Chapter 2
A Microarchitecture Case Study

We pointed out in Chapter 1 that the first half of this book presents a description of the microarchitecture of the AMD K6 3D microprocessor. In attempting to balance between giving you enough detail and too much detail, we will give a layered description of the microarchitecture. In this chapter we discuss the K6 3D's superscalar design and its multiple execution units, instruction buffers, predecode logic, multiple decoders, scheduler, branch resolution logic, operation commit unit, on-chip L1-Cache and L2-Cache, and other aspects of its microarchitecture. We follow this overview with detailed discussions of three of its main elements—its out-of-order, speculative scheduler, its operation commit unit, and its register renaming scheme in Chapter 3.

This chapter is written for all audiences (university professors and students, practitioners, and technical management). However, there are some subsections that have details that will be of more interest and use to practitioners and those in universities. The *road map* for this chapter identifies these sections.

ROAD MAP OF CHAPTER 2

Section	Audience
All major headings in the chapter	All
The following more detailed subsections: Register Number and Name Mappings Special Registers and Model Specific Registers Formats for Decoder Ops LdOps and StOps perform memory accesses and related operations. They have the following format in a decoder OpQuad: RegOp Field Descriptions SpecOp Field Descriptions LIMM Op Field Descriptions	Practitioners, University Professors and Students

An Overview of the K6 3D Microprocessor

In this section, we give an overview of how the K6 3D microprocessor works: how it fetches instructions and how they are predecoded and then decoded, how multiple internal operations result from this decoding process, how these operations go through a substantial expansion process before they are loaded into its centralized scheduler, how its pipelines are controlled and what type of work they do at each stage, how the decoding of instructions and the execution of the resulting operations are decoupled from one another, what types of caches the processor has on-chip and how they are organized, how operations are issued, and how and when predicted branches are ultimately resolved. The explanation of the particular design approaches taken within the K6 3D requires knowing a bit about the microarchitecture's internal operation set and the internal representation of operations within the scheduler, so some detail of both of these is also presented.

After completing this overview, you should be well positioned to understand the more detailed discussions of the scheduler, operation commit unit, and register renaming given in Chapter 3. Recall that our intent is to give you enough detail to allow you to simulate some important portions of the microarchitecture and associated platform and systems devices. You can gain a much greater understanding of how these chunks of the design actually work by doing such simulations. Such knowledge is basic to understanding the complex mix of cost and performance trade-off involved in taking the microarchitecture and producing a chip from it within a very aggressive time-to-market constraint.

A Range of Design Approaches

As stated in Chapter 1, one view of the K6 3D is that it is a high-performance CISC-on-RISC microprocessor. The CISC-component is the x86 instruction set architecture and the underlying RISC-component is known as the Enhanced RISC86 Microarchitecture. The most important technical implication of this constraint is that the K6 3D must be fully x86 binary code compatible, including the MMX multimedia extensions to the x86 instruction set architecture.

Suggested Readings

Complete Description of the x86 Instruction Set Architecture

For a complete description of the x86 instruction set architecture see the three-volume *The Intel Architecture Software Developer's Manual*: *Basic Architecture*, Intel Order Number 243190; *Instruction Set Reference,* Order Number 243191; and the *System Programming Guide,* Order Number 243192. These references also describe the x87 (floating-point) instruction set architecture and the MMX extensions.

The K6 3D also supports AMD-developed instruction set extensions to the MMX instructions, called the 3D instructions, which support high-performance 3D graphics, audio, and physics processing. The resulting microprocessor has a decoupled superscalar microarchitecture that uses many advanced design approaches targeted at achieving high performance. Some of the techniques that are employed in the K6 3D design are: instruction predecoding, multiple x86 instruction issue in a single clock cycle, internal single-clock RISC-like operations, superscalar operation (concurrent use of multiple execution units to execute up to six RISC-like operations per clock cycle), out-of-order execution, data forwarding, implicit register renaming, speculative execution, and the use of in-order retirement to ensure precise interrupts

Video Clips on CD-ROM

Greg Favor, Chief Architect of the K6 3D addresses the following two questions on two related video clips, "What were the principle design objectives chosen for the K6 3D microprocessor?" and "What are the key microarchitectural features incorporated into the K6 3D?"

The K6 3D uses a two-level, dynamic branch direction prediction technique that is integral to its ability to execute instructions speculatively. The branch direction prediction logic makes use of a branch history table, a branch target cache, and a return address stack, all of which combine to achieve a predicted address hit rate of better than 95%. These and other design techniques, such as employing a six-stage pipeline, enable the K6 processor to fetch, decode, issue, execute, complete, and retire multiple x86 instructions per clock. The material in this introduction provides a general overview of the K6 3D microprocessor and will be discussed in more detail in Chapter 3.

Design Note

K6 3D Code Optimizations

The coding techniques for optimizing peak performance of the K6 3D include many of those recommended for optimizing the performance of the Intel Pentium and Pentium Pro microprocessors. They also include optimizations specific to the K6 3D's implementation of the MMX and 3D instruction set extensions. The use of the K6 3D code optimizations can result in higher delivered performance than off-the-shelf software non optimized code.

A Family of Microprocessors

The K6 is a family of microprocessors. The initial member of the family, called the K6 MMX, was introduced (in May 1997) at clock speeds of 166-MHz, 200-MHz, 233-MHz, and 266-MHz, had 8.8 million transistors and was designed in a 0.35-micron process resulting in a 162-mm^2 die-size. These chips have a 66-MHz processor bus. Shrinking to a 0.25-micron process and architectural and microarchitectural enhancements (e.g., the inclusion of the 3D instructions) grew the chip to 9.3-million transistors, yet the die size shrank to 81 mm^2. This later member of the K6 family, called the K6 3D, supports the 3D instruction set extensions. The K6 3D, extended to have a 256-Mbyte L2-Cache on-chip, is called the K6 3D and it is this processor that is discussed in detail in this book. From time-to-time reference will be made to the earlier versions of the K6 family that did not support the 3D instruction set or did not have the L2-Cache on-chip, but the references make clear when this is done. These processors will simply be referred to as the K6 and will be used to describe features that are applicable to both the K6 and the K6 3D.

The characteristics of some of the members of the K6 Family are summarized in the following table. Each family member extends the microprocessor in the row above it.

Table 2.1 K6 Family Members

Processor	Clock	Bus	Process[a]	Die-Size	TC[b]	Comments
K6 MMX	166-MHz	66-MHz	0.35	162-mm2	8.8	original release
	200-MHz	66-MHz	0.35	162-mm2		higher clock
	233-MHz	66-MHz	0.35	162-mm2		higher clock
	266-MHz	66-MHz	0.35	162-mm2		higher clock
	300-MHz	66-MHz	0.25	68-mm2		K6 MMX shrink
K6 3D	300-MHz	100-MHz	0.25	81-mm2	9.3	K6 with 3D[c] & MMX[d]
	350-MHz	100-MHz	0.25	81-mm2		higher clock
K6 3D	350-MHz	100-MHz	0.25	135-mm2	21.3	K6 3D with L2-Cache on-chip
	400-MHz	100-MHz	0.25	135-mm2		higher clock

[a] in microns

[b] transistor count in millions

[c] also called "AMD-3D Technology"

[d] a superscalar dual-pipeline implementation of the x86 MMX instruction set extensions

The K6 MMX was initially implemented in a 0.35-micron CMOS process and then in a 0.25-micron CMOS process, using five layers of metal, shallow trench isolation, and tungsten local interconnect. C4 solder bump flip-chip technology is used to assemble the die into a ceramic pin grid array (PGA). The high-performance and small die sizes of these microprocessors are achieved using high-speed custom and macro blocks and placed-and-routed blocks of standard cells. The initial 0.25-micron process version of the K6 MMX can operate at a clock speed of 300-MHz, has a 100-MHz processor bus and its chip sets support AGP (Advanced Graphics Port), USB (Universal Serial Bus), and the IEEE 1394 high-performance serial bus (a.k.a. Firewire), all of which are presented in detail in Chapter 5. A die photograph of a recent version of the K6 and an overlay on top of the photograph showing the approximate placement of various K6 components is given in Figure 2.1.

Technical Presentation on CD-ROM

A presentation by Greg Favor, Principal Architect of the K6 3D, on the evolution of the K6 family entitled, "*The AMD-K6 MMX Enhanced Processor Product Roadmap*," can be found on the CD-ROM.

.
Articles on CD-ROM

To learn more about some of the detailed implementation issues of an early version of the K6 3D processor, see "Circuit Techniques in a 266-MHz MMX-Enabled Processor," by Don Draper, Matt Crowley, John Holst, Greg Favor, Albrecht Schoy, Jeff Trull, Amos Ben-Meir, Rajesh Khanna, Dennie Wendell, Ravi Krishna, Joe Nolan, Dhiraj Mallick, Hamid Partovi, Mark Roberts, Mark Johnson, and Thomas Lee. This article, appeared in the November 1997 issue of the *IEEE Journal of Solid-State Circuits*. To learn more about some of the detailed implementation issues of the 0.35-micron version of the K6, see "An x86 Microprocessor with Multimedia Extensions," by Don Draper, Matthew P. Crowley, John Holst, Greg Favor, Albrecht Schoy, Amos Ben-Meir, Jeff Trull, Raj Khanna, Dennie Wendell, Ravi Krishna, Joe Nolan, Hamid Partovi, Mark Johnson, Tom Lee, Dhiraj Mallick, Gene Frydel, Anderson Vuong, Stanley Yu, Reading Maley, and Bruce Kaufmann 1997 *ISSCC Digest of Technical Papers*. You can find the full test versions of both of the above articles on the companion CD-ROM.

Microprocessor Fabrication Process Mini-Tutorial on CD-ROM

This video clip, in which Bill Siegle, AMD's Chief Scientist, addresses several questions related to the steps involved in the fabrication of contemporary microprocessor chips, effectively becomes a mini-tutorial on the fabrication process.

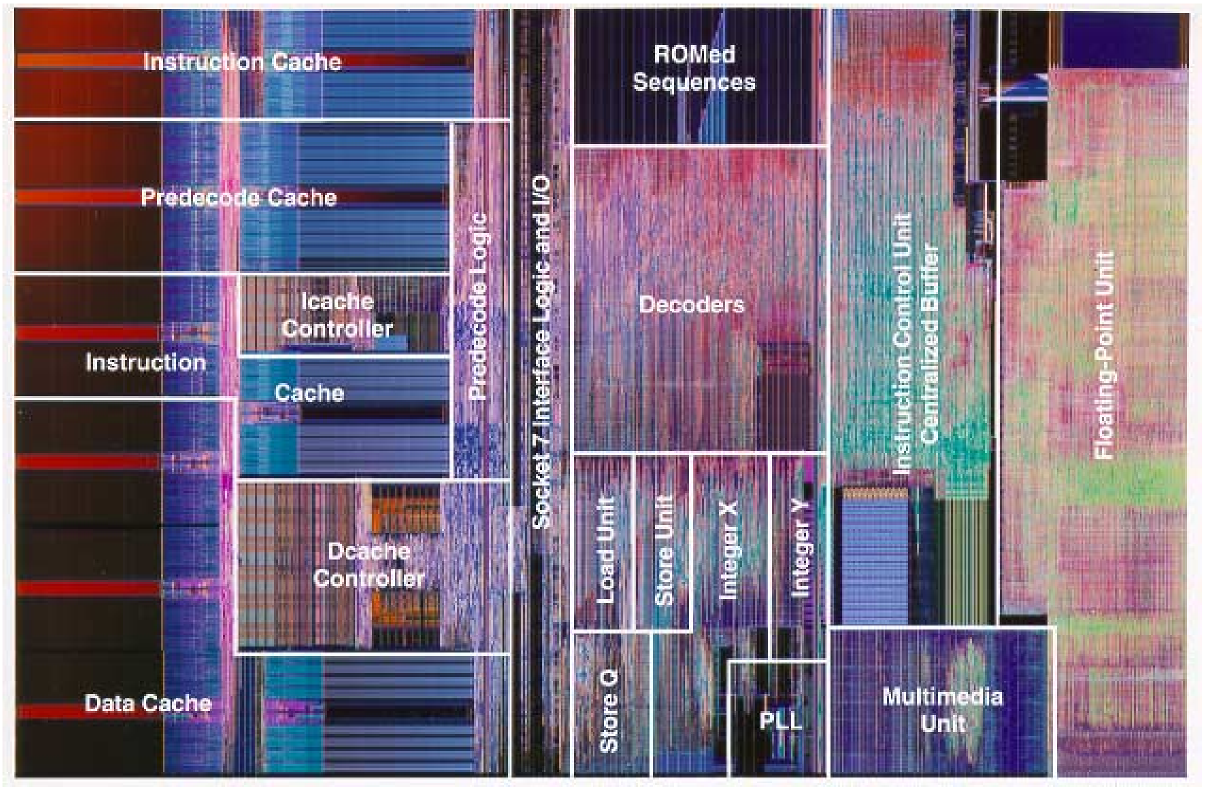


Figure 2.1 K6 Die Photograph and Overlay

K6 3D Block Diagram

Figure 2.2 is a high-level block diagram of the K6 3D microarchitecture. We will give an overview of its operation first, followed by more detail about each of the elements shown in this figure.

