This approach allows the processor more flexibility in parallelizing execution. Note that AltiVec instructions are neither cracked nor microcoded.

Fetched instructions go to a 32-instruction fetch buffer. Every cycle, up to five instructions are taken from this buffer and sent through a decode pipeline that

37. Instruction Cache Access.

is either *inline* (consisting of three stages, namely, D1, D2, and D3), or *template-based* if the instruction needs to be microcoded. The template-based decode pipeline generates up to four iops per cycle that emulate the original instruction. In any case, the decode pipeline leads to the formation of an instruction *dispatch group*.

Given the out-of-order execution of instructions, the processor needs to keep track of the program order of all instructions in various stages of execution. Rather than tracking individual instructions, the 970FX tracks instructions in dispatch groups. The 970FX forms such groups containing one to five iops, each occupying an instruction slot (0 through 4) in the group. Dispatch group formation[38] is subject to a long list of rules and conditions such as the following.

- The iops in a group must be in program order, with the oldest instruction being in slot 0.
- A group may contain up to four nonbranch instructions and optionally a branch instruction. When a branch is encountered, it is the last instruction in the current group, and a new group is started.
- Slot 4 can contain only branch instructions. In fact, *no-op* (no-operation) instructions may have to be inserted in the other slots to force a branch instruction to fall in slot 4.
- An instruction that is a branch target is always at the start of a group.
- A cracked instruction takes two slots in a group.
- A microcoded instruction takes an entire group by itself.
- An instruction that modifies an SPR with no associated rename register terminates a group.
- No more than two instructions that modify the CR may be in a group.

XFER

The iops wait for resources to become free in the XFER stage.

GD, DSP, WRT, GCT, MAP

After group formation, the execution pipeline divides into multiple pipelines for the various execution units. Every cycle, one group of instructions can be sent (or dispatched) to the issue queues. Note that instructions in a group remain together from dispatch to completion.

38. The instruction grouping performed by the 970FX has similarities to a VLIW processor.

As a group is dispatched, several operations occur before the instructions actually execute. Internal group instruction dependencies are determined (GD). Various internal resources are assigned, such as issue queue slots, rename registers and mappers, and entries in the load/store reorder queues. In particular, each iop in the group that returns a result must be assigned a register to hold the result. Rename registers are allocated in the dispatch phase before the instructions enter the issue queues (DSP, MAP).

To track the groups themselves, the 970FX uses a *global completion table* (GCT) that stores up to 20 entries in program order—that is, up to 20 dispatch groups can be in flight concurrently. Since each group can have up to 5 iops, as many as 100 iops can be tracked in this manner. The WRT stage represents the writes to the GCT.

ISS, RF

After all the resources that are required to execute the instructions are available, the instructions are sent (ISS) to appropriate issue queues. Once their operands appear, the instructions start to execute. Each slot in a group feeds separate issue queues for various execution units. For example, the FXU/LSU and the FPU draw their instructions from slots { 0, 3 } and { 1, 2 }, respectively, of an instruction group. If one pair goes to the FXU/LSU, the other pair goes to the FPU. The CRU draws its instructions from the CR logical issue queue that is fed from instruction slots 0 and 1. As we saw earlier, slot 4 of an instruction group is dedicated to branch instructions. AltiVec instructions can be issued to the VALU and the VPERM issue queues from any slot except slot 4. Table 3–8 shows the 970FX issue queue sizes—each execution unit listed has one issue queue.

The FXU/LSU and FPU issue queues have odd and even halves that are hardwired to receive instructions only from certain slots of a dispatch group, as shown in Figure 3–15.

As long as an issue queue contains instructions that have all their data dependencies resolved, an instruction moves every cycle from the queue into the appropriate execution unit. However, there are likely to be instructions whose operands are not ready; such instructions block in the queue. Although the 970FX will attempt to execute the oldest instruction first, it will reorder instructions within a queue's context to avoid stalling. Ready-to-execute instructions access their source operands by reading the corresponding register file (RF), after which they enter the execution unit pipelines. Up to ten operations can be issued in a cycle—one to each of the ten execution pipelines. Note that different execution units may have varying numbers of pipeline stages.

TABLE 3–8 Sizes of the Various 970FX Issue Queues

Execution Unit	Queue Size (Instructions)
LSU0/FXU0[a]	18
LSU1/FXU1[b]	18
FPU0	10
FPU1	10
BRU	12
CRU	10
VALU	20
VPERM	16

a. LSU0 and FXU0 share an 18-entry issue queue.
b. LSU1 and FXU1 share an 18-entry issue queue.

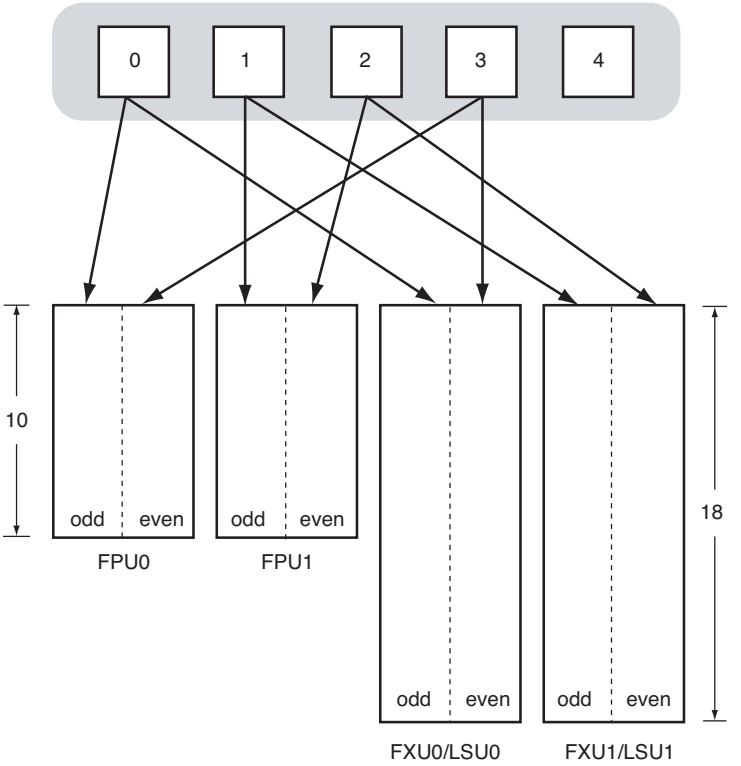

FIGURE 3–15 The FPU and FXU/LSU issue queues in the 970FX

We have seen that instructions both issue and execute out of order. However, if an instruction has finished execution, it does not mean that the program will "know" about it. After all, from the program's standpoint, instructions must execute in program order. The 970FX differentiates between an instruction finishing execution and an instruction *completing*. An instruction may finish execution (speculatively, say), but unless it completes, its effect is not visible to the program. All pipelines terminate in a common stage: the group completion stage (CP). When groups complete, many of their resources are released, such as load reorder queue entries, mappers, and global completion table entries. One dispatch group may be "retired" per cycle.

> When a branch instruction completes, the resultant target address is compared with a predicted address. Depending on whether the prediction is correct or incorrect, either all instructions in the pipeline that were fetched after the branch in question are flushed, or the processor waits for all remaining instructions in the branch's group to complete.

Accounting for 215 In-Flight Instructions

We can account for the theoretical maximum of 215 in-flight instructions by looking at Figure 3–4—specifically, the areas marked 1 through 6.

1. The Instruction Fetch Unit has a fetch/overflow buffer that can hold 16 instructions.
2. The instruction fetch buffer in the decode/dispatch unit can hold 32 instructions.
3. Every cycle, up to 5 instructions are taken from the instruction fetch buffer and sent through a three-stage instruction decode pipeline. Therefore, up to 15 instructions can be in this pipeline.
4. There are four dispatch buffers, each holding a dispatch group of up to five operations. Therefore, up to 20 instructions can be held in these buffers.
5. The global completion table can track up to 20 dispatch groups after they have been dispatched, corresponding to up to 100 instructions in the 970FX core.
6. The store queue can hold up to 32 stores.

Thus, the theoretical maximum number of in-flight instructions can be calculated as the sum 16 + 32 + 15 + 20 + 100 + 32, which is 215.

3.3.9.2 Branch Prediction

Branch prediction is a mechanism wherein the processor attempts to keep the pipeline full, and therefore improve overall performance, by fetching instructions in the hope that they will be executed. In this context, a branch is a decision point for the processor: It must predict the outcome of the branch—whether it will be *taken* or not—and accordingly prefetch instructions. As shown in Figure 3–15, the 970FX scans fetched instructions for branches. It looks for up to two branches per cycle and uses multistrategy branch prediction logic to predict their target addresses, directions, or both. Consequently, up to 2 branches are predicted per cycle, and up to 16 predicted branches can be in flight.

All conditional branches are predicted, based on whether the 970FX fetches instructions beyond a branch and speculatively executes them. Once the branch instruction itself executes in the BRU, its actual outcome is compared with its predicted outcome. If the prediction was incorrect, there is a severe penalty: Any instructions that may have speculatively executed are discarded, and instructions in the correct control-flow path are fetched.

The 970FX's dynamic branch prediction hardware includes three branch history tables (BHTs), a link stack, and a count cache. Each BHT has 16K 1-bit entries.

> The 970FX's hardware branch prediction can be overridden by software.

The first BHT is the *local predictor table*. Its 16K entries are indexed by branch instruction addresses. Each 1-bit entry indicates whether the branch should be taken or not. This scheme is "local" because each branch is tracked in isolation.

The second BHT is the *global predictor table*. It is used by a prediction scheme that takes into account the execution path taken to reach the branch. An 11-bit vector—the *global history vector*—represents the execution path. The bits of this vector represent the previous 11 instruction groups fetched. A particular bit is 1 if the next group was fetched sequentially and is 0 otherwise. A given branch's entry in the global predictor table is at a location calculated by performing an XOR operation between the global history vector and the branch instruction address.

The third BHT is the *selector table*. It tracks which of the two prediction schemes is to be favored for a given branch. The BHTs are kept up to date with the actual outcomes of executed branch instructions.

The link stack and the count cache are used by the 970FX to predict branch target addresses of *branch-conditional-to-link-register* (`bclr`, `bclrl`) and *branch-conditional-to-count-register* (`bcctr`, `bcctrl`) instructions, respectively.

So far, we have looked at *dynamic* branch prediction. The 970FX also supports static prediction wherein the programmer can use certain bits in a conditional branch operand to statically override dynamic prediction. Specifically, two bits called the "a" and "t" bits are used to provide hints regarding the branch's direction, as shown in Table 3–9.

TABLE 3–9 Static Branch Prediction Hints

"a" Bit	"t" Bit	Hint
0	0	Dynamic branch prediction is used.
0	1	Dynamic branch prediction is used.
1	0	Dynamic branch prediction is disabled; static prediction is "not taken"; specified by a "-" suffix to a branch conditional mnemonic.
1	1	Dynamic branch prediction is disabled; static prediction is "taken"; specified by a "+" suffix to a branch conditional mnemonic.

3.3.9.3 Summary

Let us summarize the instruction parallelism achieved by the 970FX. In every cycle of the 970FX, the following events occur.

- Up to eight instructions are fetched.
- Up to two branches are predicted.
- Up to five iops (one group) are dispatched.
- Up to five iops are renamed.
- Up to ten iops are issued from the issue queues.
- Up to five iops are completed.

3.3.10 AltiVec

The 970FX includes a dedicated vector-processing unit and implements the VMX instruction set, which is an AltiVec[39] interchangeable extension to the PowerPC architecture. AltiVec provides a SIMD-style 128-bit[40] vector-processing unit.

39. AltiVec was first introduced in Motorola's e600 PowerPC core—the G4.

40. All AltiVec execution units and data paths are 128 bits wide.

3.3.10.1 Vector Computing

SIMD stands for *single-instruction, multiple-data*. It refers to a set of operations that can efficiently handle large quantities of data in parallel. SIMD operations do not necessarily require more or wider registers, although more *is* better. SIMD essentially better uses registers and data paths. For example, a non-SIMD computation would typically use a hardware register for each data element, even if the register could hold multiple such elements. In contrast, SIMD would use a register to hold multiple data elements—as many as would fit—and would perform the *same operation on all elements* through a *single instruction*. Thus, any operation that can be parallelized in this manner stands to benefit from SIMD. In AltiVec's case, a vector instruction can perform the same operation on all constituents of a vector. Note that AltiVec instructions work on fixed-length vectors.