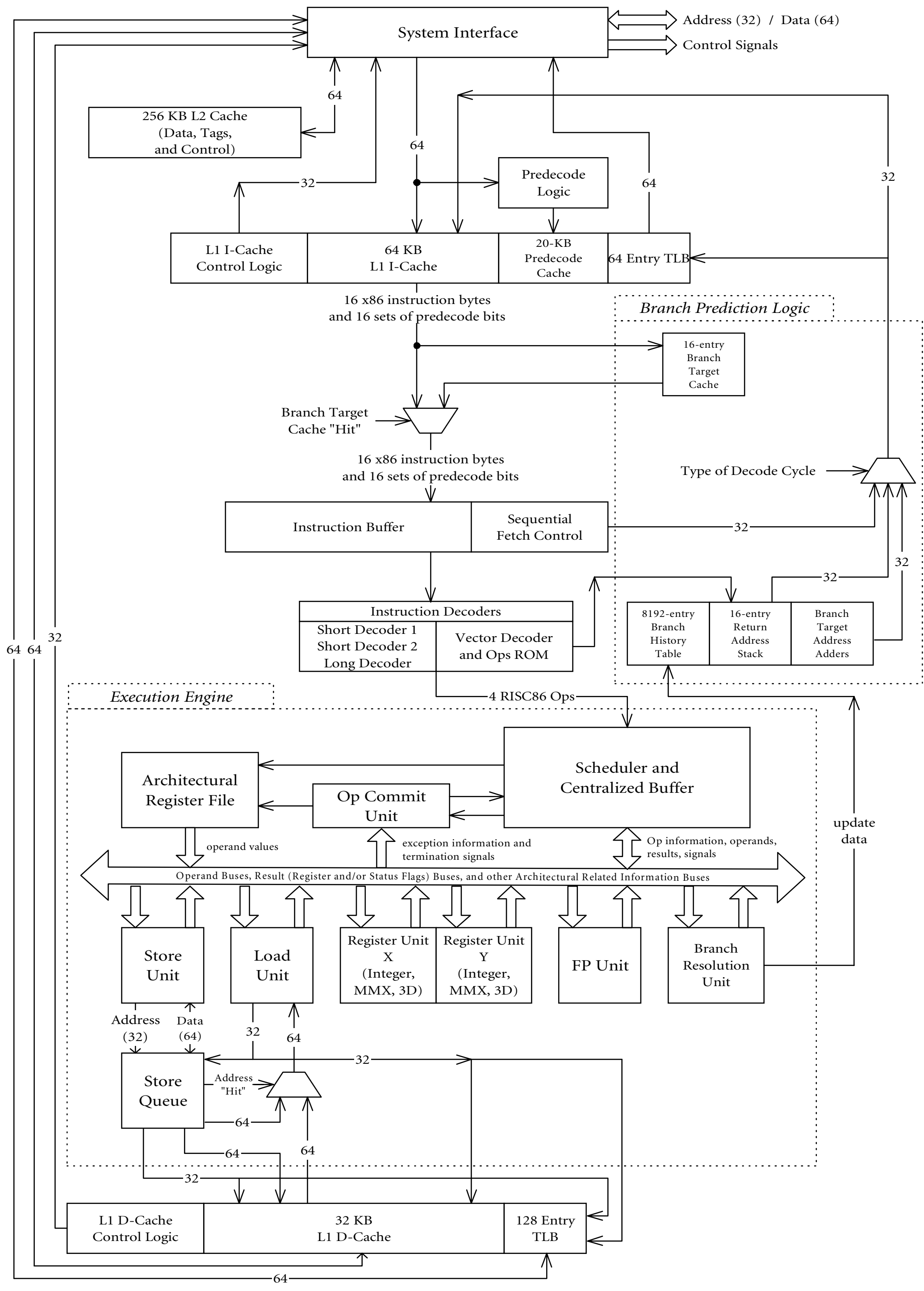

System Interface
Address (32) / Data (64)
Control Signals
64
256 KB L2 Cache (Data, Tags, and Control)
64
32
Predecode Logic
64
32
L1 I-Cache Control Logic
64 KB L1 I-Cache
20-KB Predecode Cache
64 Entry TLB
16 x86 instruction bytes and 16 sets of predecode bits
Branch Prediction Logic
16-entry Branch Target Cache
Branch Target Cache "Hit"
Type of Decode Cycle
16 x86 instruction bytes and 16 sets of predecode bits
Instruction Buffer
Sequential Fetch Control
32
32
32
Instruction Decoders
Short Decoder 1
Short Decoder 2
Long Decoder
Vector Decoder and Ops ROM
8192-entry Branch History Table
16-entry Return Address Stack
Branch Target Address Adders
64 64 32
Execution Engine
4 RISC86 Ops
Architectural Register File
Op Commit Unit
Scheduler and Centralized Buffer
update data
operand values
exception information and termination signals
Op information, operands, results, signals
Operand Buses, Result (Register and/or Status Flags) Buses, and other Architectural Related Information Buses
Store Unit
Load Unit
Register Unit X (Integer, MMX, 3D)
Register Unit Y (Integer, MMX, 3D)
FP Unit
Branch Resolution Unit
Address (32)
Data (64)
32
64
Store Queue
Address "Hit"
32
64
64
64
32
L1 D-Cache Control Logic
32 KB L1 D-Cache
128 Entry TLB
64

Figure 2.2 K6 3D Block Diagram

We will, from time to time as appropriate, use the term *bus cycle* when referring to external data transfers and *processor cycle* when referring to operations internal to the microprocessor. There is typically a 3X ratio between these two cycles in the K6.

X86 instructions are stored in the main memory. During each bus cycle, up to eight bytes of x86 instructions are fetched from main memory or the on-chip L2-Cache and loaded into the on-chip *L1 Instruction Cache* (the *L1 I-Cache*) during a cache fill. While they are being loaded into the L1 I-Cache, the x86 instruction bytes are predecoded, using predecoded logic, to assist in later, rapid identification of x86 instruction boundaries.

During each processor cycle, the L1 I-Cache or the Branch Target Cache (BTC) places 16 x86 instruction bytes into a 16-byte instruction buffer which directly feeds the instruction decoders. The multiple instruction decoders (two short decoders, a long decoder, and a vector decoder), taken as an aggregate, will be referred to as the decoders. The decoders, using a combination of the predecoded information and the x86 instruction bytes in the instruction buffer, produce and load four RISC-like operations, called *RISC86 operations*, into the scheduler of the execution engine. RISC86 operations are also called *RISC86 Ops*, *Ops*, or merely *operations*. Our usual term will be Ops. Each cycle, the decoders decode up to two x86 instructions to produce and load a set of up to four RISC86 operations into the scheduler.

Historical Comment

Peter Kogge's Insightful Book

The x86 instruction set architecture began with the Intel 4004 microprocessor, designed in 1969. Architectural innovations such as the use of pipelining, superscalar, speculative, and out-of-order execution were used in a number of mainframe computers at that time. Although these design techniques were not used in early generations of x86 microprocessors, recent members of the x86 family have employed all of them. See Peter M. Kogge's insightful book, *The Architecture of Pipelined Computers*, McGraw-Hill Book Company, 1981, for a detailed technical discussion and an interesting and reasonably complete history of the evolution of these design techniques in many important pre-microprocessor architectures that helped shape many of the current microarchitectural design approaches.

RISC86 Ops are RISC-like, fixed-format, internal Enhanced RISC86 microarchitecture instructions. Taken together, they form the "RISC86 operation set." Generally, all execute in a single clock cycle; register operations have a one-or two-cycle latency and load and store operations have a two-cycle execution latency. RISC86 Ops can be combined, as required, into sequences of Ops to perform every function of the x86 instruction set.

DESIGN NOTE

Enhanced RISC86 Microarchitecture

The Enhanced RISC86 microarchitecture and its underlying RISC86 operation set are optimized for execution of the x86 instruction set architecture, while adhering to the architectural principles of fixed length encoding, regularized fields, and a large register set, common in most RISC architectures.

Some x86 instructions are decoded (translated) into as few as zero Ops (e.g., a RISC86 NoOp) or one Op (e.g., a RISC86 register-to-register add Op). More complex x86 instructions are decoded into several Ops. A more detailed treatment of the operation set is given later in this chapter and Chapter 3. There are six types of Ops:

Table 2.2 TYPES OF RISC86 OPS

Types of Ops	Mnemonic
memory load operations	LdOps
memory store operations	StOps
integer register operations, MMX register operations, and 3D register operations	Integer, MMX, and 3D RegOps
floating-point register operations	FpOps
branch condition evaluations	BrOps
special operations (such as load immediate constant into a register)	SpecOps

When a particular discussion is applicable to both LdOps and StOps, the terms LdStOp or LdStOps will be used, as appropriate. The following simple example gives a series of x86 instructions and corresponding decoded RISC86 Ops using the resources shown in Figure 2.2.

:

EXAMPLE CODE FRAGMENT

x86 instruction	Type of Op	Comment
MOV CX,[SP+4]	LdOp	The MOV instruction is decoded into a Load Op that requires data to be loaded from memory using the Load Unit.
ADD AX,BX	RegOp	The add instruction is decoded into an ALU Add Op that can be sent to either Register Unit X or Register Unit Y.
CMP CX,[AX]	LdOp, RegOp	The CMP (compare) instruction is decoded into two Ops. A Load Op requiring data to be loaded from memory using the Load Unit followed by an ALU Sub Op that would be sent to either Register Unit X or Register Unit Y. Static flag values produced by the Sub Op reflect the result of the comparison.
JZ TA	BrOp	Conditional branch to "TA" (Target Address) based on Zflag = 1

L1–CACHE, L2– CACHE, STORE QUEUE, AND SYSTEM INTERFACE

The execution engine interfaces to the on-chip 64-Kbyte L1-Cache. This cache is split into the 32-Kbyte L1 I-Cache mentioned earlier and a 32-KByte L1 Data Cache (L1 D-Cache). Split caches, such as these, are sometimes referred to as Harvard Architectures. One of the first references to this term can be found in Cragon, H. G., "The Elements of Single-Chip Microcomputer Architecture," *Computer*, Vol. 13, No. 10, October 1980, pp. 27-41.

Both caches are 2-way set associative with a 64-byte line size and 32-byte subblocking. There are 256 sets in each cache and each set contains two ways (or lines). Cache lines are fetched from main memory or the on-chip Level-2 Cache (L2-Cache). using a burst bus transaction of four octets (or four quadwords in x86 terminology). Bus transactions are discussed in detail in Chapter 4.

The L2-Cache is a 256-Kbyte unified cache, is 4-way set associative, and has a 64-byte line-size with 32-byte subblocking. The L2-Cache employs a true LRU replacement algorithm. A store queue is used in conjunction with the L1 D-Cache. *Abortable* state changes are supported by the scheduler and the store queue through the general technique of temporarily storing (a) register and status results in the scheduler entries and (b) memory write data in store queue entries until the associated Ops are committed and retired.

DEFINITIONS

Abortable and Nonabortable

Abortable refers to changes that can be speculatively performed and later backed out of. Nonabortable changes cannot be backed out of once they are performed,

As will be seen in more detail later, the L1 I-Cache supports single cycle accesses. Both of the K6 3D's L1-Caches are interfaced through the system interface to the L2-Cache. The L1-Cache and L2-Cache are key to the scalability and performance of the K6 as the core frequencies increase.

HISTORICAL COMMENT AND DESIGN NOTES

Cache-Related Issues

In order to increase the access bandwidth, the L1 D-Cache is pipelined. It supports simultaneous loads and stores in a single clock. Bank conflicts are eliminated by performing loads first, followed by stores in a pipelined manner. Each access takes one clock cycle of time; the start of store accesses is offset by half a cycle from the start of load accesses.

Write performance is enhanced using a full write-back policy. Write-back caches are also called copy back, store in, nonstore through, or swapping caches in the literature. When data are written to a specific cache line, its "modified" (or "dirty") bit is set to indicate this. The cache line is actually stored in main memory only when the cache line is replaced. The intent is to reduce the overall traffic on the bus.

In contrast to the K6 3D, the K6 does not have an on-chip L2-Cache. The K6 has full support for an external (off-chip) L2-Cache, including a means for inhibiting the normal operation of its on-chip L1 I-Cache and L1 D-Cache (which are identical to the K6 3D's L1-Caches). This capability allows designers to disable the on-chip L1-Caches while testing the external L2-Cache. A complete description of the K6's L1-Caches and L2-Cache support can be found in the AMD-K6 MMX Processor Data Sheet which is on the companion CD-ROM. This also means that the system interface, shown in Figure 2.2 on page 69 and Figure 2.22 on page 180 for the K6 3D, is somewhat different for the K6.

Cache coherency is maintained using the MESI protocol. More will be said about the L1 and L2 caches later.

As mentioned earlier, during each processor cycle, the L1 I-Cache can place x86 instruction bytes into a 16-byte instruction buffer which directly feeds the decoders. The decoders produce and load four Ops into the scheduler's centralized buffer. The four Ops taken together are called an OpQuad. The scheduler is the heart of the K6 microarchitecture. It contains the logic necessary to manage out-of-order, speculative execution, data forwarding, implicit register renaming, and the simultaneous issue, execution, and retirement of multiple Ops per cycle. The scheduler's centralized buffer can hold up to twenty-four Ops. This is equivalent to six to twelve x86 instructions. The scheduler will be discussed in substantial detail in this book because of its importance and unique design.

Superscalar Design

Superscalar processors contain a number of execution units that can operate in parallel. The K6 3D is such a processor. It has six specialized execution units that can operate in parallel and are shown in Figure 2.2 on page 69: the Store Unit (SU), Load Unit (LU), Register Unit X (RUX), Register Unit Y (RUY), Floating-Point Unit (FPU), and the Branch Resolving Unit (BRU). As will be seen, RUX and RUY can execute integer, MMX, and 3D instructions.

The non-3D versions of the K6 actually have seven execution units. These microprocessors have a separate MMX unit that overlaps with the operation of RUX. More will be said later about the differences between the versions of the K6 that support the 3D instructions and those which do not, as appropriate throughout this book.

Historical Comment and Suggested Readings

x86 Instruction Set Architecture MMX Extensions

Intel publicly released many details of its MMX extensions to the x86 instruction set architecture in March, 1996 in a rather extensive San Francisco news release, "Intel Releases MMX™ Technology Details to Software Community to Drive New Multimedia, Game, and Internet Applications." This news release can be found on the Developers' Insight CD-ROM, Intel Corp., April 1997, Reference SKU #273000, Intel Corporation, 5000 West Chandler Blvd. CH6-413, Chandler, AZ 85226. These extensions consist of new instructions and data types aimed at increasing the performance of x86 processors in multimedia applications. Implementations of the instructions can make use of SIMD (single-instruction stream, multiple-data stream) techniques to process multiple 8, 16, or 32-bits in a 64-bit data path to achieve highly parallel performance in compute-intensive multimedia code. The instruction set extensions consist of 57 new instructions that support addition, subtraction, multiplication, multiply-accumulates, logical or arithmetic shifts, and several other operations that can be executed on all three sizes of data. See *MMX™ Technology Technical Overview* and the *MMX™ Technology Developers' Guide*, both which also are on the Developers' Insight CD-ROM cited above. Another source for related material is the Carole Dulong, David Bistry, Mickey Gutman and Mike Julier book, *The Complete Guide to MMX Technology*, McGraw-Hill, 1997.

We would like to summarize some of the notations used so far. The following sets of terms, listed in alphabetical order for convenience, are used as synonyms in the text and the companion CD-ROM:

1. branch resolving unit and BRU.
2. decoders and instruction decoders.

3. instruction(s) and x86 instruction(s.)
4. K6 3D, K6, K6 microarchitecture, and RISC86® microarchitecture.
5. L1 I-Cache and L1 Instruction Cache.
6. L1 D-Cache, L1 Data Cache, and L1 Dual-Ported Data Cache.
7. operation(s), Ops, RISC86 operation(s), and RISC86 Op(s).
8. 3D Ops.
9. Register Unit X and RUX.
10. Register Unit Y and RUY.

Historical Comment and Suggested Readings

K6 3D Technology

K6 3D Technology consists of a set of extensions to the x86 instruction set architecture. Most of these new instructions can be viewed as being floating-point analogs of the MMX instructions discussed in the preceding "Historical Comment and Suggested Reading" inset. Whereas MMX instructions operate on packed sets of 8-bit, 16-bit, and 32-bit fixed point or integer values (within 64-bit wide MMX registers), the 3D instructions operate on packed pairs of 32-bit single-precision IEEE-compatible floating-point values (also within the same 64-bit wide MMX registers). In both cases these are "single instruction stream multiple data stream" or "SIMD" type instructions.

The 3D instruction set extensions were developed with the goal of greatly accelerating floating-point intensive computations, many of which contain substantial parallelism and thus an opportunity to benefit from SIMD floating-point instructions (in contrast to existing scalar x87 instructions within the x86 instruction set architecture). From an application software perspective, the goal was to greatly accelerate a range of multimedia algorithms, particularly in the area of 3D graphics and games. With the increasing use of 3D graphics, hardware accelerators are becoming more popular. These accelerators focus on backend graphics processing, i.e., the triangle/pixel rendering stages of the 3D graphics processing pipeline. Given this, the front-end stages of the graphics pipeline are becoming the performance bottleneck. These stages, which perform geometry transform, clipping, and lighting computations are all floating-point intensive and benefit substantially from the use of 3D instructions. In addition, as games and other multimedia applications evolve toward increasingly more accurate and physics-based modeling of 3D worlds and the interactions between objects in these worlds, the need for even greater levels of floating-point performance continues to grow. Ultimately it is expected that physics-based modeling and simulation computations will equal and surpass the traditional 3D graphics processing pipeline in the amount of floating-point computations and performance that is required. Other areas, such as audio and speech processing, and artificial intelligence/neural network algorithms, will also add to this.

An AMD application note describing the MMX extensions and related optimization, *AMD-K6 MMX Enhanced x86 Code Optimization*, can be found on the companion CD-ROM, as well as a application note related to K6 3D K6 3D optimizations, *AMD-K6 3D Processor Code Optimization*.

The decoders place four Ops in the scheduler's centralized buffer each cycle. The buffer's size, matched to the typical execution lifetime of Ops, allows the decoders to operate largely independently of the execution units. Such a buffer is often called an instruction window, which we formally define below.