> SIMD-based optimization does not come for free. A problem must lend itself well to vectorization, and the programmer must usually perform extra work. Some compilers—such as IBM's XL suite of compilers and GCC 4.0 or above—also support auto-vectorization, an optimization that auto-generates vector instructions based on the compiler's analysis of the source code.[41] Auto-vectorization may or may not work well depending on the nature and structure of the code.

Several processor architectures have similar extensions. Table 3–10 lists some well-known examples.

AltiVec can greatly improve the performance of data movement, benefiting applications that do processing of vectors, matrices, arrays, signals, and so on. As we saw in Chapter 2, Apple provides portable APIs—through the Accelerate framework (`Accelerate.framework`)—for performing vector-optimized operations.[42] Accelerate is an umbrella framework that contains the vecLib and vImage[43] subframeworks. vecLib is targeted for performing numerical and scientific computing—it

41. For example, the compiler may attempt to detect patterns of code that are known to be well suited for vectorization.

42. The Accelerate framework automatically uses the best available code that it implements, depending on the hardware it is running on. For example, it will use vectorized code for AltiVec if AltiVec is available. On the x86 platform, it will use MMX, SSE, SSE2, and SSE3 if these features are available.

43. vImage is also available as a stand-alone framework.

TABLE 3–10 Examples of Processor Multimedia-Extensions

Processor Family	Manufacturers	Multimedia Extension Sets
Alpha	Hewlett-Packard (Digital Equipment Corporation)	MVI
AMD	Advanced Micro Devices (AMD)	3DNow!
MIPS	Silicon Graphics Incorporated (SGI)	MDMX, MIPS-3D
PA-RISC	Hewlett-Packard	MAX, MAX2
PowerPC	IBM, Motorola	VMX/AltiVec
SPARC V9	Sun Microsystems	VIS
x86	Intel, AMD, Cyrix	MMX, SSE, SSE2, SSE3

provides functionality such as BLAS, LAPACK, digital signal processing, dot products, linear algebra, and matrix operations. vImage provides vector-optimized APIs for working with image data. For example, it provides functions for alpha compositing, convolutions, format conversion, geometric transformations, histograms operations, and morphological operations.

> Although a vector instruction performs work that would typically require many times more nonvector instructions, vector instructions are not simply instructions that deal with "many scalars" or "more memory" at a time. The fact that a vector's members are related is critical, and so is the fact that the same operation is performed on all members. Vector operations certainly play better with memory accesses—they lead to *amortization*. The semantic difference between performing a vector operation and a sequence of scalar operations on the same data set is that you are implicitly providing more information to the processor about your intentions. Vector operations—by their nature—alleviate both data and control hazards.

AltiVec has wide-ranging applications since areas such as high-fidelity audio, video, videoconferencing, graphics, medical imaging, handwriting analysis, data encryption, speech recognition, image processing, and communications all use algorithms that can benefit from vector processing.

Figure 3–16 shows a trivial AltiVec C program.

As also shown in Figure 3–16, the `-faltivec` option to GCC enables AltiVec language extensions.

FIGURE 3–16 A trivial AltiVec program

```
// altivec.c

#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    // "vector" is an AltiVec keyword
    vector float v1, v2, v3;

    v1 = (vector float)(1.0, 2.0, 3.0, 4.0);
    v2 = (vector float)(2.0, 3.0, 4.0, 5.0);

    // vector_add() is a compiler built-in function
    v3 = vector_add(v1, v2);

    // "%vf" is a vector-formatting string for printf()
    printf("%vf\n", v3);

    exit(0);
}

$ gcc -Wall -faltivec -o altivec altivec.c
$ ./altivec
3.000000 5.000000 7.000000 9.000000
```

3.3.10.2 The 970FX AltiVec Implementation

The 970FX AltiVec implementation consists of the following components:

- A vector register file (VRF) consisting of 32 128-bit architected vector registers (VR0–VR31)
- 48 128-bit rename registers for allocation in the dispatch phase
- A 32-bit Vector Status and Control Register (VSCR)
- A 32-bit Vector Save/Restore Register (VRSAVE)
- A Vector Permute Unit (VPERM) that benefits the implementation of operations such as arbitrary byte-wise data organization, table lookups, and packing/unpacking of data
- A Vector Arithmetic and Logical Unit (VALU) that contains three parallel subunits: the Vector Simple-Integer Unit (VX), the Vector Complex-Integer Unit (VC), and the Vector Floating-Point Unit (VF)

The CR is also modified as a result of certain vector instructions.

The VALU and the VPERM are both dispatchable units that receive predecoded instructions via the issue queues.

The 32-bit VRSAVE serves a special purpose: Each of its bits indicates whether the corresponding vector register is in use or not. The processor maintains this register so that it does not have to save and restore every vector register every time there is an exception or a context switch. Frequently saving or restoring 32 128-bit registers, which together constitute 512 bytes, would be severely detrimental to cache performance, as other, perhaps more critical data would need to be evicted from cache.

Let us extend our example program from Figure 3–16 to examine the value in the VRSAVE. Figure 3–17 shows the extended program.

FIGURE 3–17 Displaying the contents of the VRSAVE

```
// vrsave.c

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

void prbits(u_int32_t);
u_int32_t read_vrsave(void);

// Print the bits of a 32-bit number
void
prbits32(u_int32_t u)
{
    u_int32_t i = 32;

    for (; i--; putchar(u & 1 << i ? '1' : '0'));

    printf("\n");
}

// Retrieve the contents of the VRSAVE
u_int32_t
read_vrsave(void)
{
    u_int32_t v;
```

(continues)

FIGURE 3–17 Displaying the contents of the VRSAVE *(continued)*

```
    __asm("mfspr %0,VRsave\n\t"
          : "=r"(v)
          :
    );

    return v;
}

int
main()
{
    vector float v1, v2, v3;

    v1 = (vector float)(1.0, 2.0, 3.0, 4.0);
    v2 = (vector float)(2.0, 3.0, 4.0, 5.0);

    v3 = vec_add(v1, v2);

    prbits32(read_vrsave());

    exit(0);
}


$ gcc -Wall -faltivec -o vrsave vrsave.c
$ ./vrsave
11000000000000000000000000000000
```

We see in Figure 3–17 that two high-order bits of the VRSAVE are set and the rest are cleared. This means the program uses two VRs: VR0 and VR1. You can verify this by looking at the assembly listing for the program.

The VPERM execution unit can do merge, permute, and splat operations on vectors. Having a separate permute unit allows data-reorganization instructions to proceed in parallel with vector arithmetic and logical instructions. The VPERM and VALU both maintain *their own copies of the VRF* that are synchronized on the half cycle. Thus, each receives its operands from its own VRF. Note that vector loads, stores, and data stream instructions are handled in the usual LSU pipes. Although no AltiVec instructions are cracked or microcoded, vector store instructions logically break down into two components: a vector part and an LSU part. In the group formation stage, a vector store is a single entity occupying one slot. However, once the instruction is issued, it occupies two issue queue slots: one in the vector store unit and another in the LSU. Address generation takes place in the LSU. There is a slot for moving the data out of the VRF in the vector unit.

This is not any different from scalar (integer and floating-point) stores, in whose case address generation still takes place in the LSU, and the respective execution unit—integer or floating-point—is used for accessing the GPR file (GPRF) or the FPR file (FPRF).

AltiVec instructions were designed to be pipelined easily. The 970FX can dispatch up to four vector instructions every cycle—regardless of type—to the issue queues. Any vector instruction can be dispatched from any slot of the dispatch group except the dedicated branch slot 4.

> It is usually very inefficient to pass data between the scalar units and the vector unit because data transfer between register files is not direct but goes through the caches.

3.3.10.3 AltiVec Instructions

AltiVec adds 162 vector instructions to the PowerPC architecture. Like all other PowerPC instructions, AltiVec instructions have 32-bit-wide encodings. To use AltiVec, no context switching is required. There is no special AltiVec operating mode—AltiVec instructions can be used along with regular PowerPC instructions in a program. AltiVec also does not interfere with floating-point registers.

> AltiVec instructions should be used at the UISA and VEA levels of the PowerPC architecture but not at the OEA level (the kernel). The same holds for floating-point arithmetic. Nevertheless, it is possible to use AltiVec and floating-point in the Mac OS X kernel beginning with a revision of Mac OS X 10.3. However, doing so would be at the cost of performance overhead in the kernel, since using AltiVec or floating-point will lead to a larger number of exceptions and register save/restore operations. Moreover, AltiVec data stream instructions cannot be used in the kernel. High-speed video scrolling on the system console is an example of the Floating-Point Unit being used by the kernel—the scrolling routines use floating-point registers for fast copying. The audio subsystem also uses floating-point in the kernel.

The following points are noteworthy regarding AltiVec vectors.

- A vector is 128 bits wide.
- A vector can be comprised of one of the following: 16 bytes, 8 half words, 4 words (integers), or 4 single-precision floating-point numbers.

- The largest vector element size is hardware-limited to 32 bits; the largest adder in the VALU is 32 bits wide. Moreover, the largest multiplier array is 24 bits wide, which is good enough for only a single-precision floating-point mantissa.[44]
- A given vector's members can be all unsigned or all signed quantities.
- The VALU behaves as multiple ALUs based on the vector element size.

Instructions in the AltiVec instruction set can be broadly classified into the following categories:

- Vector load and store instructions
- Instructions for reading from or writing to the VSCR
- Data stream manipulation instructions, such as *data-stream-touch* (`dst`), *data-stream-stop* (`dss`), and *data-stream-stop-all* (`dssall`)
- Vector fixed-point arithmetic and comparison instructions
- Vector logical, rotate, and shift instructions
- Vector pack, unpack, merge, splat, and permute instructions
- Vector floating-point instructions

> Vector single-element loads are implemented as `lvx`, with undefined fields not zeroed explicitly. Care should be taken while dealing with such cases as this could lead to denormals[45] in floating-point calculations.

3.3.11 Power Management

The 970FX supports power management features such as the following.

- It can dynamically stop the clocks of some of its constituents when they are idle.
- It can be programmatically put into predefined power-saving modes such as *doze*, *nap*, and *deep nap*.

---

44. The IEEE 754 standard defines the 32 bits of a single-precision floating-point number to consist of a sign (1 bit), an exponent (8 bits), and a mantissa (23 bits).

45. Denormal numbers—also called subnormal numbers—are numbers that are so small they cannot be represented with full precision.

- It includes *PowerTune*, a processor-level power management technology that supports scaling of processor and bus clock frequencies and voltage.

3.3.11.1 PowerTune

PowerTune allows clock frequencies to be dynamically controlled and even synchronized across multiple processors. PowerTune frequency scaling occurs in the processor core, the busses, the bridge, and the memory controller. Allowed frequencies range from *f*—the full nominal frequency—to *f/2*, *f/4*, and *f/64*. The latter corresponds to the deep nap power-saving mode. If an application does not require the processor's maximum available performance, frequency and voltage can be changed system-wide—without stopping the core execution units and without disabling interrupts or bus snooping. All processor logic, except the bus clocks, remains active. Moreover, the frequency change is very rapid. Since power has a quadratic dependency on voltage, reducing voltage has a desirable effect on power dissipation. Consequently, the 970FX has much lower typical power consumption than the 970, which did not have PowerTune.

3.3.11.2 Power Mac G5 Thermal and Power Management

In the Power Mac G5, Apple combines the power management capabilities of the 970FX/970MP with a network of fans and sensors to contain heat generation, power consumption, and noise levels. Examples of hardware sensors include those for fan speed, temperature, current, and voltage. The system is divided into discrete cooling zones with independently controlled fans. Some Power Mac G5 models additionally contain a liquid cooling system that circulates a thermally conductive fluid to transfer heat away from the processors into a radiant grille. As air passes over the grille's cooling fins, the fluid's temperature decreases.[46]

The Liquid in Liquid Cooling

The heat transfer fluid used in the liquid cooling system consists of mostly water mixed with antifreeze. A deionized form of water called DI water is used. The low concentration of ions in such water prevents mineral deposits and electric arcing, which may occur because the circulating coolant can cause static charge to build up.

46. Similar to how an automobile radiator works.

Operating system support is required to make the Power Mac G5's thermal management work properly. Mac OS X regularly monitors various temperatures and power consumption. It also communicates with the fan control unit (FCU). If the FCU does not receive feedback from the operating system, it will spin the fans at maximum speed.

A liquid-cooled dual-processor 2.5GHz Power Mac has the following fans:

- CPU A PUMP
- CPU A INTAKE
- CPU A EXHAUST
- CPU B PUMP
- CPU B INTAKE
- CPU B EXHAUST
- BACKSIDE
- DRIVE BAY
- SLOT

Additionally, the Power Mac has sensors for current, voltage, and temperature, as listed in Table 3–11.

TABLE 3–11 Power Mac G5 Sensors: An Example

Sensor Type	Sensor Location/Name
Ammeter	CPU A AD7417[a] AD2
Ammeter	CPU A AD7417 AD4
Ammeter	CPU B AD7417 AD2
Ammeter	CPU B AD7417 AD4
Switch	Power Button
Thermometer	BACKSIDE
Thermometer	U3 HEATSINK
Thermometer	DRIVE BAY
Thermometer	CPU A AD7417 AMB

(continues)

TABLE 3–11 Power Mac G5 Sensors: An Example *(Continued)*

Sensor Type	Sensor Location/Name
Thermometer	CPU A AD7417 AD1
Thermometer	CPU B AD7417 AMB
Thermometer	CPU B AD7417 AD1
Thermometer	MLB INLET AMB
Voltmeter	CPU A AD7417 AD3
Voltmeter	CPU B AD7417 AD3

a. The AD7417 is a type of analog-to-digital converter with an on-chip temperature sensor.

We will see in Chapter 10 how to programmatically retrieve the values of various sensors from the kernel.

3.3.12 64-bit Architecture

We saw earlier that the PowerPC architecture was designed with explicit support for 64- and 32-bit computing. PowerPC is, in fact, a 64-bit architecture with a 32-bit subset. A particular PowerPC implementation may choose to implement only the 32-bit subset, as is the case with the G3 and G4 processor families used by Apple. The 970FX implements both the 64-bit and 32-bit forms[47]—*dynamic computation modes*[48]—of the PowerPC architecture. The modes are dynamic in that you can switch between the two dynamically by setting or clearing bit 0 of the MSR.

3.3.12.1 64-bit Features

The key aspects of the 970FX's 64-bit mode are as follows:

- 64-bit registers:[49] the GPRs, CTR, LR, and XER
- 64-bit addressing, including 64-bit pointers, which allow one program's address space to be larger than 4GB
- 32-bit and 64-bit programs, which can execute side by side

47. A 64-bit PowerPC implementation must implement the 32-bit subset.

48. The computation mode encompasses addressing mode.

49. Several registers are defined to be 32-bit in the 64-bit PowerPC architecture, such as CR, FPSCR, VRSAVE, and VSCR.

- 64-bit integer and logical operations, with fewer instructions required to load and store 64-bit quantities[50]
- Fixed instruction size—32 bits—in both 32- and 64-bit modes
- 64-bit-only instructions such as *load-word-algebraic* (`lwa`), *load-word-algebraic-indexed* (`lwax`), and "double-word" versions of several instructions

Although a Mac OS X process must be 64-bit itself to be able to directly access more than 4GB of virtual memory, having support in the processor for more than 4GB of physical memory benefits both 64-bit and 32-bit applications. After all, physical memory *backs* virtual memory. Recall that the 970FX can track a large amount of physical memory—42 bits worth, or 4TB. Therefore, as long as there is enough RAM, much greater amounts of it can be kept "alive" than is possible with only 32 bits of physical addressing. This is beneficial to 32-bit applications because the operating system can now keep more working sets in RAM, reducing the number of page-outs—even though a single 32-bit application will still "see" only a 4GB address space.

3.3.12.2 The 970FX as a 32-bit Processor

Just as the 64-bit PowerPC is not an extension of the 32-bit PowerPC, the latter is not a performance-limited version of the former—there is no penalty for executing in 32-bit-only mode on the 970FX. There are, however, some differences. Important aspects of running the 970FX in 32-bit mode include the following.

- The sizes of the floating-point and AltiVec registers are the same across 32-bit and 64-bit implementations. For example, an FPR is 64 bits wide and a VR is 128 bits wide on both the G4 and the G5.
- The 970FX uses the same resources—registers, execution units, data paths, caches, and busses—in 64- and 32-bit modes.
- Fixed-point logical, rotate, and shift instructions behave the same in both modes.
- Fixed-point arithmetic instructions (except the negate instruction) actually produce the same result in 64- and 32-bit modes. However, the carry (CA)

50. One way to use 64-bit integers on a 32-bit processor is to have the programming language maintain 64-bit integers as two 32-bit integers. Doing so would consume more registers and would require more load/store instructions.

and overflow (OV) fields of the XER register are set in a 32-bit-compatible way in 32-bit mode.

- Load/store instructions ignore the upper 32 bits of an effective address in 32-bit mode. Similarly, branch instructions deal with only the lower 32 bits of an effective address in 32-bit mode.

3.3.13 Softpatch Facility

The 970FX provides a facility called *softpatch*, which is a mechanism that allows software to work around bugs in the processor core and to otherwise debug the core. This is achieved either by replacing an instruction with a substitute microcoded instruction sequence or by making an instruction cause a trap to software through a softpatch exception.

The 970FX's Instruction Fetch Unit contains a seven-entry array with content-addressable memory (CAM). This array is called the Instruction Match CAM (IMC). Additionally, the 970FX's instruction decode unit contains a microcode *softpatch table*. The IMC array has eight rows. The first six IMC entries occupy one row each, whereas the seventh entry occupies two rows. Of the seven entries, the first six are used to match partially (17 bits) over an instruction's major opcode (bits 0 through 5) and extended opcode (bits 21 through 31). The seventh entry matches in its entirety: a 32-bit full instruction match. As instructions are fetched from storage, they are matched against the IMC entries by the Instruction Fetch Unit's matching facility. If matched, the instruction's processing can be altered based on other information in the matched entry. For example, the instruction can be replaced with microcode from the instruction decode unit's softpatch table.

The 970FX provides various other tracing and performance-monitoring facilities that are beyond the scope of this chapter.

3.4 Software Conventions

An application binary interface (ABI) defines a system interface for compiled programs, allowing compilers, linkers, debuggers, executables, libraries, other object files, and the operating system to work with each other. In a simplistic sense, an ABI is a low-level, "binary" API. A program conforming to an API *should* be compilable from source on different systems supporting that API,

whereas a binary executable conforming to an ABI *should* operate on different systems supporting that ABI.[51]

An ABI usually includes a set of rules specifying how hardware and software resources are to be used for a given architecture. Besides interoperability, the conventions laid down by an ABI may have performance-related goals too, such as minimizing average subroutine-call overhead, branch latencies, and memory accesses. The scope of an ABI could be extensive, covering a wide variety of areas such as the following:

- Byte ordering (endianness)
- Alignment and padding
- Register usage
- Stack usage
- Subroutine parameter passing and value returning
- Subroutine prologues and epilogues
- System calls
- Object files
- Dynamic code generation
- Program loading and dynamic linking

The PowerPC version of Mac OS X uses the Darwin PowerPC ABI in its 32-bit and 64-bit versions, whereas the 32-bit x86 version uses the System V IA-32 ABI. The Darwin PowerPC ABI is similar to—but not the same as—the popular IBM AIX ABI for the PowerPC. In this section, we look at some aspects of the Darwin PowerPC ABI without analyzing its differences from the AIX ABI.

3.4.1 Byte Ordering

The PowerPC architecture natively supports 8-bit (byte), 16-bit (half word), 32-bit (word), and 64-bit (double word) data types. It uses a flat-address-space model with byte-addressable storage. Although the PowerPC architecture provides an optional little-endian facility, the 970FX does not implement it—it implements only the big-endian addressing mode. Big-endian refers to storing the "big" end of a multibyte value at the lowest memory address. In the PowerPC

51. ABIs vary in whether they strictly enforce cross-operating-system compatibility or not.

architecture, the leftmost bit—bit 0—is defined to be the *most significant bit*, whereas the rightmost bit is the *least significant bit*. For example, if a 64-bit register is being used as a 32-bit register in 32-bit computation mode, then bits 32 through 63 of the 64-bit register represent the 32-bit register; bits 0 through 31 are to be ignored. By corollary, the leftmost byte—byte 0—is the most significant byte, and so on.

> In PowerPC implementations that support both the big-endian and little-endian[52] addressing modes, the LE bit of the Machine State Register can be set to 1 to specify little-endian mode. Another bit—the ILE bit—is used to specify the mode for exception handlers. The default value of both bits is 0 (big-endian) on such processors.

3.4.2 Register Usage

The Darwin ABI defines a register to be dedicated, volatile, or nonvolatile. A *dedicated* register has a predefined or standard purpose; it should not be arbitrarily modified by the compiler. A *volatile* register is available for use at all times, but its contents may change if the context changes—for example, because of calling a subroutine. Since the caller must save volatile registers in such cases, such registers are also called *caller-save* registers. A *nonvolatile* register is available for use in a local context, but the user of such registers must save their original contents before use and must restore the contents before returning to the calling context. Therefore, it is the *callee*—and not the caller—who must save nonvolatile registers. Correspondingly, such registers are also called *callee-save* registers.

> In some cases, a register may be available for general use in one runtime environment but may have a special purpose in some other runtime environment. For example, GPR12 has a predefined purpose on Mac OS X when used for indirect function calls.

Table 3–12 lists common PowerPC registers along with their usage conventions as defined by the 32-bit Darwin ABI.

52. The use of little-endian mode on such processors is subject to several caveats as compared to big-endian mode. For example, certain instructions—such as load/store multiple and load/store string—are not supported in little-endian mode.

TABLE 3–12 Register Conventions in the 32-bit Darwin PowerPC ABI

Register(s)	Volatility	Purpose/Comments
GPR0	Volatile	Cannot be a base register.
GPR1	Dedicated	Used as the stack pointer to allow access to parameters and other temporary data.
GPR2	Volatile	Available on Darwin as a local register but used as the Table of Contents (TOC) pointer in the AIX ABI. Darwin does not use the TOC.
GPR3	Volatile	Contains the first argument word when calling a subroutine; contains the first word of a subroutine's return value. Objective-C uses GPR3 to pass a pointer to the object being messaged (i.e., "self") as an implicit parameter.
GPR4	Volatile	Contains the second argument word when calling a subroutine; contains the second word of a subroutine's return value. Objective-C uses GPR4 to pass the method selector as an implicit parameter.
GPR5–GPR10	Volatile	GPR*n* contains the (*n* – 2)th argument word when calling a subroutine.
GPR11	Varies	In the case of a nested function, used by the caller to pass its stack frame to the nested function—register is nonvolatile. In the case of a leaf function, the register is available and is volatile.
GPR12	Volatile	Used in an optimization for dynamic code generation, wherein a routine that branches indirectly to another routine must store the target of the call in GPR12. No special purpose for a routine that has been called directly.
GPR13–GPR29	Nonvolatile	Available for general use. Note that GPR13 is reserved for thread-specific storage in the 64-bit Darwin PowerPC ABI.
GPR30	Nonvolatile	Used as the frame pointer register—i.e., as the base register for access to a subroutine's local variables.
GPR31	Nonvolatile	Used as the PIC-offset table register.
FPR0	Volatile	Scratch register.
FPR1–FPR4	Volatile	FPR*n* contains the *n*th floating-point argument when calling a subroutine; FPR1 contains the subroutine's single-precision floating-point return value; a double-precision floating-point value is returned in FPR1 and FPR2.
FPR5–FPR13	Volatile	FPR*n* contains the *n*th floating-point argument when calling a subroutine.
FPR14–FPR31	Nonvolatile	Available for general use.
CR0	Volatile	Used for holding condition codes during arithmetic operations.
CR1	Volatile	Used for holding condition codes during floating-point operations.
CR2–CR4	Nonvolatile	Various condition codes.

(continues)

TABLE 3–12 Register Conventions in the 32-bit Darwin PowerPC ABI *(Continued)*

Register(s)	Volatility	Purpose/Comments
CR5	Volatile	Various condition codes.
CR6	Volatile	Various condition codes; can be used by AltiVec.
CR7	Volatile	Various condition codes.
CTR	Volatile	Contains a branch target address (for the `bcctr` instruction); contains counter value for a loop.
FPSCR	Volatile	Floating-Point Status and Control Register.
LR	Volatile	Contains a branch target address (for the `bclr` instruction); contains subroutine return address.
XER	Volatile	Fixed-point exception register.
VR0, VR1	Volatile	Scratch registers.
VR2	Volatile	Contains the first vector argument when calling a subroutine; contains the vector returned by a subroutine.
VR3–VR19	Volatile	VR*n* contains the (*n* – 1)th vector argument when calling a subroutine.
VR20–VR31	Nonvolatile	Available for general use.
VRSAVE	Nonvolatile	If bit *n* of the VRSAVE is set, then VR*n* must be saved during any kind of a context switch.
VSCR	Volatile	Vector Status and Control Register.