The scheduler's issue logic examines the Ops in the buffer, selecting appropriate ones subject to dependencies and resource constraints. It is capable of issuing up to six Ops, out-of-order, each cycle, independently of the decoders. The execution engine can also execute them out-of-order, independently of the decoders. A microprocessor is said to have a decoupled decode/execution microarchitecture when the decode of instructions takes place independently of the issuing and execution of operations. Thus, the K6 is a decoupled decode/execution superscalar processor.

After completion, Ops are committed in-order by the Op Commit Unit (OCU), shown in Figure 2.2 on page 69. The scheduler effectively serves as a re-order buffer to ensure precise exceptions and x86 compatibility.

Historical Comment, Definition, and Suggested Readings

Instruction Windows and Reservation Stations

An instruction window allows a scheduler to optimize the execution of operations by issuing them to the appropriate execution units as the units are available and as various dependencies allow. There are two basic ways to implement instruction windows: centralized or distributed.

Distributed instruction windows, typically called *reservation stations*, are located with each functional unit. The reservation stations at the functional units can be (and often are) different in size from one another. Although their individual sizes are smaller, the aggregate size of the reservation stations is typically larger than the size of the single instruction window to achieve the same amount of instruction look ahead.

Centralized instruction windows provide the storage for both the operands and results of the functional units. The K6 scheduler buffer may be viewed as a type of centralized instruction window. More will be said about this later. Note that we have been using the terminology "x86 instruction" and "RISC86 operation." We will use this terminology consistently throughout this book. The K6's scheduler's buffer can be thought of as providing either an *instruction window* or an Op window since x86 instructions are decoded into RISC86 Ops. We will use the term *centralized buffer* or *buffer* when referring to the K6's Op window.

The first article that the authors are aware of that introduced the use of reservation stations was the insightful and seminal article by R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, Vol. 11, January 1967, pp. 25-33. The full text version of this article is on the companion CD-ROM. Similarly, the first article we are aware of which uses a form of centralized instruction window is James E. Thornton's article, "Parallel Operation in the Control Data 6600," *Proc. AFIPS Fall Joint Computer Conference*, Part. II, 1964, pp. 33-40, in which the scoreboard of the CDC 6000 is described.

As is seen later, we use the terms *retired*, *committed*, and *removed* in a particular way in this book. Retiring an operation does not imply the results of the operation are either permanent or non permanent. We will use the term *committed* to mean that the results of an operation have been made permanent and the operation retired from the scheduler. *Retiring* means removal from the scheduler with or without the commitment of operation results, whichever is appropriate. Timing-wise, commitment and retirement often happen simultaneously. We will use the term *removed* to mean the operation is retired from the scheduler without making permanent changes.

Definitions

Decoupled Decode and Execution
Decoupled Execution and Commitment

A microprocessor is said to have a decoupled decode/execution microarchitecture when the decode of instructions takes place independently of the issuing and execution of operations.

However, microprocessors that support out-of-order execution have an equally important decoupling. In such processors, results are often produced out-of-program-order as the various operations may issue out-of-order and may take different amounts of time to complete. The process of commitment (i.e., making permanent changes in the architecture's state) is decoupled from the execution of the operations. This allows the facility that commits the results to re-order them in program order. A microprocessor is said to have a decoupled execution/commitment microarchitecture when the execution of operations takes place independently of the commitment of the results of these operations.

The Execution Units

As we mentioned when discussing the K6 3D block diagram, we said it has six specialized execution units that can operate in parallel: the Load Unit (LU), Store Unit (SU), Register Unit X (RUX), Register Unit Y (RUY), Floating-Point Unit (FPU), and the Branch Resolving Unit (BRU).

The LU and the SU are pipelined execution units. Before summarizing their functionality, we need to review some aspects of x86 address calculations.

x86 Address Calculations

Physical, Virtual and Logical Addresses

The x86 instruction set architecture defines a word as two bytes or sixteen bits. A double word is four bytes or thirty-two bits. The phrase "double word" is often abbreviated as "dword." The x86 architecture treats physical memory as a linear array of bytes. Each byte has a unique address which is known as its *physical address*. Since the x86 instruction set architecture uses byte addressing, memory is organized and accessed as a sequence of bytes. No matter if one or more bytes are being accessed in the x86's address space, a byte address is used to locate the first byte of the set of bytes to be accessed.

Programs that execute on the x86 use a two-part address which is translated, or mapped, into physical addresses. The translation is done by an address translation mechanism and the two-part addresses are often called *virtual addresses* because these addresses do not correspond directly to a physical address, but correspond indirectly to one through the address translation mechanism. Virtual addresses are sometimes called *logical addresses*. The virtual-to-physical mapping mechanism also provides for both memory protection and the determination of a *valid* address (i.e., that an address is present in memory).

The two-part virtual address consists of a 16-bit segment selector and a 32-bit offset. The x86 employs a two-stage mapping mechanism to translate the two-part selector and offset virtual address into a physical address. The virtual address is first translated via a segmentation mapping mechanism into a 32-bit *linear address*. The linear address is then translated into a 32-bit *physical address* via a page mapping mechanism. Thus, the two well-known virtual memory mapping techniques—segmentation and paging—are used. For the specific segmentation and paging techniques defined for the x86 instruction set architecture, see "Programming the 80386," by John H. Crawford and Patrick P. Gelsinger, Sybex, 1987.

Given this brief background in the text inset, we can return our discussion of the LU. The LU performs data memory reads. When the LU unit receives its operand values, it first performs the general calculation,

$$\text{base register} + \text{scaled index register} + \text{displacement}$$

yielding the x86 architecturally defined logical address, adds this to the segment base address to produce the architecturally defined linear address which is checked against the segment limit. The linear address is also translated to a physical address by the LU using the data translation lookaside buffer, i.e., data-TLB or D-TLB. The physical address is sent to the store queue and to L1 D-Cache in parallel. Typically data is received from the L1 D-Cache and, assuming a cache hit occurs, the data coming out of the L1 D-Cache is driven onto the LU's result bus. However, if the address matches a store queue entry then the store queue entry takes priority over a hit in the cache and data for the store queue entry is driven onto the LU's result bus. Both the L1 D-Cache and store queue "hit" analyses are based on the comparison of physical addresses received from the data-TLB, even though the D-Cache indexing is based on linear address bits.

We will see later that the LU is pipelined. The LU's pipelined design has the advantage of limiting the penalty for misaligned data loads to a latency of one cycle longer. In the x86 instruction set architecture, a misaligned access occurs either when an 8-byte (a quadword) access is made to an address that is not on an 8-byte boundary, or when a word or a double word access is made to an address that is not on a 4-byte boundary. Misaligned accesses are discussed in more detail later in this chapter, see "Faults, Traps, Abort Cycles, and the Pipelines" on page 168. Data are available from the LU after only two clocks or three clocks in the case of a misaligned access.

DESIGN NOTE

Linearly Indexed and Physically Tagged L1 Cache

Both the L1 I-Cache and the L1 D-Cache are linearly indexed and physically tagged. The L2 Cache is physically indexed and physically tagged. These concepts will be discussed later.

The SU performs address calculations for all store operations as well as for the load effective address and push operations. Logical and linear address calculations and translation to a physical address finish by fetching the memory write data from a register. Upon completion, the SU creates an entry in the store queue to hold the memory write address and data information. The store queue serves to buffer memory writes until they can be committed into the L1 D-Cache. Forwarding of write data from a store queue entry to dependent LdOps is supported. We will learn later that these entries are not valid until the very next cycle after the StOp execution completes. We will also learn the SU is also pipelined

Design Note

Memory Aligned and Register Aligned Data

Data stored in the L1 D-Cache and store queue are memory aligned. This means that byte[0] of the 64-bit wide bus always carries bytes where the lower three bits are zero, so they are effectively memory byte addresses. However, data that ends up on the result bus must be register aligned. If a one-byte read is being done, the one valid byte must be on the low byte of the bus, if a two-byte read is being done, then two valid bytes must be on the lower two bytes of the bus, and so on. The K6 employs byte rotators to convert (or map) between these two alignments. See Figure 2.16 on page 165 and Figure 2.19 on page 168.

The RUX (Register Unit X) supports all integer ALU operations—multiplies, divides (signed and unsigned), shifts, and rotates. It can also perform MMX and 3D operations. The RUY (Register Unit Y) can operate on the basic word and double word integer ALU operations—ADD, AND, CMP, OR, SUB, XOR, zero-extend and sign-extend operations. It can also perform MMX and 3D operations. In fact, RUX and RUY share arithmetic resources to execute some of the MMX operations and all of the 3D operations. The relationship between RUX and RUY and the integer, MMX, and 3D operations that each register unit supports is shown in Figure 2.3. The K6 3D has two pipelines for executing integer, MMX, and 3D RegOps.

Generally speaking, RUX and RUY are symmetric pipelines. This means that any Op can issue to either pipeline. The one exception is that some of the integer Ops can only be executed by the RUX pipeline as shown in Figure 2.3.

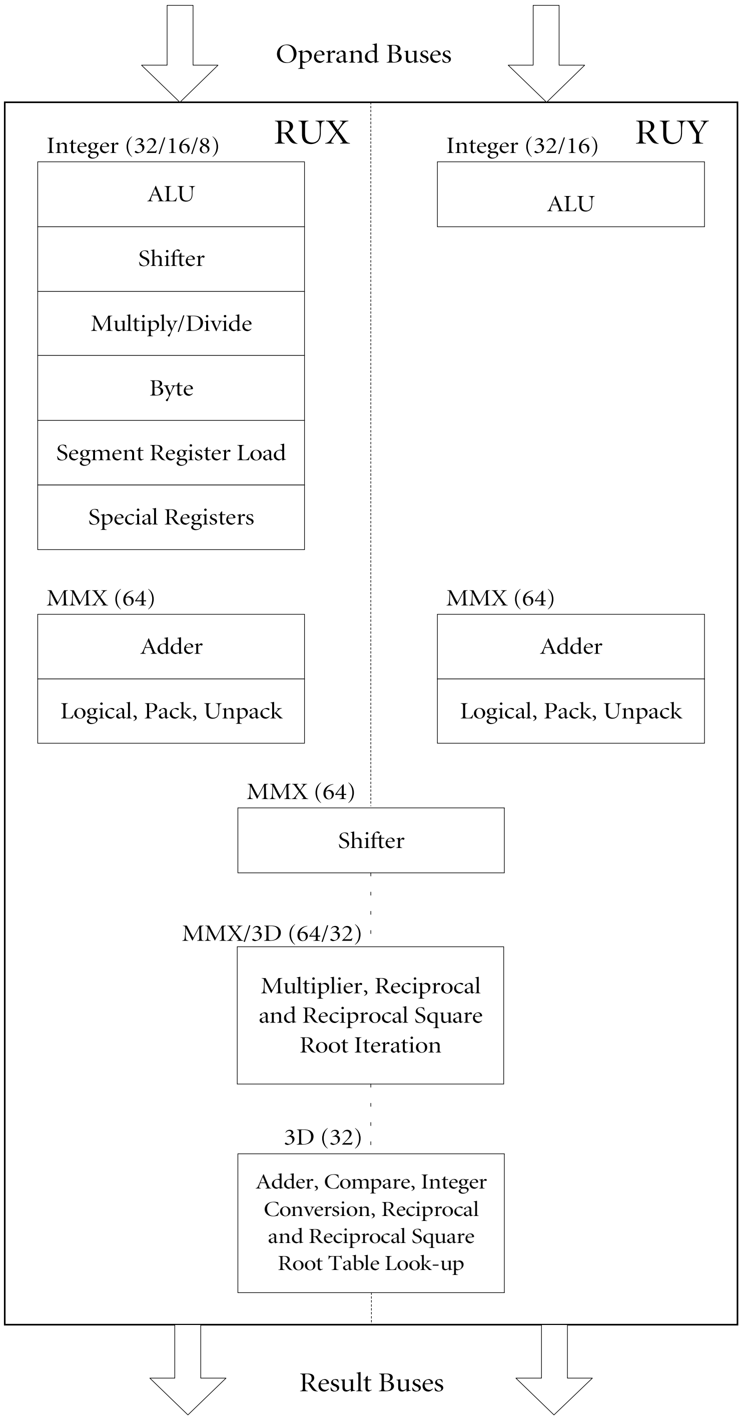


Figure 2.3 RUX AND RUY EXECUTION UNITS

Leaving these Ops aside, the scheduler can issue an Op to either pipeline. Duplicate resources for an Op are either available or they are not available, (e.g., there are two integer ALUs and two MMX adders, but there is only one MMX multiplier). An Op that has duplicate resources available to it can proceed down either pipeline irrespective of what operations may be processing down the other pipeline. In particular, there can be two such Ops simultaneously in execution, one proceeding down each of the two pipelines. Ops that have only one copy of the execution logic available can proceed down the appropriate pipeline. Two such Ops can proceed down both pipelines but cannot start to execute simultaneously which means that one of the OPs incurs a one-cycle *pipeline stall.*

3D instructions can be considered to be a floating-point analog of the integer MMX instructions. Their primary purpose is to provide high-performance floating-point vector operations to enhance performance on 3D graphics-oriented applications. The 3D instructions that operate in a vector fashion operate on two sets of 32-bit single-precision floating-point numbers in parallel. The FPU, in contrast, operates internally on a single pair of floating-point numbers that have an 80-bit representation in accordance with the IEEE floating-point standard (see IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard No. 754, 1988). The 3D floating-point operations are realized in RUX and RUY. All non-3D floating-point operations are executed in the FPU. More will be said shortly about the last of the six execution units, the BRU (Branch Resolving Unit), and the branch direction prediction logic which are shown in Figure 2.3 on page 81.

Historical Comment and Suggested Readings

The K6 Floating-Point Unit

The FPU, in all members of the K6 family of microprocessors, is a direct descendant of the FPU-core that appeared in the NexGen Nx586 microprocessors, which was designed to be instruction set architecture compatible with the Intel x87 floating-point unit. (The x87 instruction set architecture is the instruction set architecture of the x87 floating-point unit.) Thus the K6 family of microprocessors uses some concepts native to the Nx586, such as tags (see the Nx586 Databook cited below). There are a number of important design implications that arise from the decision to use this core. For example, the FPU operates out of its own register file with its own rename registers. The x87 has two 80-bit operands coming in and one 80-bit result coming out in addition to architecturally defined floating-point flag bits that reside in the architecturally defined floating-point status word register. Further, floating-point operations may also modify an architecturally defined top-of-stack pointer field. See *Nx586 Processor Databook*, NexGen Inc., Preliminary, December 6, 1994, Order # NxDOC-DB001-03-W.

Latencies

A brief word about *latencies*. In the K6 3D, register operations fall into one of three categories: those that have a latency of one cycle, those that have a latency of two cycles, and those (like divide) that have a latency of more than two cycles. In general, all integer operations and all of the non multiply MMX instructions have a single cycle latency. MMX multiply operations have a two-cycle latency. All of the 3D operations have a two-cycle latency. Subject to dependencies and execution resource contention, two RegOps can start execution each clock regardless of whether they are one-cycle and/or two-cycle operations. For example, a 3D multiply and a 3D add can begin execution each clock cycle. At most one two-cycle latency Op, for which there are not duplicated execution resources (such as an MMX multiply), can be initiated in each cycle. One of the Ops of a pair of Ops wanting to start execution must be delayed if the Ops use a shared execution resource. This results in a delay of one cycle. Back-to-back two-cycle latency Ops that use different execution resources (such as an MMX multiply and a 3D add) can be initiated in the same cycle, one in the RUX pipeline and one in the RUY pipeline. It is important to know instruction latencies when examining instruction dependencies.

Example Code Fragments on CD-ROM

The following two application notes on the CD-ROM contain a number of examples of the timings and latencies of the execution behavior of several code fragments as a function of decode constraints, dependencies, and resource constraints, "AMD-K6 3D Processor Code Optimization," and "AMD-K6 MMX Enhanced Processor x86 Code Optimization."

The details of the inputs and outputs of RUX and RUY on the operand and result buses in Figure 2.3, as well as the inputs and outputs for each of the execution units shown in Figure 2.2 on page 69, will be discussed in later sections of this chapter. Briefly, for example, RUX and RUY take their inputs from the register operand buses, execute the required Ops producing register and status flag result values, and drive them out onto the corresponding register result and status flag buses. It is up to the scheduler to keep track of which result and status flag values are scheduled (or marked) to actually be modified.

Status Flags, Faults, Traps, Interrupts, and Abort Cycles

The x86 instruction set architecture supports a register called the EFLAGS register that contains a number of x86 status and control flags (see for example, Intel's publication, the *Intel Architecture Software Developer's Manual, Volume 3: System programming Guide* for a complete listing of all x86 status and control flags, exceptions, and interrupts). The values of these flags are used to control various functions in the processor. The x86 instruction set architecture defines a set of arithmetic status flags and a set of processor control flags. The x86 instruction set architecture also defines and supports exceptions and interrupts, both of which typically result in a transfer of control outside of the currently executing instruction stream. X86 exceptions occur when an unusual or invalid situation is detected during the execution of an instruction. There are two types of x86 exceptions defined, x86 faults and x86 traps. The difference between x86 faults and x86 traps is that an instruction is either aborted (x86 fault) or completed (x86 trap) before the processing of the exception. An interrupt is defined by the x86 instruction set architecture as an event external to the processor. Therefore, interrupts occur asynchronously to the execution of instructions within the processor; i.e., an interrupt has no relation to the specific instruction executing when the interrupt is recognized. X86 exceptions and interrupts are handled at instruction boundaries; that is "in between" two instructions versus within an instruction. Some x86 exceptions and x86 interrupts are given in the following table:

SAMPLE OF x86 FAULT AND TRAP EXCEPTIONS

x86 Exception Name	Type
Divide Error	Fault
Instruction Breakpoint	Fault
Data Breakpoint	Trap
Segment Not Present	Fault
Page Fault	Fault
General Protection	Fault

In particular, note that the x86 instruction set architecture treats an instruction breakpoint as a fault and a data breakpoint as a trap.

It is important to know which execution units can set status flags, which can cause exceptions or traps, and how status flags, faults, traps, and interrupts are treated. *We will always use the preface "x86" to identify an architectural status flag, fault, trap, or interrupt, such as an "x86 trap." Without this modifier, these terms will always be referring to the K6 3D microarchitecture.* Furthermore, we typically will not use the word "exception" but rather use "fault" or "trap" or both, as appropriate. All *x86 faults*, *x86 traps*, and *x86 interrupts* are ultimately handled by microarchitectural fault and trap handling mechanisms. Interrupts will be discussed later in this chapter.

Only integer RegOps can produce status flag values. At the microarchitecture level, the K6 3D has eight status flags—the six x86 instruction set architecture visible flags and two flags for use within sequences of Ops used in the implementation of complex instructions. All integer RegOps that have a defined behavior for status flags also have the option of modifying the status flags.

A fault or trap at the microarchitectural level causes an abort cycle that then leads to the execution of a sequence of Ops which ultimately turns the abort action into an architecturally defined exception. The only execution units that can produce faults are the LU, the SU, and the FPU. In other words, RegOps can never fault: only LdOps, StOps, and FpOps can fault. RegOps can modify status flag values. The microprocessor must support the x87 architecturally defined floating-point flag bits that reside in the architecturally defined floating-point status word register.

Design Note

Microarchitectural Faults and Traps

In addition to supporting all of the x86 architectural exceptions and interrupts, the K6 3D supports some microarchitectural faults and traps which are:

Fault/Trap	Type
Fault Op	Fault
Self-Modifying Code Check	Trap

An abort cycle causes the invocation of a fault handler to determine what caused the fault or trap and act appropriately. Traps are handled somewhat differently than faults. When an x86 trap occurs (such as a data breakpoint trap), information associated with it is loaded into the scheduler and associated with the Op that caused it. Then, when that Op is going to be committed, the Op Commit Unit recognizes that a trap has been detected and sets a pending trap flip-flop. In effect, traps are accumulated as pending traps until the end of an x86 instruction is reached. The Op Commit Unit can recognize that it is retiring the last of all of the Ops associated with a given instruction. If there are any pending traps at the end of the commitment of an instruction, a "fault" is recognized at the beginning of the next instruction which, in turn, causes an abort cycle.

upper and lower portions of the processor

It is useful to think of the microprocessor consisting of an *upper portion* and a *lower portion* when explaining the abort cycle. This is shown in Figure 2.4 where the two portions are shaded differently. When an *abort cycle* is required, the following sequence of actions could occur to complete the Ops in the scheduler that should be completed, with the non completed Ops being discarded:

1. let all Ops older than the Op that initiated the abort cycle commit and retire by allowing them to naturally progress down to the bottom row of the scheduler's buffer. The commitment of this Op and all younger Ops is inhibited.
2. when all older Ops have been committed, the entire machine, i.e., both the upper portion and the lower portion, can then be flushed.
3. after the flushing has been completed, instruction fetching and decoding can begin at the appropriate point.

Design Note

Restarting the Upper Portion of the Processor

Restarting the upper portion of the processor does not affect the operation of the L1 I-Cache, therefore it is not included in the upper portion shown in Figure 2.4 on page 87. Likewise, restarting the lower portion does not affect the L1 D-Cache, so it is not part of the lower portion. If, for example, the L1 I-Cache (or the L1 D-Cache) is processing a "miss," then it is irrevocably committed to the processing of the miss.

This sequence is a direct result of a cost/performance trade-off. It allows for a simplified scheduler design. Several things are happening at the same time that need to be understood to appreciate this statement. On the one hand, the scheduler does not require logic in each entry to determine whether or not the associated Op should be flushed. When the flushing action occurs, all Ops in the scheduler will be marked "invalid." In this design approach, we will learn that a potential latency of one to two cycles in the flush action has been accepted while a simplification in the scheduler circuitry has been achieved.

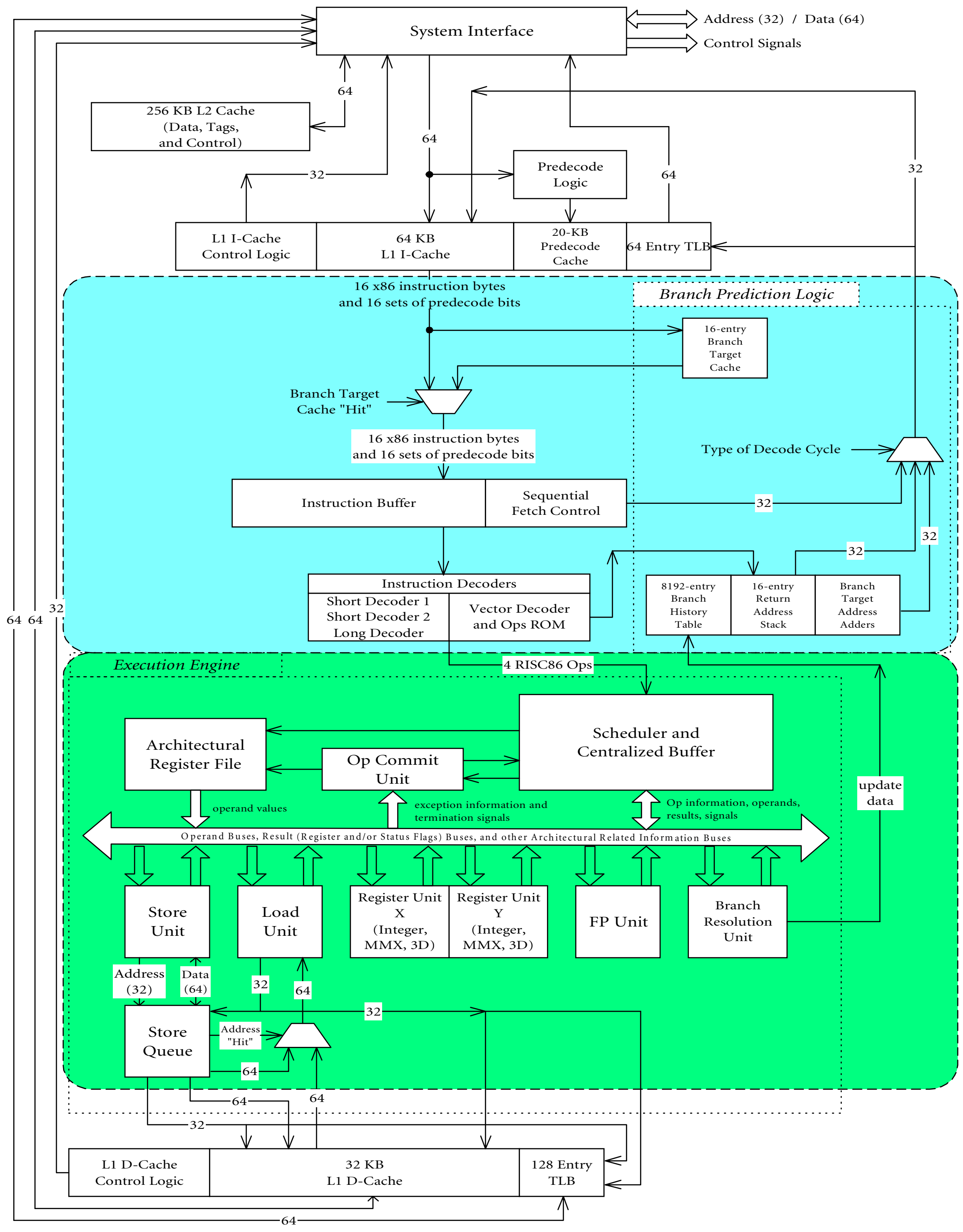

System Interface
Address (32) / Data (64)
Control Signals
256 KB L2 Cache (Data, Tags, and Control)
64
32
64
Predecode Logic
64
32
L1 I-Cache Control Logic
64 KB L1 I-Cache
20-KB Predecode Cache
64 Entry TLB
16 x86 instruction bytes and 16 sets of predecode bits
Branch Prediction Logic
16-entry Branch Target Cache
Branch Target Cache "Hit"
16 x86 instruction bytes and 16 sets of predecode bits
Type of Decode Cycle
Instruction Buffer
Sequential Fetch Control
32
32
32
Instruction Decoders
Short Decoder 1
Short Decoder 2
Long Decoder
Vector Decoder and Ops ROM
8192-entry Branch History Table
16-entry Return Address Stack
Branch Target Address Adders
64 64 32
Execution Engine
4 RISC86 Ops
Architectural Register File
Op Commit Unit
Scheduler and Centralized Buffer
update data
operand values
exception information and termination signals
Op information, operands, results, signals
Operand Buses, Result (Register and/or Status Flags) Buses, and other Architectural Related Information Buses
Store Unit
Load Unit
Register Unit X (Integer, MMX, 3D)
Register Unit Y (Integer, MMX, 3D)
FP Unit
Branch Resolution Unit
Address (32)
Data (64)
32
64
32
Store Queue
Address "Hit"
64
64
64
32
L1 D-Cache Control Logic
32 KB L1 D-Cache
128 Entry TLB
64

Figure 2.4 Upper and Lower Portions of the Processor

Importantly, for a BrOp-related abort, the K6 modifies the above sequence to reduce the flush performance penalty. The upper portion of the machine can be flushed while the older Ops are being completed. If Ops are generated before the lower portion of the machine can accept them (e.g., the lower portion of the processor may be in the process of invalidating the younger Ops in the scheduler), then these Ops are held until they can be consumed (see the OpQuad Buffer in Figure 2.8 on page 127.) The point is that since the fetch, decode, and execution of the Ops in the execution units are decoupled, the process of fetching and decoding the required instructions can be overlapped (done concurrently) with the completion and flushing of the appropriate Ops. More will be said about this in the section titled "Handling Faults, Traps, and Precise Interrupts" beginning on page 175.

Architectural and Microarchitectural Registers

X86 instructions obtain their operands from and place their results in either the architectural registers or main memory. The execution units, in turn, must access these operands and produce the required results.

Definitions

Architectural and Microarchitectural Register Files

An architecture has a set of registers accessible by its instruction set for storing values associated with operand values, status flags, and other architectural state-related information. This set of registers is often called the *architectural register set* or *architectural register file*. The values stored in it at any instant in time are called the *architectural machine state* or *instruction set architecture machine state*. The microarchitecture typically has a different number of registers, most often a larger number, that are used not only to store the architectural machine state but also to store *microarchitectural machine state,* i.e., operand values, status flags, and state information that is used exclusively in the microarchitecture and not visible to the instruction set architecture.

The K6 3D must support all of the registers defined by the x86 instruction set architecture. These registers include:

1. eight 32-bit integer general purpose registers.
2. six 16-bit segment selector registers and associated segment descriptor registers.
3. one 32-bit (EIP) instruction pointer register.

4. x87 floating-point unit registers (a stack of eight 80-bit internal floating-point registers, a 16-bit status word register, a 16-bit control word register, and a 16-bit tag word register).
5. eight 64-bit MMX registers which, from an instruction set architecture perspective, are aliased with the eight FPU stack registers.
6. one 32-bit EFLAGS register.
7. five 32-bit control registers.
8. eight 32-bit debug (breakpoint) registers.
9. memory management registers, namely the x86 Global Descriptor Table Register, the Local Descriptor Table Register, the Interrupt Descriptor Table Register, and the Task Register.[16]

The K6 3D supports the 3D registers associated with the 3D instruction extensions to the x86 instruction set architecture. The 3D registers are conceptually and physically one and the same as the MMX registers identified in the above list. In addition, the K6 3D supports a number of special registers which are described in the section titled "Special Registers and Model Specific Registers" beginning on page 94.

Integer Registers

The K6 has twenty-four 32-bit integer registers in the integer architectural/microarchitectural register file plus it has twenty-four 32-bit integer renaming registers. The twenty-four registers in the integer architectural/microarchitectural register file consist of eight architecture registers that correspond to the x86 32-bit general purpose registers (EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI) and sixteen microarchitecture scratch registers (t0 through t15). The twenty-four renaming registers are located in the scheduler's twenty-four Op entries—one per entry.

The x86's 32-bit integer architectural register set supports addressing, for byte operations, of either of the lower two bytes of half of some, but not all, of the registers. Based on a register size specification, the 3-bit register numbers within x86 instructions are interpreted as either high (H) or low (L) byte registers or as word or double-word registers. The relationship between these interpretations is seen in the following table.

[16] For example, the Global Descriptor Table Register holds the 32-bit base address and 16-bit segment limit for the currently active Global Descriptor Table.

Table 2.3 x86 General Purpose Register Names and Sizes

32-Bit Name (dword)	16-bit Name (word)	8-bit Name (high order byte)	8-bit Name (low order byte)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
EDI	DI	—	—
ESI	SI	—	—
ESP	SP	—	—
EBP	BP	—	—

The integer microarchitecture register set also supports similar addressing of the lower two bytes of half of these scratch registers: registers +1 through +4 and registers +8 through +11. This is similar to the way in which byte addressing is supported in the x86 instruction set architecture registers.

The x87 Floating-Point Registers

As noted above, the x86 floating-point unit has eight 80-bit internal floating-point registers, a 16-bit status word register, a 16-bit control word register, and a 16-bit tag word register. The eight data registers are 80-bits wide registers and comply with the IEEE floating-point standard extended precision format. As such, the x87 instruction set architecture views them as 80-bit registers. The K6 FPU does its own local register renaming for all of its registers.