3.4.2.1 Indirect Calls

We noted in Table 3–12 that a function that branches indirectly to another function stores the target of the call in GPR12. Indirect calls are, in fact, the default scenario for dynamically compiled Mac OS X user-level code. Since the target address would need to be stored in a register in any case, using a standardized register allows for potential optimizations. Consider the code fragment shown in Figure 3–18.

FIGURE 3–18 A simple C function that calls another function

```
void
f1(void)
{
    f2();
}
```

By default, the assembly code generated by GCC on Mac OS X for the function shown in Figure 3–18 will be similar to that shown in Figure 3–19, which has been annotated and trimmed down to relevant parts. In particular, note the use of GPR12, which is referred to as `r12` in the GNU assembler syntax.

FIGURE 3–19 Assembly code depicting an indirect function call

```
...
_f1:
        mflr r0          ; prologue
        stmw r30,-8(r1)  ; prologue
        stw r0,8(r1)     ; prologue
        stwu r1,-80(r1)  ; prologue
        mr r30,r1        ; prologue
        bl L_f2$stub     ; indirect call
        lwz r1,0(r1)     ; epilogue
        lwz r0,8(r1)     ; epilogue
        mtlr r0          ; epilogue
        lmw r30,-8(r1)   ; epilogue
        blr              ; epilogue
...
L_f2$stub:
        .indirect_symbol _f2
        mflr r0
        bcl 20,31,L0$_f2
L0$_f2:
        mflr r11

        ; lazy pointer contains our desired branch target
        ; copy that value to r12 (the 'addis' and the 'lwzu')
        addis r11,r11,ha16(L_f2$lazy_ptr-L0$_f2)
        mtlr r0
        lwzu r12,lo16(L_f2$lazy_ptr-L0$_f2)(r11)

        ; copy branch target to CTR
        mtctr r12

        ; branch through CTR
        bctr
.data
.lazy_symbol_pointer
L_f2$lazy_ptr:
        .indirect_symbol _f2
        .long dyld_stub_binding_helper
```

3.4.2.2 Direct Calls

If GCC is instructed to statically compile the code in Figure 3–18, we can verify in the resultant assembly that there is a direct call to `f2` from `f1`, with no use of GPR12. This case is shown in Figure 3–20.

FIGURE 3–20 Assembly code depicting a direct function call

```
        .machine ppc
        .text
        .align 2
        .globl _f1
_f1:
        mflr r0
        stmw r30,-8(r1)
        stw r0,8(r1)
        stwu r1,-80(r1)
        mr r30,r1
        bl _f2
        lwz r1,0(r1)
        lwz r0,8(r1)
        mtlr r0
        lmw r30,-8(r1)
        blr
```

3.4.3 Stack Usage

On most processor architectures, a stack is used to hold automatic variables, temporary variables, and return information for each invocation of a subroutine. The PowerPC architecture does not explicitly define a stack for local storage: There is neither a dedicated stack pointer nor any push or pop instructions. However, it is conventional for operating systems running on the PowerPC—including Mac OS X—to designate (per the ABI) an area of memory as the stack and grow it *upward*: from a high memory address to a low memory address. GPR1, which is used as the stack pointer, points to the top of the stack.

Both the stack and the registers play important roles in the working of subroutines. As listed in Table 3–12, registers are used to hold subroutine arguments, up to a certain number.

Functional Subtleties

The terms *function*, *procedure*, and *subroutine* are sometimes used in programming language literature to denote similar but slightly differing entities. For example, a function is a procedure that always returns a result, but a "pure" procedure does not return a result. *Subroutine* is often used as a general term for either a function or a procedure. The C language does not make such fine distinctions, but some languages do. We use these terms synonymously to represent the fundamental programmer-visible unit of callable execution in a high-level language like C.

Similarly, the terms *argument* and *parameter* are used synonymously in informal contexts. In general, when you declare a function that "takes arguments," you use *formal parameters* in its declaration. These are placeholders for *actual parameters*, which are what you specify when you call the function. Actual parameters are often called *arguments*.

The mechanism whereby actual parameters are matched with (or *bound* to) formal parameters is called *parameter passing*, which could be performed in various ways, such as *call-by-value* (actual parameter represents its value), *call-by-reference* (actual parameter represents its location), *call-by-name* (actual parameter represents its program text), and variants.

If a function `f1` calls another function `f2`, which calls yet another function `f3`, and so on in a program, the program's stack grows per the ABI's conventions. Each function in the call chain owns part of the stack. A representative runtime stack for the 32-bit Darwin ABI is shown in Figure 3–21.

In Figure 3–21, `f1` calls `f2`, which calls `f3`. `f1`'s stack frame contains a *parameter area* and a *linkage area*.

The parameter area must be large enough to hold the largest parameter list of all functions that `f1` calls. `f1` typically will pass arguments in registers as long as there are registers available. Once registers are exhausted, `f1` will place arguments in its parameter area, from where `f2` will pick them up. However, `f1` must reserve space for all arguments of `f2` in any case—even if it is able to pass all arguments in registers. `f2` is free to use `f1`'s parameter area for storing arguments if it wants to free up the corresponding registers for other use. Thus, in a subroutine call, the caller sets up a parameter area in its own stack portion, and the callee can access the caller's parameter area for loading or storing arguments.

The linkage area begins after the parameter area and is at the top of the stack—adjacent to the stack pointer. The adjacency to the stack pointer is important: The linkage area has a fixed size, and therefore the callee can find the

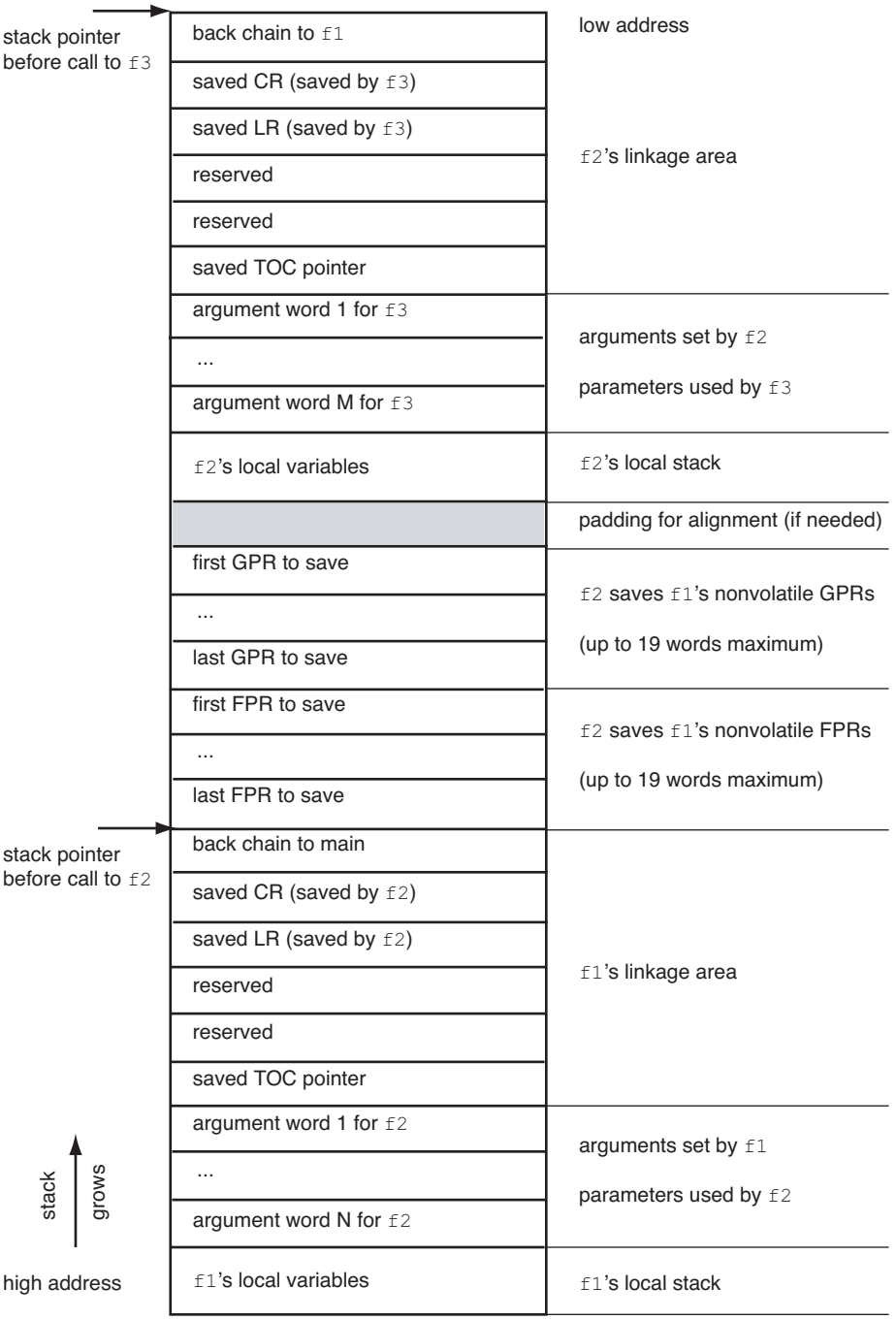

FIGURE 3–21 Darwin 32-bit ABI runtime stack

caller's parameter area deterministically. The callee can save the CR and the LR in the caller's linkage area if it needs to. The stack pointer is always saved by the caller as a back chain to its caller.

In Figure 3–21, `f2`'s portion of the stack shows space for saving nonvolatile registers that `f2` changes. These must be restored by `f2` before it returns to its caller.

Space for each function's local variables is reserved by growing the stack appropriately. This space lies below the parameter area and above the saved registers.

The fact that a called function is responsible for allocating its own stack frame does not mean the programmer has to write code to do so. When you compile a function, the compiler inserts code fragments called the *prologue* and the *epilogue* before and after the function body, respectively. The prologue sets up the stack frame for the function. The epilogue undoes the prologue's work, restoring any saved registers (including CR and LR), incrementing the stack pointer to its previous value (that the prologue saved in its linkage area), and finally returning to the caller.

A 32-bit Darwin ABI stack frame is 16-byte aligned.

Consider the trivial function shown in Figure 3–22, along with the corresponding annotated assembly code.

FIGURE 3–22 Assembly listing for a C function with no arguments and an empty body

```
$ cat function.c
void
function(void)
{
}
$ gcc -S function.c
$ cat function.s
...
_function:
      stmw r30,-8(r1) ; Prologue: save r30 and r31
      stwu r1,-48(r1) ; Prologue: grow the stack 48 bytes
      mr r30,r1       ; Prologue: copy stack pointer to r30
      lwz r1,0(r1)    ; Epilogue: pop the stack (restore frame)
      lmw r30,-8(r1)  ; Epilogue: restore r30 and r31
      blr             ; Epilogue: return to caller (through LR)
```

The Red Zone

Just after a function is called, the function's prologue will decrement the stack pointer from its existing location to reserve space for the function's needs. The area above the stack pointer, where the newly called function's stack frame would reside, is called the *Red Zone*.

In the 32-bit Darwin ABI, the Red Zone has space for 19 GPRs (amounts to $19 \times 4 = 76$ bytes) and 18 FPRs (amounts to $18 \times 8 = 144$ bytes), for a total of 220 bytes. Rounded up to the nearest 16-byte boundary, this becomes 224 bytes, which is the size of the Red Zone.

Normally, the Red Zone is indeed occupied by the callee's stack frame. However, if the callee does not call any other function—that is, it is a *leaf function*—then it does not need a parameter area. It may also not need space for local variables on the stack if it can fit all of them in registers. It may need space for saving the nonvolatile registers it uses (recall that if a callee needs to save the CR and LR, it can save them in the caller's linkage area). As long as it can fit the registers to save in the Red Zone, it does not need to allocate a stack frame or decrement the stack pointer. Note that by definition, there is only one leaf function active at one time.

3.4.3.1 Stack Usage Examples

Figures 3–23 and 3–24 show examples of how the compiler sets up a function's stack depending on the number of local variables a function has, the number of parameters it has, the number of arguments it passes to a function it calls, and so on.