MMX and 3D Registers

The x86 MMX register set consists of eight 64-bit registers which the MMX instructions access directly using the register names MM0 through MM7. Although the eight MMX registers are defined in the x86 instruction set architecture as separate registers, they are aliased to the eight registers in the FPU data register stack. The MMX registers are mapped onto the lower sixty-four bits of the x87 registers, with the upper sixteen bits defined to effectively be all ones. The 3D instructions, being floating-point analogs of the MMX instructions, also use and share the MMX registers with the MMX instructions. Subsequently, we will call these registers the MMX/3D registers.

There are nine MMX/3D 64-bit architecture/microarchitecture registers and twelve MMX/3D 64-bit renaming registers. The nine architectural/microarchitectural registers consist of eight that correspond to the x86 architecture MMX 64-bit registers (MM0 through MM7) and one microarchitecture scratch 64-bit register (MMt1).

Design Note

Number of MMX Registers

The K6 has six 64-bit MMX renaming registers which reflects its single-pipeline implementation. The K6 3D, which also supports the 3D instruction set extensions to the x86 instruction set architecture, has twelve 64-bit MMX/3D renaming registers, which reflects its dual-pipeline implementation.

A presentation by Lance Smith, giving an overview of a non-3D version of the K6, is entitled, "*The AMD-K6 Processor: Microarchitecture Overview and Product Update*," and can be found on the CD-ROM.

Register Number and Name Mappings

X86 instructions specify general registers via a 3-bit register number. In the microarchitecture, the K6 adds two leading 0's to the x86's 3-bit register number to form a 5-bit internal architecture/microarchitecture register number. Table 2.4 gives the correspondence between these 5-bit numbers and the various integer, MMX/3D, and scratch registers. The interpretation of a register number as either an integer or an MMX/3D register is, obviously, based on the instruction that is accessing the register. The uses of reg, regm, MMreg, and MMregm are explained in Chapter 3.

.

Table 2.4 Register Number/Name Correspondence

Register Number	32-bit Register Name	1-Byte Register Name	64-bit MMX/3D Register Name
00000	EAX	AL	MMreg
00001	ECX	CL	MMreg
00010	EDX	DL	MMreg
00011	EBX	BL	MMreg
00100	ESP	AH	MMregm
00101	EBP	CH	MMregm
00110	ESI	DH	MMregm
00111	EDI	BH	MMregm
01000	t1	t1L	MMt1
01001	t2	t2L	—
01010	t3	t3L	—
01011	t4	t4L	—
01100	t5	t1H	—
01101	t6	t2H	—
01110	t7	t3H	—
01111	t0/_[a]	t4H	—
10000	t8	t8L	—
10001	t9	t9L	—
10010	t10	t10L	—
10011	t11	t11L	—
10100	t12	t8H	—
10101	t13	t9H	—
10110	t14	t10H	—
10111	t15	t11H	—
11000	reg	reg	MM0
11001	reg	reg	MM1

Table 2.4 Register Number/Name Correspondence (Cont)

Register Number	32-bit Register Name	1-Byte Register Name	64-bit MMX/3D Register Name
11010	reg	reg	MM2
11011	reg	reg	MM3
11100	regm	regm	MM4
11101	regm	regm	MM5
11110	regm	regm	MM6
11111	regm	regm	MM7

[a] The “t0” and “_” mnemonics are synonymous. “_” is used when an operand or result value is a “don’t care.” t0 is like the “traditional” RISC R0 register.

Design Note

Register Size Specification

In the section titled “Formats for Decoder Ops” beginning on page 142, we will learn that the register size, from an Op perspective, is specified by either the ASz or DSz field of the Op. ASz is used for base and index registers in LdStOps. DSz is used for the data register in LdStOps and the source operand and result or destination registers in RegOps. The scratch integer register set supports addressing of the lower two bytes of half of these registers: t1-t4 and t8-t11.

architectural/microarchitectural register file

The combination of the integer architectural/microarchitectural register file and the MMX/3D architectural/microarchitectural file will be called thet Architectural Register File and is shown in Figure 2.2 on page 69. The K6 3D’s implicit renaming scheme is discussed in Chapter 3. The microarchitectural and renaming registers just discussed are not the only microarchitectural registers in the K6 3D. There are additional special registers that the scheduler, OCU, and execution units use in many aspects of their work. We will now describe these special registers as some of them are referenced in the pseudo-RTL descriptions that appear in Chapter 3. You may want to skim this section now, but revisit it from time to time when reading Chapter 3.

Historical Comment, Definition and Suggested Readings

Register Renaming

Register mapping is the process of associating specific microarchitectural (physical) registers with specific architectural (virtual) registers. The mapping can be *static* (bound before execution) or *dynamic* (done at execution time). If the process is dynamic, i.e., the "renaming" (re-mapping) occurs during execution, it is called *register renaming*. The mappings must be *complete*, i.e., each architectural register having a valid value must have a corresponding microarchitectural register mapped to it at each point in time when that valid value is associated with the architectural register. Register renaming can be used to remove various types of dependencies (in particular, write-after-read and write-after-write dependencies).

Although register renaming has been used recently by a number of microprocessor vendors including Intel and AMD, its use is not new with microprocessors—see, for example, the 1967 Tomasulo article referenced below. At the same time though, microprocessors did not begin employing this technique until over 20 years after being used in mainframes. According to Peuto:

> *"The MIPS R2000 in 1986 was the first microprocessor to implement a simple pipeline with branch prediction favoring the branch-taken path. This pipeline was adopted for the 80486 in 1989. Register renaming and instruction scheduling, both concepts from the 360/91, were used by Intel for Pentium Pro in 1995, after several RISC micro-processors had already adopted them."*

For a detailed discussion concerning a register renaming scheme, see R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, Vol. 11, January 1967, pp. 25-33. A copy of this article is the companion CD-ROM. See also: Mike Johnson, *Superscalar Microprocessor Design*, Prentice Hall. 1991 and David A. Patterson and John L. Hennessy, *Computer Architecture: A Quantitative Approach*, 2nd Edition, Morgan Kaufmann Publishers, Inc. 1996. A related article, "The Microprocessors Follow Mainframe Path," by Bernard L. Peuto, in *Microprocessor Report*, April 21, 1997, is also on the CD-ROM,

Special Registers and Model Specific Registers

The special registers shown in the RUX pipeline of Figure 2.3 on page 81 are not accessible from x86 instructions. They are accessible only through special RegOps in OpQuad Sequences. What are OpQuads? The instruction decoders, during a decode cycle, always produce a group of four Ops which is called an *OpQuad* (see Figure 2.6 on page 115). What are OpQuad Sequences? A sequence of OpQuads fetched from an on-chip ROM (called the *OpQuad ROM*) is called an *OpQuad Sequence*. Only the hardware decodes of common/simple instructions produce a single OpQuad. OpQuad sequences result from the decode of more complex instructions. OpQuads and OpQuad sequences are discussed in considerably more detail in the section titled "OpQuad Sequences" beginning on page 137.

Video Clip on CD-ROM

	Greg Favor, Principal Architect of the K6 3D, addresses the following question in this video clip, “Why do you translate x86 instructions into RSIC86 Op sequences?”

The special registers are used for a variety of purposes including internal configuration, debugging, and the processing of traps. Although the special registers are not meant for general use, some of them contain information that is made available to BIOS and operating system implementers via reads and writes to what are termed *model-specific registers*.

SUGGESTED READINGS

Model-Specific Registers

Discussions of the K6 3D and K6 model-specific registers and the instructions that access the data in them, RDMSR (Read Model-Specific Register) and WRMSR (Write Model Specific Register), can be found in the *AMD K86 Family BIOS Design Application Note* on the CD-ROM.

scratchpad memory

Some of the special registers reside physically within the RUX, others are external to the RUX in other execution units, other blocks of the machine, or in a special scratchpad memory. In general, when the special registers external to the RUX and in other blocks of the machine are read in an OpQuad Sequence, two reads are required. The first read loads a temporary internal RUX register from the external unit. The second read delivers the data to its destination from this temporary register.

The following codes will be used in the “Access” column in the tables that follow:

INTERPRETATION FOR THE ACCESS COLUMN

Access	Interpretation
R	Read-only special registers
W	Write-only special registers
R/W	Readable and writable special registers
E/W	Write-only special registers; readable copy maintained in scratchpad memory

Additionally, all "Reserved" bits are read as zero and should be written with either a zero for forward compatibility with software use of these bits. Some x86 architectural registers (e.g., some of the CRxx, DRxx, and TRx registers plus other registers such as TR and data segment selector registers) are maintained only in scratchpad memory and are thus R/W scratchpad memory locations. Special registers in scratchpad memory are read using a single LdOP versus one or two RegOPs. Such registers are not included in the tables that follow.

Table 2.5 SR0, General Control and Status Register

Bit	Name	Function	Access
0:1	CPL	Copy of architectural current privilege level (CPL)	R
2	IOS	I/O sensitivity status	R
3	V86	V86 mode = EFlags.VM && CR0.PE	R
4	REAL	Real Mode = !CR0.PE	R
5	EWBE	External write buffer empty	R
6	BusBsy	Indicates if there are any active/asserted internal requests for bus cycles in the processor system bus	R
7	PMSP	POP memory base = SP (from OpQuad Sequence environment)	R
8	FLUSHP	FLUSH# request pending	R
9	SMIP	SMI# request pending	R
10	INITP	INIT request pending	R
11	NMIP	NMI request pending	R
12	INTRP	INTR request pending	R
13	STPCLKP	STPCLK request pending	R
14	VME	Virtual Mode Extension	R/W
15	PVI	Protected Virtual Interrupt	R/W
16	ClrFLUSHP	Clear FLUSH# edge latch	W
17	ClrSMIP	Clear SMI# edge latch	W
18	ClrINITP	Clear INIT edge latch	W
19	ClrNMIP	Clear NMI edge latch	W
20	ClrISTF	Clear INTR/STPCLK# temporary mask flag	W
21	ClrBSNTF	Clear IBrkPt/SMI#/NMI temporary mask flg	W

Table 2.5 SR0, General Control and Status Register (Cont)

Bit	Name	Function	Access
22	NF	NMI mask flag	R/W
23	RIF	Halt instruction fetch	W
24	SMIACT	System management mode active	R/W
25	FERR	Floating-point error pending	R/W
26	StopClk	Allows an OpQuad sequence to stop clock	R/W
27	HaltClk	Allows an OpQuad sequence to stop clock	R/W
28	IGNNE	Ignore CR0.NE	R
29	RBGO	RAM BIST go/initiate	W
30	RBDN	RAM BIST done status	R
31	RBPF	RAM BIST pass/fail status	R

The K6 3D implements various test and debug modes to enable the functional and manufacturing testing of systems and boards that use the processor. In addition, the debug features of the processor allow designers to debug the instruction execution of software components. Some of these test and debug features, which were discussed in Chapter 1, are:

1. built-in self-test (BIST) which is invoked after the falling transition of the x86 RESET signal and runs internal tests that exercise most on-chip RAM and ROM structures, e.g., the L1-Cache, and the TLBs.
2. a tri-state test mode that causes the processor to float its output and bidirectional pins.
3. a boundary-scan test access port (TAP)—which supports the IEEE standard that defines synchronous scanning test methods for complex logic circuits, such as boards containing a processor. The Joint Test Action Group (JTAG) test access function is defined in the *IEEE Standard Test Access Port and Boundary-Scan Architecture*, IEEE 1149.1-1990, IEEE Press.
4. an L1-Cache Inhibit—a feature that disables the processor's internal L1 instruction and data caches.
5. debug support—consists of all x86-compatible software debug features, including the debug extensions.

Boundary-scan testing uses a shift register consisting of the serial interconnection of boundary-scan cells that correspond to each I/O buffer of the processor. This register chain, called a Boundary Scan Register (BSR), can be used to capture the state of every processor pin and to drive every processor output and bidirectional pin to a known state. You will see support for these features in a number of the fields of various special registers, such as the two BIST-related bits in SR0.

The contents of the SR1 register are defined in Table 2.6.

Table 2.6 SR1, Fault Control and Status Register

Bit	Name	Function	Access
2:0	FID	Fault ID from the OCU	R
3	TSA	TS access fault	R
4	ClrDTF	Clear x86 debug trap pending flag	W
5	ClrSSTF	Clear x86 single-step trap pending flag	W
6	FPF	FpOp fault	R
7	EF	OpQuad Sequence fault	R
10:8	IPFI	Instruction page fault information	R
11	DlyPG	Delay new CR0 PG bit effect	W
14:12	DPFI	Data page fault information	R
15	BIM	Burn-In Mode	R
19:16	DBN	x86 Data Break Point debug status	R
23:20	IBN	x86 Instruction Break Points debug status	R
26:24	SubOpcd	Sub-Opcode (MODR/M[5:3] from OpQuad Sequence Environment)	R
28:27	OCPL	Old CPL (from OpQuad Sequence Environment)	R
29	RBD	RAM BIST Disable	R
30	SSTF	x86 single-step trap pending flag	R
31	SDM	Select Direct Mapped	R

The contents of the SR2 Instruction Page Fault Register are defined as follows:

Table 2.7 SR2, Instruction and Page Fault Register

Bit	Name	Function	Access
31:0	—	Logical address of last instruction fetch page fault	R

The contents of the SR3 Data Page Fault Register are defined as:

Table 2.8 SR3, Page Fault Register

Bit	Name	Function	Access
31:0	—	Logical address of last operand page fault	R

The contents of the SR4 Fault PC Register are as follows:

Table 2.9 SR4, Fault PC Register

Bit	Name	Function	Access
31:0	—	Logical address of last operand page fault	R

The contents of the SR5 register follow. The SR5 Configuration Register in RUX is write only. There is a readable shadow copy of it kept in the scratchpad memory. An OpQuad Sequence always updates these copies together to keep them in synchronization. There are additional debug features only available via special debug packages used during "silicon" debug. These are in contrast to the test and debug features mentioned on page 97 which are publicly accessible.

Table 2.10 SR5, Configuration Register (in RUX)

Bit	Name	Function	Access
0	L1ICD	L1 I-Cache disable	E/W
1	L1DCD	L1 D-Cache disable	E/W
2	L1CI	L1 Cache inhibit (TR12.CI)	E/W
3	DE	Debug extension enable (CR4.DE)	E/W
4	PSE	Page size enable (CR4.PSE)	E/W
5	WAD	Write allocate disable	E/W
6	PDD	Power down disable	E/W
7	NPFCD	NP freeze clock disable	E/W
8	SMO	Strong memory order	E/W
9	VSMO	Very strong memory order	E/W
10	SMCD	Self-Modifying Code trap disable	E/W
11	BPTD	Branch Prediction Table disable	E/W
12	BTBD	Branch Target Buffer disable	E/W

Table 2.10 SR5, Configuration Register (in RUX) (Cont)

Bit	Name	Function	Access
13	ROBD	RegOp bumping disable	E/W
14	LCKD	Lock disable	E/W
15	STQFD	STQ forward data disable	E/W
16	DCERLR	D-Cache enable random line replacement	E/W
17	DCSLD	D-Cache speculative load disable	E/W
18	ICERLR	I-Cache enable random line replacement	E/W
19	WBCD	Write back cache disable	E/W
20	SLDD	Speculative load disable	E/W
21	DTBDM	DTB direct mapped	E/W
22	DCDM	D-Cache direct mapped	E/W
23	ICDM	I-Cache direct mapped	E/W
28:24	REGN	Register number	E/W
29	RUYD	RUY disable	E/W
30	BPTNT	When BPTD=1, BPTNT indicates prediction direction; 1 = not taken, 0 = taken	E/W
31	ICPFD	I-Cache prefetch disable	E/W

The copy of the SR13 Instruction Decode Control Register in RUX is write only. There is a read/write shadow copy of it kept in the scratchpad memory. With the exception of setting SetVEC1, an OpQuad Sequence always updates these copies together to keep them in synchronization. Just as with SR5, these are additional debug features used during silicon debug. The contents of the SR13 register are defined as follows:

Table 2.11 SR13, Instruction Decode Control Register

Bit	Name	Function	Access
7:0	SDD	Short decode disable bit mask	E/W
9:8	MDD	Multiple decode disable	E/W
10	LDD	Long decode disable	E/W
11	SetVEC1	Set “force HDD for one decode”	W
12	ExtExcpVEC	External OpQuad Sequence exception group	E/W

Table 2.11 SR13, Instruction Decode Control Register (Cont)

Bit	Name	Function	Access
13	ESCDD	ESC (FPU) decode disable	E/W
14	MMXDD	MMX/3D decode disable	E/W
15	SD2D	OF opcode short decode disable	E/W
31:16	ExtVEC	External OpQuad Sequence decode group	E/W

The SDD and SD2D bits, described in the following design note, are also for silicon debug purposes—bypassing broken hardware and/or patching the OpQuad Sequences for complex instructions.