`f1` is identical to the "null" function that we encountered in Figure 3–22, where we saw that the compiler reserves 48 bytes for the function's stack. The portions shown as shaded in the stacks are present either for alignment padding or for some current or future purpose not necessarily exposed through the ABI. Note that GPR30 and GPR31 are always saved, GPR30 being the designated frame pointer.

`f2` uses a single 32-bit local variable. Its stack is 64 bytes.

`f3` calls a function that takes no arguments. Nevertheless, this introduces a parameter area on `f3`'s stack. A parameter area is at least eight words (32 bytes) in size. `f3`'s stack is 80 bytes.

`f4` takes eight arguments, has no local variables, and calls no functions. Its stack area is the same size as that of the null function because space for its arguments is reserved in the parameter area of its caller.

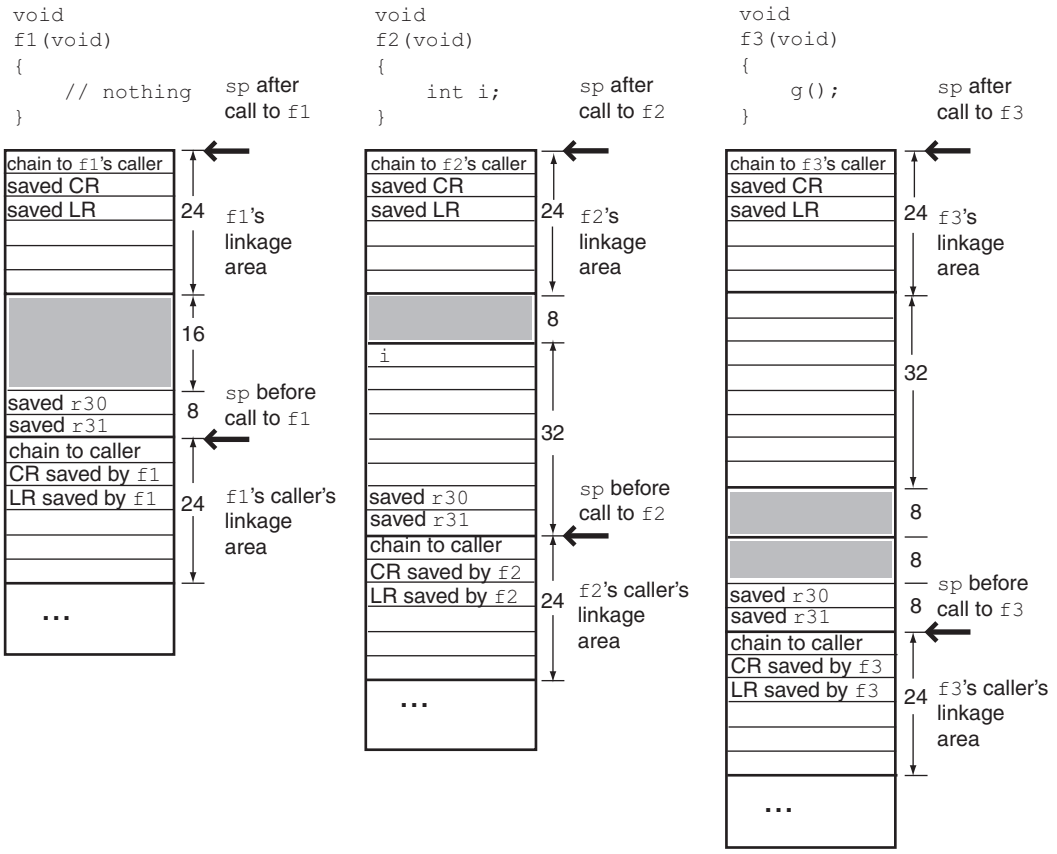

FIGURE 3–23 Examples of stack usage in functions

`f5` takes no arguments, has eight word-size local variables, and calls no functions. Its stack is 64 bytes.

3.4.3.2 Printing Stack Frames

GCC provides built-in functions that may be used by a function to retrieve information about its callers. The current function's return address can be retrieved by calling the `__builtin_return_address()` function, which takes a single argument—the *level*, an integer specifying the number of stack frames to walk. A level of 0 results in the return address of the current function. Similarly, the `__builtin_frame_address()` function may be used to retrieve the frame

```
void
f4(int a1, int a2, int a3, int a4
  int a5, int a6, int a7, int a8)
{
    /* no local variables */
    /* no functions called */
}
```

```
void
f5(void)
{
    int l1, l2, l3, l4,
        l5, l6, l7, l8;
    /* no functions called */
}
```

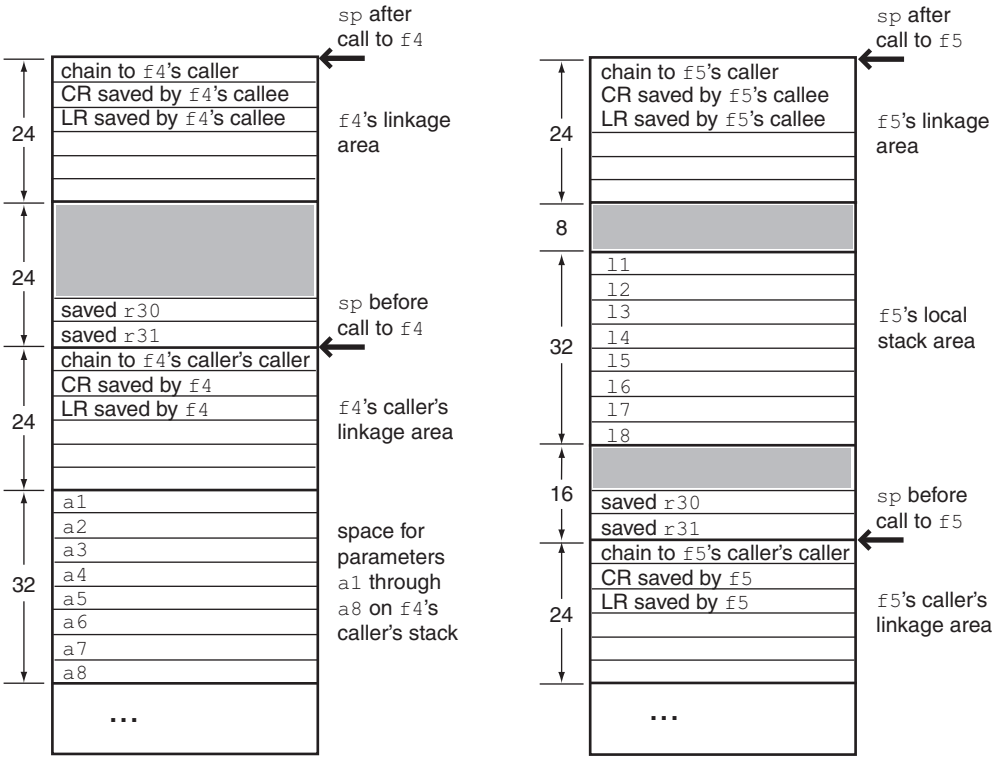

FIGURE 3–24 Examples of stack usage in functions (continued from Figure 3–23)

address of a function in the call stack. Both functions return a `NULL` pointer when the top of the stack has been reached.[53] Figure 3–25 shows a program that uses these functions to display a stack trace. The program also uses the `dladdr()` function in the `dyld` API to find the various function addresses corresponding to return addresses in the call stack.

53. For `__builtin_frame_address()` to return a `NULL` pointer upon reaching the top of the stack, the first frame pointer must have been set up correctly.

FIGURE 3–25 Printing a function call stack trace[54]

```
// stacktrace.c

#include <stdio.h>
#include <dlfcn.h>

void
printframeinfo(unsigned int level, void *fp, void *ra)
{
    int     ret;
    Dl_info info;

    // Find the image containing the given address
    ret = dladdr(ra, &info);
    printf("#%u %s%s in %s, fp = %p, pc = %p\n",
           level,
           (ret) ? info.dli_sname : "?",          // symbol name
           (ret) ? "()" : "",                     // show as a function
           (ret) ? info.dli_fname : "?", fp, ra); // shared object name
}

void
stacktrace()
{
    unsigned int level = 0;
    void    *saved_ra  = __builtin_return_address(0);
    void   **fp        = (void **)__builtin_frame_address(0);
    void    *saved_fp  = __builtin_frame_address(1);

    printframeinfo(level, saved_fp, saved_ra);
    level++;
    fp = saved_fp;
    while (fp) {
        saved_fp = *fp;
        fp = saved_fp;
        if (*fp == NULL)
            break;
        saved_ra = *(fp + 2);
        printframeinfo(level, saved_fp, saved_ra);
        level++;
    }
}
```

(continues)

54. Note in the program's output that the function name in frames #5 and #6 is `tart`. The `dladdr()` function strips leading underscores from the symbols it returns—even if there is no leading underscore (in which case it removes the first character). In this case, the symbol's name is `start`.

FIGURE 3–25 Printing a function call stack trace *(continued)*

```
void f4() { stacktrace(); }
void f3() { f4(); }
void f2() { f3(); }
void f1() { f2(); }

int
main()
{
    f1();
    return 0;
}
```

```
$ gcc -Wall -o stacktrace stacktrace.c
$ ./stacktrace
#0 f4() in /private/tmp/./stacktrace, fp = 0xbffff850, pc = 0x2a3c
#1 f3() in /private/tmp/./stacktrace, fp = 0xbffff8a0, pc = 0x2a68
#2 f2() in /private/tmp/./stacktrace, fp = 0xbffff8f0, pc = 0x2a94
#3 f1() in /private/tmp/./stacktrace, fp = 0xbffff940, pc = 0x2ac0
#4 main() in /private/tmp/./stacktrace, fp = 0xbffff990, pc = 0x2aec
#5 tart() in /private/tmp/./stacktrace, fp = 0xbffff9e0, pc = 0x20c8
#6 tart() in /private/tmp/./stacktrace, fp = 0xbffffa40, pc = 0x1f6c
```

3.4.4 Function Parameters and Return Values

We saw earlier that when a function calls another with arguments, the parameter area in the caller's stack frame is large enough to hold all parameters passed to the called function, regardless of the number of parameters actually passed in registers. Doing so has benefits such as the following.

- The called function might want to call further functions that take arguments or might want to use registers containing its arguments for other purposes. Having a dedicated parameter area allows the callee to store an argument from a register to the argument's "home location" on the stack, thus freeing up a register.
- It may be useful to have all arguments in the parameter area for debugging purposes.
- If a function has a variable-length parameter list, it will typically access its arguments from memory.

3.4.4.1 Passing Parameters

Parameter-passing rules may depend on the type of programming language used—for example, procedural or object-oriented. Let us look at parameter-passing rules for C and C-like languages. Even for such languages, the rules further depend on whether a function has a fixed-length or a variable-length parameter list. The rules for fixed-length parameter lists are as follows.

- The first eight parameter words (i.e., the first 32 bytes, not necessarily the first eight arguments) are passed in GPR3 through GPR10, unless a floating-point parameter appears.
- Floating-point parameters are passed in FPR1 through FPR13.
- If a floating-point parameter appears, but GPRs are still available, then the parameter is placed in an FPR, as expected. However, the next available GPRs that together sum up to the floating-point parameter's size are skipped and not considered for allocation. Therefore, a single-precision floating-point parameter (4 bytes) causes the next available GPR (4 bytes) to be skipped. A double-precision floating-point parameter (8 bytes) causes the next two available GPRs (8 bytes total) to be skipped.
- If not all parameters can fit within the available registers in accordance with the skipping rules, the caller passes the excess parameters by storing them in the parameter area of its stack frame.
- Vector parameters are passed in VR2 through VR13.
- Unlike floating-point parameters, vector parameters do not cause GPRs—or FPRs, for that matter—to be skipped.
- Unless there are more vector parameters than can fit in available vector registers, no space is allocated for vector parameters in the caller's stack frame. Only when the registers are exhausted does the caller reserve any vector parameter space.

Let us look at the case of functions with variable-length parameter lists. Note that a function may have some number of required parameters preceding a variable number of parameters.

- Parameters in the variable portion of the parameter list are passed in both GPRs and FPRs. Consequently, floating-point parameters are always *shadowed* in GPRs instead of causing GPRs to be skipped.
- If there are vector parameters in the fixed portion of the parameter list, 16-byte-aligned space is reserved for such parameters in the caller's parameter area, even if there are available vector registers.

- If there are vector parameters in the variable portion of the parameter list, such parameters are also shadowed in GPRs.
- The called routine accesses arguments from the fixed portion of the parameter list similarly to the fixed-length parameter list case.
- The called routine accesses arguments from the variable portion of the parameter list by copying GPRs to the callee's parameter area and accessing values from there.

3.4.4.2 Returning Values

Functions return values according to the following rules.

- Values less than one word (32 bits) in size are returned in the least significant byte(s) of GPR3, with the remaining byte(s) being undefined.
- Values exactly one word in size are returned in GPR3.
- 64-bit fixed-point values are returned in GPR3 (the 4 low-order bytes) and GPR4 (the 4 high-order bytes).
- Structures up to a word in size are returned in GPR3.
- Single-precision floating-point values are returned in FPR1.
- Double-precision floating-point values are returned in FPR1.
- A 16-byte long double value is returned in FPR1 (the 8 low-order bytes) and FPR2 (the 8 high-order bytes).
- A composite value (such as an array, a structure, or a union) that is more than one word in size is returned via an implicit pointer that the caller must pass. Such functions require the caller to pass a pointer to a memory location that is large enough to hold the return value. The pointer is passed as an "invisible" argument in GPR3. Actual user-visible arguments, if any, are passed in GPR4 onward.

3.5 Examples

Let us now look at several miscellaneous examples to put some of the concepts we have learned into practice. We will discuss the following specific examples:

- Assembly code corresponding to a recursive factorial function
- Implementation of an atomic compare-and-store function

- Rerouting function calls
- Using a cycle-accurate 970FX simulator

3.5.1 A Recursive Factorial Function

In this example, we will understand how the assembly code corresponding to a simple, high-level C function works. The function is shown in Figure 3–26. It recursively computes the factorial of its integer argument.

FIGURE 3–26 A recursive function to compute factorials

```
// factorial.c

int
factorial(int n)
{
    if (n > 0)
        return n * factorial(n - 1);
    else
        return 1;
}

$ gcc -Wall -S factorial.c
```

The GCC command line shown in Figure 3–26 generates an assembly file named `factorial.s`. Figure 3–27 shows an annotated version of the contents of this file.

Noting Annotations

Whereas the listing in Figure 3–27 is hand-annotated, GCC can produce certain types of annotated output that may be useful in some debugging scenarios. For example, the `-dA` option annotates the assembler output with some minimal debugging information; the `-dp` option annotates each assembly mnemonic with a comment indicating which pattern and alternative were used; the `-dP` option intersperses assembly-language lines with transcripts of the register transfer language (RTL); and so on.

FIGURE 3–27 Annotated assembly listing for the function shown in Figure 3–26

```
; factorial.s

.section __TEXT,__text
    .globl _factorial
_factorial:

    ; LR contains the return address, copy LR to r0.
    mflr r0

    ; Store multiple words (the registers r30 and r31) to the address starting
    ; at [-8 + r1]. An stmw instruction is of the form "stmw rS,d(rA)" -- it
    ; stores n consecutive words starting at the effective address (rA|0)+d.
    ; The words come from the low-order 32 bits of GPRs rS through r31. In
    ; this case, rS is r30, so two words are stored.
    stmw r30,-8(r1)

    ; Save LR in the "saved LR" word of the linkage area of our caller.
    stw r0,8(r1)

    ; Grow the stack by 96 bytes:
    ;
    ; * 24 bytes for our linkage area
    ; * 32 bytes for 8 words' worth of arguments to functions we will call
    ;   (we actually use only one word)
    ; * 8 bytes of padding
    ; * 16 bytes for local variables (we actually use only one word)
    ; * 16 bytes for saving GPRs (such as r30 and r31)
    ;
    ; An stwu instruction is of the form "stwu rS, d(rA)" -- it stores the
    ; contents of the low-order 32 bits of rS into the memory word addressed
    ; by (rA)+d. The latter (the effective address) is also placed into rA.
    ; In this case, the contents of r1 are stored at (r1)-96, and the address
    ; (r1)-96 is placed into r1. In other words, the old stack pointer is
    ; stored and r1 gets the new stack pointer.
    stwu r1,-96(r1)

    ; Copy current stack pointer to r30, which will be our frame pointer --
    ; that is, the base register for accessing local variables, etc.
    mr r30,r1

    ; r3 contains our first parameter
    ;
    ; Our caller contains space for the corresponding argument just below its
    ; linkage area, 24 bytes away from the original stack pointer (before we
    ; grew the stack): 96 + 24 = 120
    ; store the parameter word in the caller's space.
    stw r3,120(r30)
```

(continues)

FIGURE 3–27 Annotated assembly listing for the function shown in Figure 3–26 *(continued)*

```
    ; Now access n, the first parameter, from the caller's parameter area.
    ; Copy n into r0.
    ; We could also use "mr" to copy from r3 to r0.
    lwz r0,120(r30)

    ; Compare n with 0, placing result in cr7 (corresponds to the C line).
    ; "if (n > 0)")
    cmpwi cr7,r0,0

    ; n is less than or equal to 0: we are done. Branch to factorial0.
    ble cr7,factorial0

    ; Copy n to r2 (this is Darwin, so r2 is available).
    lwz r2,120(r30)

    ; Decrement n by 1, and place the result in r0.
    addi r0,r2,-1

    ; Copy r0 (that is, n - 1) to r3.
    ; r3 is the first argument to the function that we will call: ourselves.
    mr r3,r0

    ; Recurse.
    bl _factorial

    ; r3 contains the return value.
    ; Copy r3 to r2
    mr r2,r3

    ; Retrieve n (the original value, before we decremented it by 1), placing
    ; it in r0.
    lwz r0,120(r30)

    ; Multiply n and the return value (factorial(n - 1)), placing the result
    ; in r0.
    mullw r0,r2,r0

    ; Store the result in a temporary variable on the stack.
    stw r0,64(r30)

    ; We are all done: get out of here.
    b done

factorial0:
    ; We need to return 1 for factorial(n), if n <= 0.
    li r0,1
```

(continues)

FIGURE 3–27 Annotated assembly listing for the function shown in Figure 3–26 *(continued)*

```
    ; Store the return value in a temporary variable on the stack.
    stw r0,64(r30)

done:
    ; Load the return value from its temporary location into r3.
    lwz r3,64(r30)

    ; Restore the frame ("pop" the stack) by placing the first word in the
    ; linkage area into r1.
    ;
    ; The first word is the back chain to our caller.
    lwz r1,0(r1)

    ; Retrieve the LR value we placed in the caller's linkage area and place
    ; it in r0.
    lwz r0,8(r1)

    ; Load LR with the value in r0.
    mtlr r0

    ; Load multiple words from the address starting at [-8 + r1] into r30
    ; and r31.
    lmw r30,-8(r1)

    ; Go back to the caller.
        blr
```

3.5.2 An Atomic Compare-and-Store Function

We came across the *load-and-reserve-conditional* (`lwarx`, `ldarx`) and *store-conditional* (`stwcx.`, `stdcx.`) instructions earlier in this chapter. These instructions can be used to enforce storage ordering of I/O accesses. For example, we can use `lwarx` and `stcwx.` to implement an atomic compare-and-store function. Executing `lwarx` loads a word from a word-aligned location but also performs the following two actions *atomically* with the load.

- It creates a reservation that can be used by a subsequent `stwcx.` instruction. Note that a processor cannot have more than one reservation at a time.
- It notifies the processor's storage coherence mechanism that there is now a reservation for the specified memory location.

`stwcx.` stores a word to the specified word-aligned location. Its behavior depends on whether the location is the same as the one specified to `lwarx` to create a reservation. If the two locations are the same, `stwcx.` will perform the store *only if* there has been no other store to that location since the reservation's creation—one or more other stores, if any, could be by another processor, cache operations, or through any other mechanism. If the location specified to `stwcx.` is different from the one used with `lwarx`, the store may or may not succeed, but the reservation will be lost. A reservation may be lost in various other scenarios, and `stwcx.` will fail in all such cases. Figure 3–28 shows an implementation of a compare-and-store function. The Mac OS X kernel includes a similar function. We will use this function in our next example to implement function rerouting.

FIGURE 3–28 A hardware-based compare-and-store function for the 970FX

```
// hw_cs.s
//
// hw_compare_and_store(u_int32_t old,
//                      u_int32_t new,
//                      u_int32_t *address,
//                      u_int32_t *dummyaddress)
//
// Performs the following atomically:
//
// Compares old value to the one at address, and if they are equal, stores new
// value, returning true (1). On store failure, returns false (0). dummyaddress
// points to a valid, trashable u_int32_t location, which is written to for
// canceling the reservation in case of a failure.

        .align  5
        .globl  _hw_compare_and_store

_hw_compare_and_store:
        // Arguments:
        //      r3      old
        //      r4      new
        //      r5      address
        //      r6      dummyaddress

        // Save the old value to a free register.
        mr      r7,r3

looptry:
        // Retrieve current value at address.
        // A reservation will also be created.
        lwarx   r9,0,r5
```

(continues)

FIGURE 3–28 A hardware-based compare-and-store function for the 970FX *(continued)*

```
        // Set return value to true, hoping we will succeed.
        li      r3,1

        // Do old value and current value at address match?
        cmplw   cr0,r9,r7

        // No! Somebody changed the value at address.
        bne--   fail

        // Try to store the new value at address.
        stwcx.  r4,0,r5

        // Failed! Reservation was lost for some reason.
        // Try again.
        bne--   looptry

        // If we use hw_compare_and_store to patch/instrument code dynamically,
        // without stopping already running code, the first instruction in the
        // newly created code must be isync. isync will prevent the execution
        // of instructions following itself until all preceding instructions
        // have completed, discarding prefetched instructions. Thus, execution
        // will be consistent with the newly created code. An instruction cache
        // miss will occur when fetching our instruction, resulting in fetching
        // of the modified instruction from storage.
        isync

        // return
        blr

fail:
        // We want to execute a stwcx. that specifies a dummy writable aligned
        // location. This will "clean up" (kill) the outstanding reservation.
        mr      r3,r6
        stwcx.  r3,0,r3

        // set return value to false.
        li      r3,0

        // return
        blr
```

3.5.3 Function Rerouting

Our goal in this example is to intercept a function in a C program by substituting a new function in its place, with the ability to call the original function from the new function. Let us assume that there is a function `function(int, char *)`,

which we wish to replace with `function_new(int, char *)`. The replacement must meet the following requirements.

- After replacement, when `function()` is called from anywhere within the program, `function_new()` is called instead.
- `function_new()` can use `function()`, perhaps because `function_new()` is meant to be a wrapper for the original function.
- The rerouting can be programmatically installed or removed.
- `function_new()` is a normal C function, with the only requirement being that it has the exact same prototype as `function()`.

3.5.3.1 Instruction Patching

Assume that `function()`'s implementation is the instruction sequence i_0, i_1, ..., i_M, whereas `function_new()`'s implementation is the instruction sequence j_0, j_1, ..., j_N, where M and N are some integers. A caller of `function()` executes i_0 first because it is the first instruction of `function()`. If our goal is to arrange for all invocations of `function()` to actually call `function_new()`, we could overwrite i_0 in memory with an unconditional branch instruction to j_0, the first instruction of `function_new()`. Doing so would leave `function()` out of the picture entirely. Since we also wish to call `function()` from within `function_new()`, we cannot clobber `function()`. Moreover, we also wish to be able to turn off the rerouting and restore `function()` as it originally was.

Rather than clobber i_0, we save it somewhere in memory. Then, we allocate a memory region large enough to hold a few instructions and mark it executable. A convenient way to preallocate such a region is to declare a dummy function: one that takes the exact same number and type of arguments as `function()`. The dummy function will simply act as a stub; let us call it `function_stub()`. We copy i_0 to the beginning of `function_stub()`. We craft an instruction—an unconditional jump to i_1—that we write as the second instruction of `function_stub()`.

We see that we need to craft two branch instructions: one from `function()` to `function_new()`, and another from `function_stub()` to `function()`.

3.5.3.2 Constructing Branch Instructions

PowerPC unconditional branch instructions are self-contained in that they encode their target addresses within the instruction word itself. Recall from the previous

example that it is possible to update a word—a single instruction—atomically on the 970FX using a compare-and-store (also called compare-and-*update*) function. It would be more complicated in general to overwrite multiple instructions. Therefore, we will use unconditional branches to implement rerouting. The overall concept is shown in Figure 3–29.