Design Note

SDD and SD2D Bits

The SDD and SD2D bits are used in conjunction with the predecoding logic discussed in the section titled "Predecoding Logic" beginning on page 115. *You may want to reread this design note after you have studied that section*. The SDD and Sd2D bits are used to prevent the marking of an instruction as being able to be decoded by one of the short decoders. When an instruction is inhibited from being short-decoded, it is either long decoded or, in most cases, vector decoded. When it is vector decoded, the vector decoder provides an OpQuad Sequence entry point which, by default, is located in the on-chip OpQuad ROM but may be forced to come from the off-chip memory. When the SDD and SD2D bits are changed, the I-Cache and predecode cache should be flushed. Each of the eight SDD bits control two rows of the one-byte x86 instruction opcode map. Thus, SDD[0] controls opcode rows 0 and 1, SDD[1] controls opcode rows 2 and 3, and so on. The SD2D bit controls all opcodes in the two-byte x86 instruction opcode map. Each of the 16 bits in ExtVEC controls two rows in the x86 instruction opcode map. Specifically, the low-order eight bits control the rows of the one-byte x86 instruction opcode map, and the high-order eight bits control the rows of the two-byte opcode map. To have opcodes handled by external OpQuad Sequence entry points, both the appropriate SDD/SD2D bit(s) and the appropriate ExtVEC bit(s) have to be set. An OpQuad Sequence should not update SR13 in the scratchpad memory when the SETVEC1 bit is set, since that bit is not "sticky" in the special registers internal to RUX.

The contents of the SR16 MMX/3D Status Bits Register are defined as follows:

Table 2.12 SR16, MMX/3D Status Bits Register

Bit	Name	Function	Access
7:0	MMXD	MMX/3D data register dirty bits	R
8	MMXSTC	MMX/3D store instruction committed	R
31:9	Reserved	—	—

Reading SR16 clears both SR15.MMXD[7:0] and SR15.MMXSTC. Registers SR17 and SR18 are associated with the Time Stamp Counter (TSC) Bits; SR17 with TSC High and SR18 with TSC Low as seen in Table 2.13.

Table 2.13 SR17 and SR18, Time Stamp Control Registers

Bit	Name	Function	Access
31:0	TSCL	Must synchronize read and write with TSCH to avoid overflow to TSCH.	R/W
63:32	TSCH	Must synchronize read and write with TSCL to avoid overflow from TSCL.	R/W

The copy of the SR21 Configuration Register RUX is write only. There is a read/write shadow copy of it kept in the scratchpad memory. An OpQuad Sequence always updates these copies together to keep them in synchronization. The contents of the SR21 register are defined as follows:

Table 2.14 SR21, Configuration Register (in RUX)

Bit	Name	Function	Access
0	Reserved	—	—
1	NAD	NA# Disable	E/W
2	SIE	Stop Interrupt Enable	E/W
3	FEEC	Force External OpQuad Sequence Cacheable	E/W
4	SSD	String SMI Disable	E/W
14:5	Reserved	—	—
15	INVC	INValidate Caches	E/W
16	WAE15M	Write Allocate Enable 15M-16M	E/W

Table 2.14 SR21, Configuration Register (in RUX) (Cont)

Bit	Name	Function	Access
23:17	WAELIM[6:0]	Write Allocate Enable Limit	E/W
24	PDED	Predecode Cache Disable	E/W
31:25	Reserved	—	—

Design Note

Upper Limit of Memory

The WAELIM[6:0] defines the upper limit of memory where write allocates are allowed. This is done in 4M quantities as follows: Lower-Limit = 0x0, UpperLimit = WAELIM[6:0] * 4M (max = 508 Mbyte). Excluded areas are: 640K - 1M and, if WAE15M = 0, 15M -16M.

The contents of the SR24 NP Presence and Opcode Register are defined as follows:

Table 2.15 SR24, NP Presence and Opcode Register

Bit	Name	Function	Access
10:0	FpOpcd	NP opcode register	R/W
30:11	Reserved	—	—
31	NPNotPres	NP Not Present	R

The contents of the SR25 NP Code Selector Register are defined as follows:

Table 2.16 SR25, NP Code Selector Register

Bit	Name	Function	Access
15:0	FpOpcdSelNP	Code pointer (selector part)	R/W
27:16	Reserved	—	—
31:28	PrfxCnt	Prefix count from the OpQuad Sequence execution environment	R

The contents of the SR26 NP Code Offset Register are defined as follows:

Table 2.17 SR26, NP Code Offset Register

Bit	Name	Function	Access
31:0	FpCodeOffs	NP code pointer (offset part)	R/W

The contents of the SR27 NP Data Selector Register are defined as follows:

Table 2.18 SR27, NP Data Selector Register

Bit	Name	Function	Access
15:0	FpDataSel	NP data pointer (selector part)	R/W
31:16	Reserved	—	—

The contents of the SR28 NP Data Offset Register are defined as follows:

Table 2.19 SR28, NP Data Offset Register

Bit	Name	Function	Access
31:0	FpDateOffs	NP data pointer (offset part)	R/W

The contents of the SR29 NpCFG (FPU Configuration) Register are defined as follows:

Table 2.20 SR29, NPCFG (FPU Configuration) Register

Bit	Name	Function	Access
0	ClearBeforeEx-ception	Clears the Before Exception (stack fix up) bit in NP	E/W
1	Do Shared State	Allows "share-state" overlapping of p-ops which write their result one cycle after execution, with the following p-op (assuming dependencies are met).	E/W
2	Enable Hyper-Flg	Enables NPPop[12] (hyper flag). When not set, has the effect of always asserting NPPop[12].	E/W
3	StoreExMode	Used when issuing a "dummy" store (i.e., emulating an FST which does not store to memory). Specifically, inhibits hardware checking of result precision and rounding.	E/W
5:4	Reserved	—	—
6	FastFXCH	Enables single-pop FXCH mode	E/W

Table 2.20 SR29, NPCFG (FPU Configuration) Register (Cont)

Bit	Name	Function	Access
7	false_depend-ency_suppress	Enhancement to the share-state mechanism which eliminates false dependencies. See bit 1 below.	E/W
8	Disable0Cycle	Disables handling of 0-cycle Ops in NP	E/W
13:9	Reserved	—	—
14	mask_hyperterm	Disables hyper termination for the next p-op only. This bit then resets itself (internal to NP).	E/W
15	Reserved	—	E/W
16	Busy	Force pending error (i.e., always hyper terminate)	E/W
21:17	Reserved	—	E/W
22	UpperTSC-Word	Upper 16 bits of 32-bit Tag/Status/Control store will be filled with inverse value of this bit	E/W
31:23	Reserved	—	E/W

Branch Direction Prediction Logic and the Branch Resolving Unit

Branches in x86 code fit into two categories: unconditional branches, which always change program flow (that is, the branches are always taken), and conditional branches, which may or may not divert program flow (that is, the branches are taken or not-taken based on the evaluation of a specified condition).

branch direction prediction logic

Because x86 programs are heavily saturated with conditional branches, branch direction prediction logic is used to avoid execution penalties associated with such changes in program flow. Up to 10% of typical application code consists of unconditional branches and another 10% to 20% conditional branches. The K6 branch direction prediction logic has been designed to handle this type of program behavior and to reduce its negative effects on instruction execution, such as stalls due to delayed instruction fetching and the draining of the processor pipeline.

branch target address

The K6 handles unconditional branches by redirecting instruction fetching to the target address of the unconditional branch. Branch target addresses are calculated on-the-fly using fast adders during the decode stage (see the Branch Target Address Adders in Figure 2.2 on page 69). The adders calculate all possible target addresses before the instructions are fully decoded and the branch prediction logic then chooses the correct branch target address.

branch history table
BHT

The branch target address for a conditional branch is not immediately known, however, and the K6 uses a dynamic branch direction prediction mechanism to predict the direction of the branch. Target address handling is similar for both conditional branches (and unconditional PC-relative branches). That is, branch target addresses are calculated on-the-fly using fast adders during the decode stage. As in the unconditional branch case, the adders calculate all possible target addresses in parallel with the determination and selection of the correct target address. The processor then chooses either the predicted target address or the sequential next instruction address adders, based on a prediction of the direction of the branch, as the next address to continue instruction decoding from. The K6's dynamic branch direction prediction logic uses a two-level, adaptive, branch direction prediction algorithm based on the contents of an 8192-entry branch history table (BHT). The BHT is only used to predict the direction of a conditional branch. It stores information about the direction of past conditional branches. It does not store predicted target addresses, nor does it include information about unconditional branches.

branch target cache
BTC

A branch target cache (BTC) is used for both conditional and unconditional PC-relative branches. If a conditional branch is predicted to be not taken, then the processor simply continues decoding and executing the next sequential x86 instruction. When a conditional branch is predicted to be taken or the branch is unconditional, the BTC supplies the first 16 bytes of target instructions directly to the instruction buffer. Assuming the target address hits in this cache, this design approach can avoid a 1-clock decode delay while the first or target I-Cache fetch takes place.[17] The BTC is organized as sixteen entries of sixteen bytes each. Thus, the BTC works with the BHT and delivers instruction bytes directly to the decoders to avoid the otherwise one-clock decode delay for taken branches. The BHT direction prediction rate is estimated to be greater than 95%. The BTC hit rate ranges from 40-60%. In total, the branch prediction logic achieves a predicted branch prediction rate of greater than 95%.

return address stack
RAS

The K6's branch direction prediction logic also employs a 16-entry Return Address Stack (RAS) to minimize fetch and decode stalls associated with subroutine entry (CALL) and exit (RET) instructions. The RAS is specifically designed to optimize subroutine call/return instruction pairs by caching the return address of each call instruction and supplying it as the predicted target address of the corresponding return instruction.

---

[17] Notice the "BTC Hit" control of the instruction multiplexer in front of the instruction buffer in Figure 2.6 on page 115. We will, for the current time, take this control to mean:

```
IF (the instruction being decoded is a branch)
   AND (the branch is predicted taken)
   AND (there was a "hit" in the BTC for the target address)
THEN (select the output of the BTC)
ELSE (select the output of the I-Cache and Predecode Cache)
```

Software is typically constructed using subroutines that are invoked from various places in a program. This is usually done to save space. The subroutine is entered with the execution of a CALL instruction. At that time, among other things, the processor pushes the address of the next sequential instruction following the CALL instruction onto the RAS as well as onto the architectural stack in memory. When the processor encounters a RET instruction within or at the end of the subroutine, the branch prediction logic pops the return address from RAS, as well as reading from the stack in memory (for later prediction checking purposes), and speculatively begins fetching from that location.

branch resolving unit BRU

The Branch Resolving Unit BRU) is separate from the branch direction prediction logic shown in Figure 2.2 on page 69 and enables efficient speculative instruction execution. The BRU gives the processor the ability to execute instructions beyond conditional branches before knowing whether the branch prediction was correct. To accomplish this, the K6 processor does not commit the results of the speculative executed instructions until all preceding conditional branch instructions have been resolved by the BRU. Once the status flag values for evaluating a branch condition are valid, the BRU resolves the conditional branch as either correctly or incorrectly predicted.

If the prediction was incorrect, the processor discards the speculatively executed operations to the point of the mispredicted branch instruction and restores the machine state to that point; execution then continues down the correct branch path.

If the prediction was correct, the BRCOND Op of the branch instruction is so marked and the result of this and the following instructions are allowed to be committed. There are obviously no instruction execution delays in this case. The BRCOND OP represents the branch condition or condition code to be evaluated by the BRU. Equivalently, the BRCOND Op represents the branch condition evaluation operation to be executed by the BRU.

Suggested Readings

Branch Prediction

Because of the importance of branch prediction in achieving high performance, there has been, and continues to be, substantial work in this area. Some articles which, in part, review alternative approaches and are, therefore, of general interest are:

1. Y. N. Patt, W. M. Hwu, and M. Shebanow, "HPS, a New Microarchitecture: Rationale and Introduction," *Proceedings of the 18th Annual Workshop on Microprogramming* (Micro-18), 1985, pp. 103-108.
2. Y. N. Patt, S. V. Melvin, W. M. Hwu, "Critical Issues Regarding HPS, A High Performance Microarchitecture", *Proceedings of the 18th Annual Workshop on Microprogramming* (Micro-18), 1985, pp. 109-116.
3. T. Y. Yeh and Y. N. Patt, "Two-Level Adaptive Training Branch Prediction," *Proceedings of the 24th Annual International Symposium on Computer Architecture* (ISCA), 1991, pp. 124-134.
4. J. E. Smith, "A Study of Branch Prediction Strategies," *Proceedings of the 8th Symposium on Computer Architecture*, 1981, pp. 135-148.
5. D. R. Kaeli and P. G. Emma, "Improving the Accuracy of History-Based Branch Prediction," *IEEE Transactions on Computers*, April 1997.
6. J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, Vol. 17, No. 1, 1984, pp. 6-22.
7. P. K. Dubey and M. J. Flynn, "Branch Strategies: Modeling and Optimization, *IEEE Transactions on Computers*, Vol. 40, No. 10, 1991, pp. 1159-1167.
8. T. Y. Yeh and Y. N. Patt, "Branch History Table Indexing to Prevent Pipeline Bubbles in Wide-Issue Superscalar Processors," *Proceedings of the 26th Annual International Symposium on Microarchitecture*, (Micro-26), 1993, pp. 164-175.

You can find the full text versions of the first five articles on the companion CD-ROM.

The L1 and L2 Caches (Revisited)

The K6 3D on-chip L1 I-Cache has a subblock organization as shown in Table 2.21. Each 64-byte line is configured as two 32-byte subblocks. The two subblocks share a common tag, but have separate pairs of cache coherency protocol bits that are used to track the state of each cache subblock. There is some variability in cache-related terminology applied to specific microprocessors. See, for example, Harvey Cragon's book, *Memory Systems and Pipeline Processors*, Jones and Bartlett Publishers, 1996, where he contrasts some of these differences. The line and subblock organization of the L1-Cache is shown in Table 2.21.

Table 2.21 LINE/SUBBLOCK L1 I-CACHE ORGANIZATION

Tag	Byte 31	Byte 30	…	Byte 1	Byte 0	MESI Bits	Subblock 0
Address	Byte 31	Byte 30	…	Byte 1	Byte 0	MESI Bits	Subblock 1

Suggested Readings

Cache Design

Proper cache design is absolutely central to achieving high performance in systems. Because of this, we have included five articles on the CD-ROM that examine cache-related design, implementation, and performance issues from a variety of perspectives.