The encoding of an unconditional branch instruction on the PowerPC is shown in Figure 3–30. It has a 24-bit address field (LI). Since all instructions are 4 bytes long, the PowerPC refers to words instead of bytes when it comes to branch target addresses. Since a word is 4 bytes, the 24 bits of LI are as good as 26 bits for our purposes. Given a 26-bit-wide effective branch address, the branch's maximum *reachability* is 64MB total,[55] or 32MB in either direction.

The Reach of a Branch

The "reachability" of a branch is processor-specific. A jump on MIPS uses 6 bits for the operand field and 26 bits for the address field. The effective addressable jump distance is actually 28 bits—four times more—because MIPS, like PowerPC, refers to the number of words instead of the number of bytes. All instructions in MIPS are 4 bytes long; 28 bits give you 256MB (±128MB) of total leeway. SPARC uses a 22-bit signed integer for branch addresses, but again, it has two zero bits appended, effectively providing a 24-bit program counter relative jump reachability. This amounts to reachability of 16MB (±8MB).

The AA field specifies whether the specified branch target address is absolute or relative to the current instruction (AA = 0 for relative, AA = 1 for absolute). If LK is 1, the effective address of the instruction following the branch instruction is placed in the LR. We do not wish to clobber the LR, so we are left with relative and absolute branches. We know now that to use a relative branch, the branch target must be within 32MB of the current instruction, but more importantly, we need to retrieve the address of the current instruction. Since the PowerPC does not have a program counter[56] register, we choose to use an unconditional branch with AA = 1 and LK = 0. However, this means the absolute address must be ±32MB relative to *zero*. In other words, `function_new` and

55. 2^{26} bytes.

56. The conceptual Instruction Address Register (IAR) is not directly accessible without involving the LR.

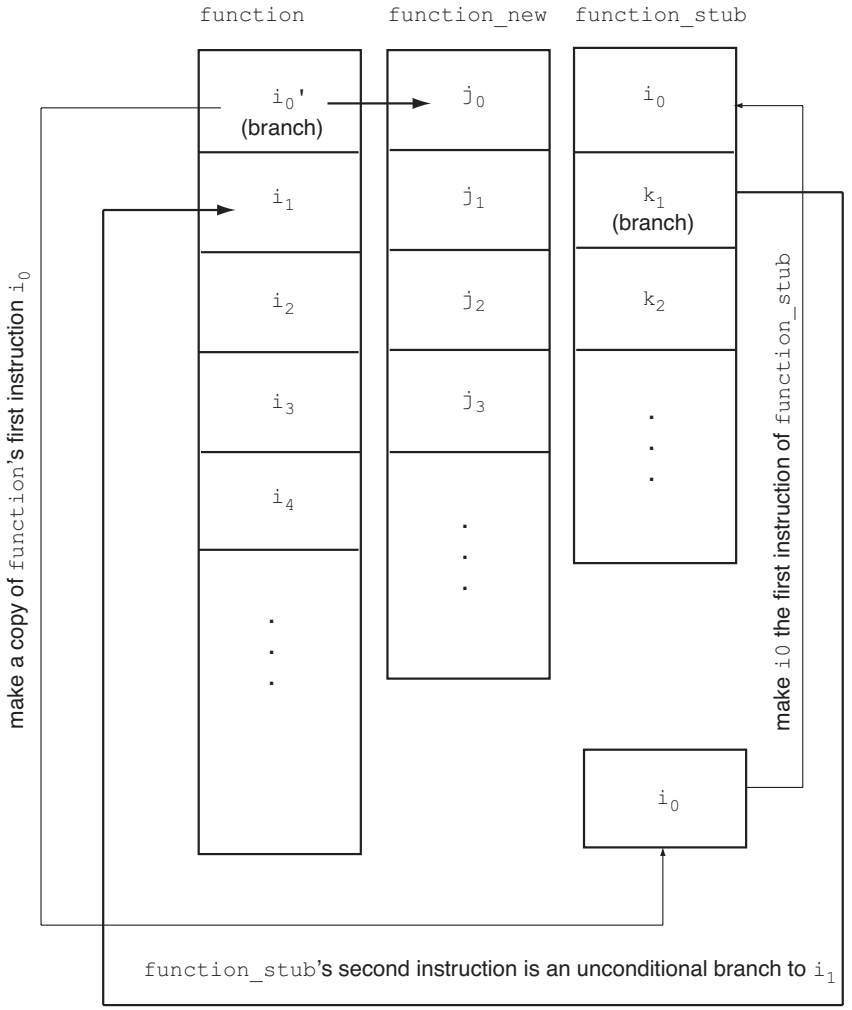

FIGURE 3–29 Overview of function rerouting by instruction patching

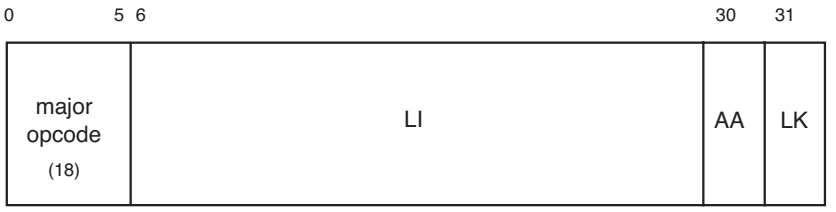

FIGURE 3–30 Unconditional branch instruction on the PowerPC

`function_stub` must reside in virtual memory *within the first 32MB or the last 32MB of the process's virtual address space!* In a simple program such as ours, this condition is actually likely to be satisfied due to the way Mac OS X sets up process address spaces. Thus, in our example, we simply "hope" for `function_new()` (our own declared function) and `function_stub()` (a buffer allocated through the `malloc()` function) to have virtual addresses that are less than 32MB. This makes our "technique" eminently unsuitable for production use. However, there is almost certainly free memory available in the first or last 32MB of any process's address space. As we will see in Chapter 8, Mach allows you to allocate memory at specified virtual addresses, so the technique can also be improved.

Figure 3–31 shows the code for the function-rerouting demo program. Note that the program is 32-bit only—it will behave incorrectly when compiled for the 64-bit architecture.

FIGURE 3–31 Implementation of function rerouting by instruction patching

```
// frr.c

#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>

// Constant on the PowerPC
#define BYTES_PER_INSTRUCTION 4

// Branch instruction's major opcode
#define BRANCH_MOPCODE 0x12

// Large enough size for a function stub
#define DEFAULT_STUBSZ 128

// Atomic update function
//
int hw_compare_and_store(u_int32_t  old,
                         u_int32_t  new,
                         u_int32_t *address,
                         u_int32_t *dummy_address);
```

(continues)

FIGURE 3–31 Implementation of function rerouting by instruction patching *(continued)*

```
// Structure corresponding to a branch instruction
//
typedef struct branch_s {
    u_int32_t OP: 6;  // bits 0 - 5, primary opcode
    u_int32_t LI: 24; // bits 6 - 29, LI
    u_int32_t AA: 1;  // bit 30, absolute address
    u_int32_t LK: 1;  // bit 31, link or not
} branch_t;

// Each instance of rerouting has the following data structure associated with
// it. A pointer to a frr_data_t is returned by the "install" function. The
// "remove" function takes the same pointer as argument.
//
typedef struct frr_data_s {
    void *f_orig; // "original" function
    void *f_new;  // user-provided "new" function
    void *f_stub; // stub to call "original" inside "new"
    char  f_bytes[BYTES_PER_INSTRUCTION]; // bytes from f_orig
} frr_data_t;

// Given an "original" function and a "new" function, frr_install() reroutes
// so that anybody calling "original" will actually be calling "new". Inside
// "new", it is possible to call "original" through a stub.
//
frr_data_t *
frr_install(void *original, void *new)
{
    int         ret = -1;
    branch_t    branch;
    frr_data_t *FRR = (frr_data_t *)0;
    u_int32_t   target_address, dummy_address;

    // Check new's address
    if ((u_int32_t)new >> 25) {
        fprintf(stderr, "This demo is out of luck. \"new\" too far.\n");
        goto ERROR;
    } else
        printf("    FRR: \"new\" is at address %#x.\n", (u_int32_t)new);

    // Allocate space for FRR metadata
    FRR = (frr_data_t *)malloc(sizeof(frr_data_t));
    if (!FRR)
        return FRR;

    FRR->f_orig = original;
    FRR->f_new = new;
```

(continues)

FIGURE 3–31 Implementation of function rerouting by instruction patching *(continued)*

```
// Allocate space for the stub to call the original function
FRR->f_stub = (char *)malloc(DEFAULT_STUBSZ);
if (!FRR->f_stub) {
    free(FRR);
    FRR = (frr_data_t *)0;
    return FRR;
}

// Prepare to write to the first 4 bytes of "original"
ret = mprotect(FRR->f_orig, 4, PROT_READ|PROT_WRITE|PROT_EXEC);
if (ret != 0)
    goto ERROR;

// Prepare to populate the stub and make it executable
ret = mprotect(FRR->f_stub, DEFAULT_STUBSZ, PROT_READ|PROT_WRITE|PROT_EXEC);
if (ret != 0)
    goto ERROR;

memcpy(FRR->f_bytes, (char *)FRR->f_orig, BYTES_PER_INSTRUCTION);

// Unconditional branch (relative)
branch.OP = BRANCH_MOPCODE;
branch.AA = 1;
branch.LK = 0;

// Create unconditional branch from "stub" to "original"
target_address = (u_int32_t)(FRR->f_orig + 4) >> 2;
if (target_address >> 25) {
    fprintf(stderr, "This demo is out of luck. Target address too far.\n");
    goto ERROR;
} else
    printf("    FRR: target_address for stub -> original is %#x.\n",
           target_address);
branch.LI = target_address;
memcpy((char *)FRR->f_stub, (char *)FRR->f_bytes, BYTES_PER_INSTRUCTION);
memcpy((char *)FRR->f_stub + BYTES_PER_INSTRUCTION, (char *)&branch, 4);

// Create unconditional branch from "original" to "new"
target_address = (u_int32_t)FRR->f_new >> 2;
if (target_address >> 25) {
    fprintf(stderr, "This demo is out of luck. Target address too far.\n");
    goto ERROR;
} else
    printf("    FRR: target_address for original -> new is %#x.\n",
           target_address);
branch.LI = target_address;
```

(continues)

FIGURE 3–31 Implementation of function rerouting by instruction patching *(continued)*

```
    ret = hw_compare_and_store(*((u_int32_t *)FRR->f_orig),
                               *((u_int32_t *)&branch),
                               (u_int32_t *)FRR->f_orig,
                               &dummy_address);
    if (ret != 1) {
        fprintf(stderr, "Atomic store failed.\n");
        goto ERROR;
    } else
        printf("    FRR: Atomically updated instruction.\n");

    return FRR;

    ERROR:
    if (FRR && FRR->f_stub)
        free(FRR->f_stub);
    if (FRR)
        free(FRR);
    return FRR;
}

int
frr_remove(frr_data_t *FRR)
{
    int       ret;
    u_int32_t dummy_address;

    if (!FRR)
        return 0;

    ret = mprotect(FRR->f_orig, 4, PROT_READ|PROT_WRITE|PROT_EXEC);
    if (ret != 0)
        return -1;

    ret = hw_compare_and_store(*((u_int32_t *)FRR->f_orig),
                               *((u_int32_t *)FRR->f_bytes),
                               (u_int32_t *)FRR->f_orig,
                               &dummy_address);

    if (FRR && FRR->f_stub)
        free(FRR->f_stub);

    if (FRR)
        free(FRR);

    FRR = (frr_data_t *)0;

    return 0;
}
```

(continues)

FIGURE 3–31 Implementation of function rerouting by instruction patching *(continued)*

```
int
function(int i, char *s)
{
    int   ret;
    char *m = s;

    if (!s)
        m = "(null)";

    printf(" CALLED: function(%d, %s).\n", i, m);
    ret = i + 1;
    printf(" RETURN: %d = function(%d, %s).\n", ret, i, m);

    return ret;
}

int (* function_stub)(int, char *);

int
function_new(int i, char *s)
{
    int   ret = -1;
    char *m = s;

    if (!s)
        m = "(null)";

    printf(" CALLED: function_new(%d, %s).\n", i, m);

    if (function_stub) {
        printf("CALLING: function_new() --> function_stub().\n");
        ret = function_stub(i, s);
    } else {
        printf("function_new(): function_stub missing.\n");
    }

    printf(" RETURN: %d = function_new(%d, %s).\n", ret, i, m);

    return ret;
}

int
main(int argc, char **argv)
{
    int         ret;
    int         arg_i = 2;
    char       *arg_s = "Hello, World!";
    frr_data_t *FRR;
```

(continues)

FIGURE 3–31 Implementation of function rerouting by instruction patching *(continued)*

```
    function_stub = (int(*)(int, char *))0;

    printf("[Clean State]\n");
        printf("CALLING: main() --> function().\n");
    ret = function(arg_i, arg_s);

    printf("\n[Installing Rerouting]\n");
    printf("Maximum branch target address is %#x (32MB).\n", (1 << 25));
    FRR = frr_install(function, function_new);
    if (FRR)
        function_stub = FRR->f_stub;
    else {
        fprintf(stderr, "main(): frr_install failed.\n");
        return 1;
    }

    printf("\n[Rerouting installed]\n");
    printf("CALLING: main() --> function().\n");
    ret = function(arg_i, arg_s);

    ret = frr_remove(FRR);
    if (ret != 0) {
        fprintf(stderr, "main(): frr_remove failed.\n");
        return 1;
    }

    printf("\n[Rerouting removed]\n");
    printf("CALLING: main() --> function().\n");
    ret = function(arg_i, arg_s);

    return 0;
}
```

Figure 3–32 shows a sample run of the function-rerouting demonstration program.