1. A. J. Smith, "Cache Memory Design: An Evolving Art," *IEEE Spectrum*, 1987.
2. M. Cekleov and M. Dubois, "Virtual-Address Caches," IEEE Micro, 1997.
3. S. P. VanderWiel and D. J. Lilja, "When Caches Aren't Enough: Data Prefetching Techniques," *Computer*, July 1997.
4. E. van der Deijl, G. Kanbier, O. Temam, and E. D. Granston, "A Cache Visualization Tool," Com*puter*, July 1997.
5. D. Burger, J. R.Goodman and A. Kägi, "Limited Bandwidth to Affect Processor Design," *IEEE Micro*, November/December 1997.

An article which describes several memory consistency models and their relationship to performance, programmability, and portability is given in the article by Sarita V. Adve and Kourosh Gharachorloo, "Shared Memory Consistency Models: A Tutorial," in *Computer*, December 1996

You can find the full text versions of the above five articles on caches and the Adve article on the companion CD-ROM.

The K6 uses the MESI cache coherency protocol. K6 3D I-Cache lines have only two coherency states, valid and invalid, rather than the full four MESI coherency states of the D-Cache lines. This is a result of the fact that the K6's I-Cache lines are read-only.

.

Definition and Suggested Reading

MESI Protocol

MESI = Modified, Exclusive, Shared, Invalid. MESI is a four state cache-coherency protocol that is used in multiprocessor systems in which each processor has one or more caches associated with it and cache consistency must be maintained across these caches. See the article by J. Gallant, "Protocols Keep Data Consistent" *EDN*, Vol. 36, No. 5, March 1991, pp.42-50 and the *International Standard ISO/IEC, ANSI/IEEE Std. 896.I*, 1994 Edition, IEEE, New York, 1994.

cache line replacement
cache subblock replacement

Two forms of cache misses and associated cache fills can take place in a cache organized this way—a cache line replacement[18] and a cache subblock replacement. In the case of a cache line replacement, the miss is due to a tag mismatch. In this case, a new cache line is allocated, the required 32-byte subblock is filled from the L2-cache or external memory, and the other 32-byte subblock within the cache line is marked as invalid. In the case of a cache subblock replacement, the tag matches but the requested subblock is marked as invalid. The required 32-byte subblock is filled from the L2-cache or external memory, and the other subblock within the cache line remains in its current state, i.e., its cache coherency bits are not changed. In either case, L1 I-Cache fills are done on a subblock basis.

[18] Cache line replacements are also referred to as full-line cache misses.

Design Note

L1-Cache Design

The K6 L1-Caches are 32-Kbyte in size and organized as two-way set-associative. This means that the set index is comprised of bits [13:6] of the address. Since the x86 architecture uses linear to physical translation, the physical address is available later than the linear. For timing and performance reasons, the set index needs to use the untranslated bits to start the cache access in parallel to the linear-to-physical translation. In the x86 architecture, only bits[31:12] are translated. Bits[11:0] are identical in linear and physical address spaces. Because of this the K6 3D uses bits [13:6] as the set index. Both ways of a set are physically tagged with the full physical page address (bits [31:12]). Since bits [13:12] are linear, this means that a particular physical cache line can reside in one of a group of 4-sets in the cache. This creates a potential synonym (or aliasing) problem, since a cache lookup indexes into only one of the four possible sets. What happens if the line in question resides in the other three possible sets? The K6 3D deals with this in hardware by avoiding the creation of synonyms in the first place. The tag RAMs of the L1-Caches are designed in a manner that allows all four possible sets to be read out simultaneously. If the line is found in one of the 4-sets, but it is not in the indexed set, it will be invalidated (possibly written back if dirty) and then refetched from the L2 or external memory and put into the indexed set.

The K6 implements a hit-under-miss scheme for both the instruction and data caches. The instruction cache can continue to supply hit data while processing a miss. There are no restrictions. If a new cache access misses in the cache and there is already a pending miss, then the new cache miss is held up until the first miss completes. The L1 D-Cache is similar in its behavior in that it can continue to supply hit data while processing a fill. The only restriction is that the read operation that initiated the fill must have received its data before subsequent read operations can access the cache.

The above behavior provides sufficient performance with a reasonable effort in logic design and implementation. There are more aggressive techniques that are completely nonblocking or nonblocking to a certain depth. For example, if a read operation misses in the L1 D-Cache, the read is allowed to complete without its data (allowing a younger read to advance into the cache lookup stage). The miss can be queued up in a miss queue. The read that completed without its data needs to receive it at a later time when the fill is ready to provide it. Multiple misses will result in more of these types of reads getting completed and queued in the miss queue. While this is all “doable,” it creates complicated situations that are more difficult to deal with. The additional effort was not considered worth the increase in performance.

In general many of the microarchitectural performance trade-offs were taken based on a K6 3D trace-driven performance model that was accurate within a few percentage points to the K6 3D’s actual behavior. If a particular feature provided a large enough percentage increase in performance, it was carefully considered and usually adopted.

In summary, both the I-Cache and the D-Cache are linearly indexed and physically tagged. Synonyms and aliasing are handled in hardware. At most one synonym at a time is allowed in the caches. Both caches maintain mutual exclusion with respect to each other and, as will be seen, this eases the way in which self-modifying code is handled. The hit-under-miss cache-fill strategy is supported. The L2-Cache on the other hand is physically indexed and physically tagged and thus is not concerned with synonyms and aliasing.

cache prefetching

The K6 performs cache prefetching for L1 I-Cache cache-line replacements. Cache prefetching results in the filling of the required 32-byte subblock first, and a prefetch of the second subblock. However, the prefetch of the 32-byte subblock that is not required is initiated in the forward direction only—that is, the second 32-byte subblock is fetched only if the requested subblock is the first subblock within the cache line. From the perspective of the external bus, the two subblock fills typically appear as two 32-byte burst read cycles occurring back-to-back or, if allowed, as pipelined bus cycles. The K6 3D prefetches both L1 I-Cache and L1 D-Cache subblocks.

The sizes and associativities of the K6 3D's TLBs were selected based on academic studies as well as professional papers. Later, the sizes were looked at in the K6 performance model to guarantee that the choices were appropriate. Larger TLBs, with greater associativity, are generally important in Windows 3.1, Windows 95, Windows NT, and Unix-like environments. The environments that are less stressful are older 16-bit DOS and Windows code and Spec benchmarks.

Suggested Readings

Processor Memory Mismatch Problem

An interesting article by Michael K. Milligan and Harvey G. Cragon, "Processor Implementation Using Queues," *IEEE Micro*, 1995, discusses the evolution of instruction and branch target queues and their relationship to interleaved memory and caches. The article also discusses the use of queues to support variable-length instructions and reduce misalignment problems. The article by Wen-mei Hwu and Thomas M.Conte, "The Susceptibility of Programs to Context Switching," shows the importance of analytical modeling and simulation in cache-related studies.

The L2-Cache is 4-way set associative, with a total of 1K sets. Each set has 4 ways: Way 0, Way 1, Way 2, and Way 3. Each way contains one 64-byte line. Each line has two 32-byte subblocks. Thus, the overall L2 Cache size is 4*1K lines = 4*1K*64bytes/line = 256K Bytes. As noted earlier, the L2-Cache is physically indexed and physically tagged. Bits [15:6] of the physical address determine the set number. The starting byte location within a way is determined by bits [5:0] of the physical address. The L2-Cache uses true LRU replacement within a set. An L2 instruction I-TLB was added to the K6 3D to help out on the instruction TLB misses given the small size of the L1 I-TLB.

The Instruction Buffer and Instruction Registers

The K6 can fetch up to sixteen x86 instruction bytes per clock from either the on-chip L1 I-Cache or the Branch Target Cache as shown in Figure 2.6 on page 115. The fetched x86 instruction bytes along with their corresponding predecode bits are placed into a 16-byte instruction buffer which feeds two instruction registers that supply the decoders. Instruction Register 1 supplies the vector decoder, the long decoder, and short decoder 1, while Instruction Register 2 supplies short decoder 2.

instruction buffer
instruction register 1
instruction register 2

An instruction fetch retrieves sixteen bytes, 4-byte aligned, and all within one 32-byte cache subblock. In the case of branch target fetches after a BTC miss, generally 13-16 useful instruction bytes are retrieved, depending on the byte offset of the target address within the first 4-byte word. This is true except when near the end of a cache subblock.

New instruction bytes are loaded into the instruction buffer as preceding instruction bytes are consumed by the decoders. Instructions are loaded and replaced in the instruction buffer with 4-byte granularity. However, instructions can be consumed from the instruction buffer with byte granularity. This means that the loading and reloading of bytes into the instruction buffer is controlled with 4-byte granularity. This simplifies control logic and eases certain speed-critical logic paths. When a control transfer occurs, the entire instruction buffer is flushed and reloaded with a new set of sixteen instruction bytes.

The K6's decode logic is designed to decode up to two x86 instructions per clock. The decode logic accepts x86 instruction bytes and predecode bits from the instruction buffer, locates the actual instruction boundaries, and generates Ops from the x86 instructions. The Ops are then loaded into a centralized scheduler that controls and tracks all aspects of Op issue, execution, and commitment.

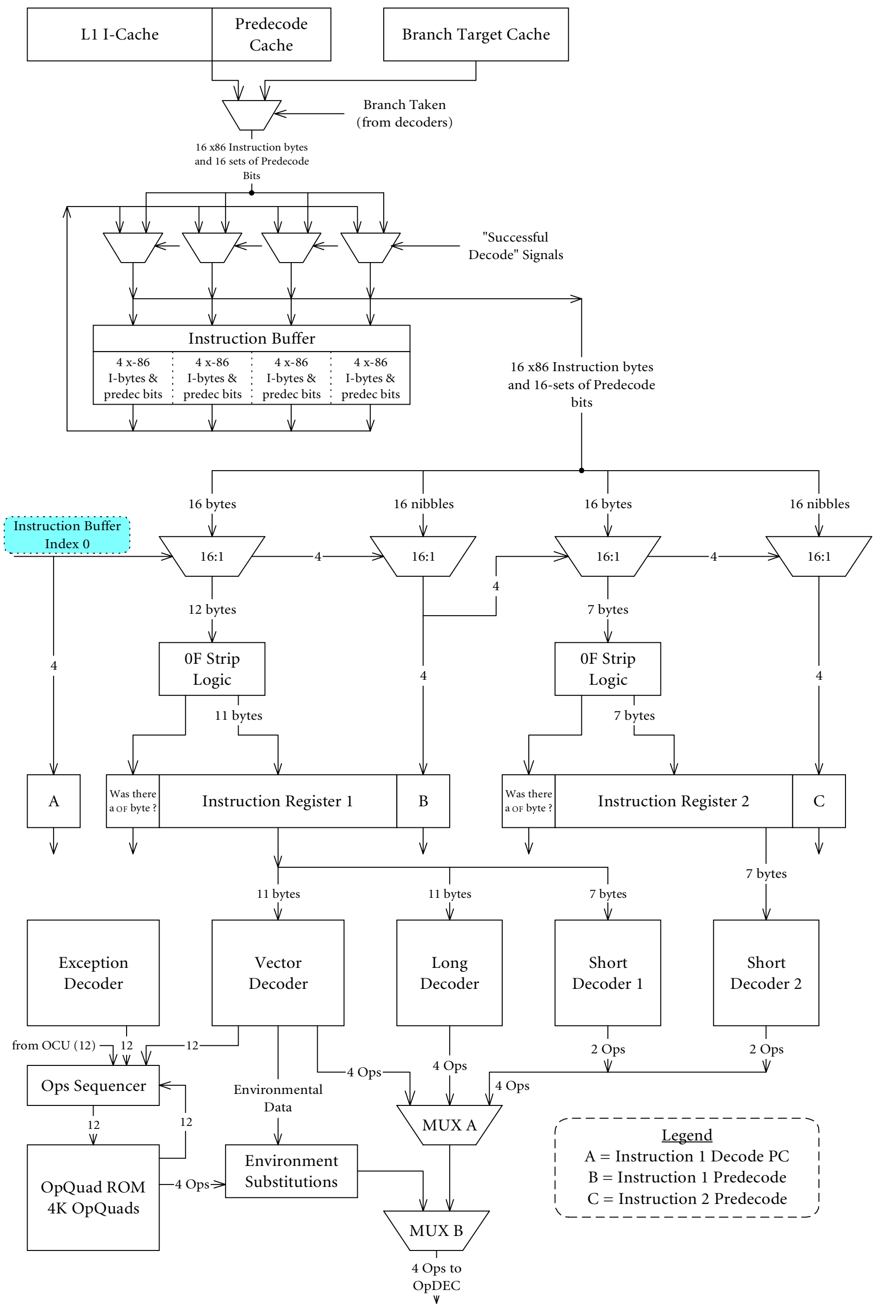


Figure 2.5 Instruction Buffer, Instruction Registers 1 & 2, and the Decoders

NOTATION

Shading in Figures

Elements of figures that are shaded, such as the “Instruction Buffer Index 0” in the above figure, are shown in color on the companion CD-ROM. Both notations are used to indicate that the particular element is expanded in a following diagram. For example, the signal “Instruction Buffer Index 0,” shown in the shaded box in Figure 2.6, is actually the output of a 5-to-1 multiplexer, shown in Figure 2.6.

.

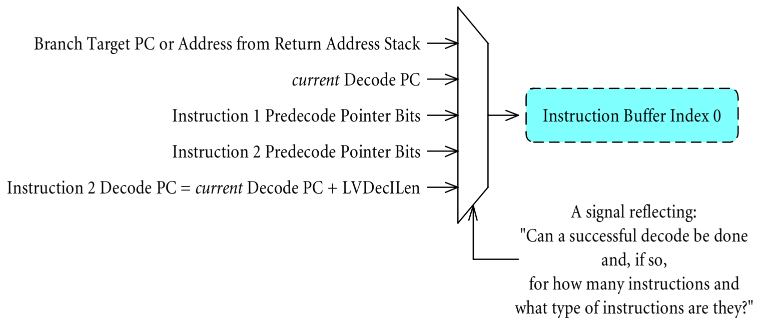

Branch Target PC or Address from Return Address Stack
current Decode PC
Instruction 1 Predecode Pointer Bits
Instruction 2 Predecode Pointer Bits
Instruction 2 Decode PC = *current* Decode PC + LVDecILen
Instruction Buffer Index 0
A signal reflecting: "Can a successful decode be done and, if so, for how many instructions and what type of instructions are they?"

Figure 2.6 INSTRUCTION BUFFER INDEX 0 MULTIPLEXER

Some of the inputs and the control of this multiplexer use information developed by the predecode logic shown in the high-level block diagram of the K6 3D microarchitecture shown in Figure 2.2 on page 69. The predecode logic is discussed in the very next section. This discussion includes an explanation of both Figure 2.6 and Figure 2.6.

PREDECODING LOGIC

As is well documented in the literature, decoding x86 instructions is particularly difficult; see Mike Johnson’s book *Superscaler Microprocessor Design,* Prentice-Hall 1991. X86 instructions are variable-length (from one byte to fifteen bytes long) and can have complex addressing modes. They can be modified by one or more prefix bytes, which can appear before any instruction and can affect the instruction’s execution. Instruction can have a variable-size *displacement field* of zero, one, two, or four displacement bytes. Instructions can also have a variable-size *immediate field* of zero, one, two, or four immediate bytes. The displacement and immediate fields are both optional and independent—that is, either one or both may be present and they may be of different sizes within an instruction. A number of fields in the first few bytes of x86 instructions are used to indicate whether or not other fields are present. All of this contributes to the difficulty in determining the length of the current instruction which in turn contributes to the difficulty in determining where the next x86 instruction actually begins relative to the current instruction.