FIGURE 3–32 Function rerouting in action

```
$ gcc -Wall -o frr frr.c
$ ./frr
[Clean State]
CALLING: main() --> function().
CALLED: function(2, Hello, World!).
RETURN: 3 = function(2, Hello, World!).
```

(continues)

FIGURE 3–32 Function rerouting in action *(continued)*

```
[Installing Rerouting]
Maximum branch target address is 0x2000000 (32MB).
FRR: "new" is at address 0x272c.
FRR: target_address for stub -> original is 0x9a6.
FRR: target_address for original -> new is 0x9cb.
FRR: Atomically updated instruction.

[Rerouting installed]
CALLING: main() --> function().
CALLED: function_new(2, Hello, World!).
CALLING: function_new() --> function_stub().
CALLED: function(2, Hello, World!).
RETURN: 3 = function(2, Hello, World!).
RETURN: 3 = function_new(2, Hello, World!).

[Rerouting removed]
CALLING: main() --> function().
CALLED: function(2, Hello, World!).
RETURN: 3 = function(2, Hello, World!).
```

3.5.4 Cycle-Accurate Simulation of the 970FX

Apple's CHUD Tools package includes the `amber` and `simg5` command-line programs that were briefly mentioned in Chapter 2. `amber` is a tool for tracing all threads in a process, recording every instruction and data access to a trace file. `simg5`[57] is a cycle-accurate core simulator for the 970/970FX. With these tools, it is possible to analyze the execution of a program at the processor-cycle level. You can see how instructions are broken down into iops, how the iops are grouped, how the groups are dispatched, and so on. In this example, we will use `amber` and `simg5` to analyze a simple program.

The first step is to use `amber` to generate a trace of a program's execution. `amber` supports a few trace file formats. We will use the TT6E format with `simg5`.

Tracing the execution of an entire application—even a trivial program—will result in the execution of an extremely large number of instructions. Execution of the "empty" C program in Figure 3–33 causes over 90,000 instructions to be traced. This is so because although the program does not have any programmer-provided code (besides the empty function body), it still contains the runtime environment's startup and teardown routines.

57. `simg5` was developed by IBM.

FIGURE 3–33 Tracing an "empty" C program using `amber`

```
$ cat null.c
main()
{
}
$ gcc -o null null.c
$ amber ./null
...
Session Instructions Traced:  91353
Session Trace Time:           0.46 sec [0.20 million inst/sec]
...
```

Typically, you would not be interested in analyzing the execution of the language runtime environment. In fact, even within your own code, you may want to analyze only small portions at a time. It becomes impractical to deal with a large number—say, more than a few thousand—of instructions using these tools. When used with the `-i` or `-I` arguments, `amber` can toggle tracing for an application upon encountering a privileged instruction. A readily usable example of such an instruction is one that accesses an OEA register from user space. Thus, you can instrument your code by surrounding the portion of interest with two such illegal instructions. The first occurrence will cause `amber` to turn tracing on, and the second will cause tracing to stop. Figure 3–34 shows the program we will trace with `amber`.

FIGURE 3–34 A C program with instructions that are illegal in user space

```
// traceme.c

#include <stdlib.h>

#if defined(__GNUC__)
#include <ppc_intrinsics.h>
#endif

int
main(void)
{
    int i, a = 0;

    // supervisor-level instruction as a trigger
    // start tracing
    (void)__mfspr(1023);
```

(continues)

FIGURE 3–34 A C program with instructions that are illegal in user space *(continued)*

```
    for (i = 0; i < 16; i++) {
        a += 3 * i;
    }

    // supervisor-level instruction as a trigger
    // stop tracing
    (void)__mfspr(1023);

   exit(0);
}
```

We trace the program in Figure 3–34 using `amber` with the `-I` option, which directs `amber` to trace only the instrumented thread. The `-i` option would cause all threads in the target process to be traced. As shown in Figure 3–35, the executable will not run stand-alone because of the presence of illegal instructions in the machine code.

FIGURE 3–35 Tracing program execution with `amber`

```
$ gcc -S traceme.c # GCC 4.x
$ gcc -o traceme traceme.c
$ ./traceme
zsh: illegal hardware instruction  ./traceme
$ amber -I ./traceme
...
* Targeting process 'traceme' [1570]
* Recording TT6E trace
* Instrumented executable - tracing will start/stop for thread automatically
* Ctrl-Esc to quit

* Tracing session #1 started *

Session Instructions Traced: 214
Session Traced Time:          0.00 sec [0.09 million inst/sec]

* Tracing session #1 stopped *

* Exiting... *
```

`amber` creates a subdirectory called `trace_xxx` in the current directory, where `xxx` is a three-digit numerical string: `001` if this is the first trace in the directory. The `trace_xxx` directory contains further subdirectories, one per

thread in your program, containing TT6E traces of the program's threads. A trace provides information such as what instructions were issued, what order they were issued in, what were the load and store addresses, and so on. Our program has only one thread, so the subdirectory is called `thread_001.tt6e`. As shown in Figure 3–35, `amber` reports that 214 instructions were traced. Let us account for these instructions by examining the generated assembly `traceme.s`, whose partial contents (annotated) are shown in Figure 3–36. Note that we are interested only in the portion between the pair of `mfspr` instructions. However, it is noteworthy that the instruction immediately following the first `mfspr` instruction is not included in amber's trace.

FIGURE 3–36 Accounting for instructions traced by `amber`

```
; traceme.s (compiled with GCC 4.x)
        mfspr r0, 1023              →
        stw   r0,60(r30)    ; not traced
        ; instructions of interest begin here
        li    r0,0
        stw   r0,68(r30)
        b     L2
L3:
        lwz   r2,68(r30)    ; i[n]
        mr    r0,r2         ; i[n + 1]
        slwi  r0,r0,1       ; i[n + 2]
        add   r2,r0,r2      ; i[n + 3]
        lwz   r0,64(r30)    ; i[n + 4]
        add   r0,r0,r2      ; i[n + 5]
        stw   r0,64(r30)    ; i[n + 6]
        lwz   r2,68(r30)    ; i[n + 7]
        addi  r0,r2,1       ; i[n + 8]
        stw   r0,68(r30)    ; i[n + 9]
L2:
        lwz   r0,68(r30)    ; i[n + 10]
        cmpwi cr7,r0,15     ; i[n + 11]
        ble   cr7,L3        ; i[n + 12]
        mfspr r0, 1023            →
```

Each of the 3 instructions before the `L3` loop label are executed only once, whereas the rest of the instructions that lie between the `L3` loop label and the second `mfspr` instruction are executed during one or all iterations of the loop. Instructions `i[n]` through `i[n + 9]` (10 instructions) are executed exactly 16 times, as the C variable `i` is incremented. The assembly implementation of the loop begins by jumping to the `L2` label and checks whether `i` has attained the

value 16, in which case the loop terminates. Since `i` is initially zero, instructions `i[n + 10]` through `i[n + 12]` (3 instructions) will be executed exactly 17 times. Thus, the total number of instructions executed can be calculated as follows:

$$3 + (10 \times 16) + (3 \times 17) = 214$$

Let us now run `simg5` on this trace. `simg5` allows you to change certain characteristics of the simulated processor, for example, by making the L1 I-cache, the L1 D-cache, the TLBs, or the L2 cache infinite. There also exists a Java viewer for viewing `simg5`'s output. `simg5` can automatically run the viewer upon finishing if the `auto_load` option is specified.

```
$ simg5 trace_001/thread_001.tt6e 214 1 1 test_run1 -p 1 -b 1 -e 214 -auto_load
...
```

Figure 3–37 shows `simg5`'s output as displayed by the Java viewer. The left side of the screen contains a cycle-by-cycle sequence of processor events. These are denoted by labels, examples of which are shown in Table 3–13.

TABLE 3–13 Processor Event Labels

Label	Event	Notes
FVB	Fetch	Instruction fetched into the instruction buffer
D	Decode	Decode group formed
M	Dispatch	Dispatch group dispatched
su	Issue	Instruction issued
E	Execute	Instruction executing
f	Finish	Instruction finished execution
C	Completion	Instruction group completed

In the `simg5` output, we can see the breaking down of architected instructions into iops. For example, the second instruction of interest in Figure 3–36 has two corresponding iops.

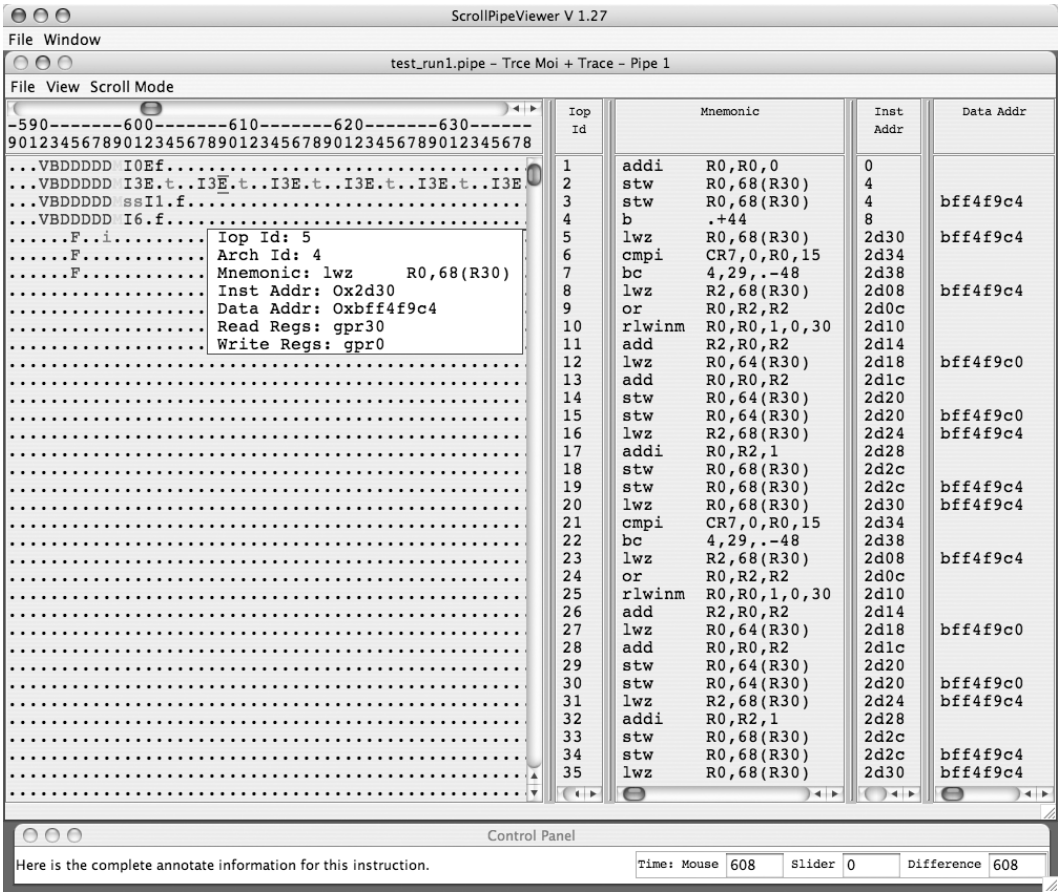

FIGURE 3–37 `simg5` output