The difficulties in decoding instructions and determining their lengths notwithstanding, two primary goals in the design that have a direct impact on performance are: (1) to be able to do multiple decodes per cycle and (2) to make the cycle time as short as possible. The K6 supports the decode of more than one x86 instruction per cycle and employs a predecoding technique to assist in this process. In essence, predecoding annotates each instruction byte with information that later enables the decoders to quickly locate the next instruction boundary and thus to efficiently decode multiple x86 instructions simultaneously. The predecode logic computes five *predecode bits* associated with each instruction byte. This is shown in Figure 2.7.

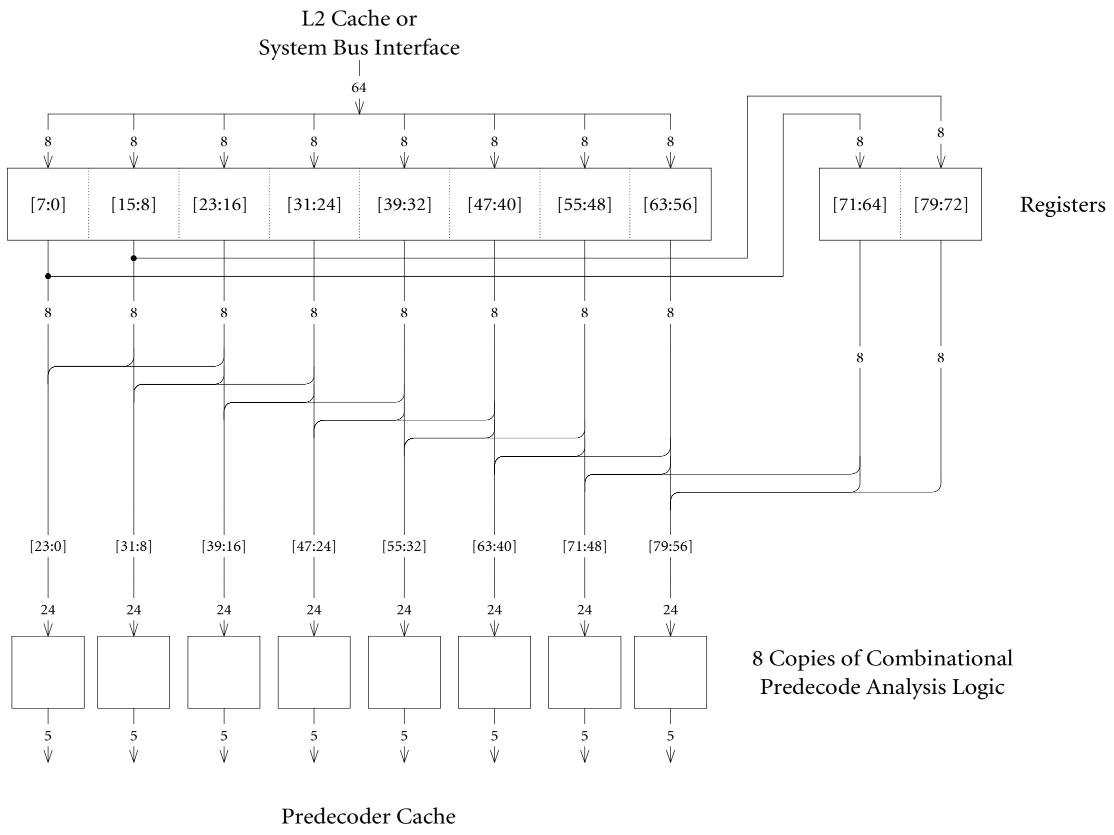


Figure 2.7 PREDECODER LOGIC

Predecode Bits

Predecode Cache

The predecode bits provide a pointer to the first byte of the next instruction. The predecode bits are computed by determining the instruction length and adding this to the low-order PC bits of the predecoded instruction. These bits are stored in a 20-Kbyte predecode cache, separate from the K6's L1 I-Cache. There are five predecode bits per instruction byte and thus the 20-Kbyte for the 32-Kbyte L1 I-Cache. L1 I-Cache lines are filled from main memory or from the L2-Cache using a burst transaction. Each instruction byte is analyzed using the predecode logic on a byte-by-byte basis as an L1 I-Cache line is filled. The analysis consists of assuming the byte under consideration is the beginning of an instruction and then determining the next instruction boundary based on this assumption. Note that the predecode analysis logic in Figure 2.7 can use up to three adjacent instruction bytes in its analysis.

The predecode logic produces six predecode bits per instruction byte. One of the six does not need to be stored in the predecode cache as it can be quickly and readily generated as data are read from the predecode cache. Of the remaining five bits:

1. three bits represent the instruction length in the form of a pointer to the first byte of the next instruction (a fourth bit having been discarded).
2. one bit indicates if the instruction length is D-bit dependent.
3. one bit indicates if the instruction is a Mod R/M instruction or not (this bit is only valid for short or long decodable instructions).

The three bits representing the instruction length are, essentially, the low three bits of the linear or physical program counter (PC) of the next instruction. The fourth and most significant bit (MSB) of the pointer is actually generated when the predecode bits are used. This means, in reality, that although five predecode bits are stored in the predecode cache, six predecode bits are used during instruction decode.

unsuccessful predecode

If for any reason the predecoder cannot compute the above five bits, it will set the three bits of pointer information to the current instruction. This means, effectively, that the computed instruction length is zero and is, essentially, an *unsuccessful predecode* indication to the decoders.

Combinational Predecode Analysis Logic

Deciding if an instruction can be decoded by one of the short decoders is done by comparing the lower bits of the instruction's PC (program counter) with the pointer produced by the predecode logic—i.e., the three bits pointing to the first byte of the next instruction as described in the next section. If they are different, the instruction has a length of one to seven bytes (as implied by the difference between these two pointers) and can be decoded by one of the short decoders. If they are equal, implying an instruction length of zero, the instruction cannot be decoded by one of the short decoders and must be decoded by either the long decoder or the vector decoder as appropriate.

Comparative Analysis and Suggested Readings

Predecode Logic

In an interesting article, "Superscalar Instruction Issue," *IEEE Micro*, September/October 1997, by Dezso Sima, he notes:

> *"Decoding in superscalar processors is a considerably more complex task than in the case of scalar processors and becomes even more so as the issue rate increases. Higher issue rates, however, can unduly lengthen the decoding cycle or can give rise to multiple decoding cycles unless decoding is enhanced. An increasingly common method of enhancement is predecoding. This partial decoding takes place in advance of common decoding, in which instructions are loaded into the instruction cache. The majority of the latest processors use predecoding: the PowerPC 620, PA 7200, PA 8000, UltraSparc, and R10000."*

An audio clip, giving an overview of the predecoding techniques used in the K6 3D, can be found on the companion CD-ROM.

Sima points out that a number of vendors use predecoding in their microprocessor implementations for somewhat different reasons. However, there appears to be conflicting reports about Intel's use of predecoding. Consider the following analysis found in "Intel's Long-Awaited P55C Disclosed," by Michael Slater, *Microprocessor Report*, Vol. 10, No. 14, February 28, 1996, pp. 1-3:

> *"In the P54C, a tag bit is added to each byte as it is stored in the instruction cache to identify instruction boundaries. The P54C's decoder depends on this bit to feed the two instruction pipelines in a single cycle. The P55C's extra cycle allows instructions to be paired on the fly, eliminating the need for the cache predecode bits and allowing instructions to be paired even on an instruction-cache miss."*

continued on the next page.

Comparative Analysis and Suggested Readings (cont.)

Now contrast this with the following description found in "Centaur Gallops Into x86 Market," by Linley Gwennap, *Microprocessor Report*, Vol. 11, No. 7, June 2, 1997, pp. 1-6:

> *"These caches are simpler than in other competitive chips. For example, virtually all other Pentium-class processors have a dual-ported data cache; to match its scalar core, the C6 has a single-ported data cache, reducing die area. The C6 also has no predecode bits in the instruction cache, a feature found in Pentium (but not Pentium/MMX) and AMD's K5 and K6. These extra bits cause the caches on these chips to consume more die area than the same amount of cache on the Centaur chip."*

So far, the following analysis of the P6 (a.k.a. the PPro) suggests that it does not use predecoding, "Intel's P6 Uses Decoupled Superscalar Design," by Linley Gwennap, *Microprocessor Report*, Vol. 9, No. 2, February 16, 1995, pp. 1-7:

> *"Part of the problem in a superscalar x86 processor is identifying the starting point of the second and subsequent instructions in a group. The K5 includes predecode information in its instruction cache to hasten this process, but the P6 does not, to avoid both instruction-cache bloat and the bottleneck of predecoding instructions as they are read from the L2 cache."*

Predecoding is also used to assist in the attempts to obtain increased code compaction. Here the predecoders are used, in part, to expand compressed code. See, for example, "Embedded Vendors Seek Differentiation: Signal Processing, Code Compression, ASIC Cores Enable Specialization," by Jim Turley, *Microprocessor Report*, Vol. 11, No. 1, January 27, 1997, pp. 1-6:

> *"Following on the heels of ARM's Thumb, MIPS introduced MIPS-16, a similar approach to compressing 32-bit instructions into 16-bit words. Thumb has already begun showing up in products; chips equipped with the MIPS-16 predecoder should roll out by mid-1997."*

See also Deszo Sima, Terence Fountain, and Peter Kacsuk, *Advanced Computer Architectures, A Design Space Approach*, by Addison-Wesley, 1997.

Copies of the Sima article appearing in *IEEE Micro* and the four referenced *Microprocessor Report* articles which were cited above are on the CD-ROM.

Design Note

Unsuccessful Predecode

An *unsuccessful predecode* signal is represented by generating a predecode pointer that points to the beginning of the predecoded instruction (i.e., to the instruction's PC). This implies an instruction length of zero. Thus, the unsuccessful predecode signal is also being interpreted to mean the instruction "*cannot be decoded by a short decoder.*" The short decoders only operate on instructions that are less than or equal to seven bytes in length. Therefore, if the combination predecode analysis logic determines that an instruction requires more than seven bytes, it considers this an unsuccessful predecode as well, i.e., it is an instruction that cannot be decoded by a short decoder.

The combinational predecode analysis logic shown in Figure 2.7 on page 116 accomplishes several things as it processes three adjacent instruction bytes. It examines:

1. the instruction's opcode byte to determine if: (a) the instruction is a ModR/M instruction and (b) there are any immediate bytes and, if so, how many.
2. the second byte, the ModR/M byte, and decodes the address mode specified by this byte if the instruction is a ModR/M instruction. This analysis will tell if: (a) there is an SIB byte and (b) there are displacement bytes and, if so, how many.
3. the SIB byte, if one is present, to determine if there is a displacement. This will occur only in certain ModR/M address mode cases.

In most cases the existence of a displacement and its size can be determined from the ModR/M byte. In a small number of cases, which are a subset of when an SIB byte is present, the SIB byte must be examined as well to determine if there is a displacement. These processing steps indicate why the combinational predecode analysis logic must examine at least three instruction bytes. In the most general case, however, up to four bytes must be analyzed since the first instruction byte may have been a preface or "0F" byte. If a "0F" byte is present, the predecode analysis logic looks at the next byte as the "real" instruction opcode byte and then does the processing indicated in the above three steps using three instruction bytes. If the bytes required to do these steps are not available, the combinational predecode analysis logic sets the three predecoder pointer bits to zero (resulting in an "unsuccessful predecode" signal to the decoders).

There are eight sets of predecode logic that get reused four times during an I-Cache fill. Cache fills take place in the form of a burst or block transfer of four octets. In the case where the last octet of the cache line is being predecoded, the predecode logic for the last two bytes modifies its behavior and recognizes that it is predecoding not just the last two instruction bytes of any octet, but the last two bytes of a cache line. The last octet of a cache line is actually the first one to be read. More generally, the octets are read in decreasing or reverse order. The logic takes into account the fact that only one or two bytes are available for predecoding. If the predecode logic needs to examine more than the one or two bytes available to determine the instruction's length, then the three predecode pointer bits are set to imply the instruction is not short-decodable.

While the *average* x86 total instruction length is less than four bytes, when the predecode of an instruction detects a length greater than seven bytes (e.g., the opcode byte plus a ModR/M byte plus a 4-byte displacement value plus a 4-byte immediate value), the instruction's length cannot be represented by the three predecode bits. Consequently, the predecoder logic sets the pointer bits to indicate the instruction is not short-decodable. In summary the combination predecode analysis logic has:

1. preface "0F" byte stripping logic.
2. three chunks of logic to look at the opcode byte and the ModR/M and SIB bytes, if they are present. These chunks of logic each produce "partial length information," The first clump of logic (for the opcode) gives a "base instruction length." The second and third clumps of logic give a "ModR/M and SIB instruction length"—which includes the ModR/M byte (if present) plus the SIB byte (if present) plus the number of displacement bytes, if any. The overall instruction length is the sum of these two lengths. That length added to the instruction's address gives the address of the next instruction. The lower three bits of this address is the predecode pointer.
3. a chunk of logic to (a) determine how many bytes must be examined and if enough bytes are not available to indicate an unsuccessful predecode and (b) determine if the instruction length is less than or equal to seven and, if not, force the outputs to indicate an unsuccessful predecode.

Use of the Predecode Bits

We now describe how the predecode bits are used during the instruction decode cycle. The predecode bits are used as two of the five inputs to the multiplexer shown in Figure 2.6 on page 115. The output of this multiplexer is used to align the first instruction as shown in Figure 2.6 on page 115. During the decode process, a number of things occur in parallel. For example, for the first instruction, the predecode logic needs to compare its predecode pointer to the Decode PC to determine if that instruction can be decoded by a short decoder. Similarly, for the second instruction, the logic examines the predecode pointer for the first instruction and compares that with the predecode pointer of the second instruction to determine if the second instruction can also be decoded by a short decoder. Based on those two pieces of information, it can be determined: (a) how many instruction bytes of the instruction buffer need to be "valid", starting from the Decode PC, for the first instruction, and (b) how many instruction bytes of the instruction buffer need to be "valid," starting from the end of the first instruction, for the second instruction.

Definition

Decode PC

The term *Decode PC* refers to the pointer into the instruction buffer, shown in Figure 2.6 on page 115, which points to the start of the instruction to be decoded. In particular, it refers to the low four bits of this pointer.

The instruction registers are *blindly* loaded at the beginning of the decode cycle, (based on the predecode bits), and then everything is examined to see whether one or ideally two instructions are, in fact, decodable. Furthermore, all three decoders blindly operate in parallel before knowing what kind of decode, if any, will be possible. If there is a valid instruction in Instruction Register 1 and if it can be decoded by short decoder 1, then short decoder 1 decodes it. Otherwise, the instruction will be sent to the long decoder and see if that decoder can decode it. If so, the long decoder continues processing the instruction. If the long decoder cannot decode the instruction, than by default the instruction vector decoder will process the instruction. A valid instruction will be decoded by short decoder 2 only if it can be decoded by a short decoder and short decoder 1 has decoded a valid, short-decodable instruction.

"valid" signals:
SDEC0_V
SDEC1_V
LDEC_V
VDEC_V

Without giving a complete description of this control logic within the decoders, we point out that there are four key control signals that get generated by this control logic called SDEC0_V, SDEC1_V, LDEC_V, VDEC_V respectively, where the "V" stands for "valid." The use of these signals will aid us in understanding the use of the predecoder information.

Logic within the decoders examines SDEC0_V first, if it is asserted then one short decode can be done. If SDEC1_V is also asserted, then we can do two short decodes, otherwise we'll just settle for one. If SDEC0_V is not asserted, then LDEC_V is examined. If LDEC_V is asserted, a long decode can be done. Otherwise hopefully VDEC_V is asserted so a vector decode can be done. If it is not, then no instruction decode can be successfully done.

There is also "downstream" control logic that looks at SDEC0_V, SDEC1_V, LDEC_V, and VDEC_V. One of the things that emerges from this logic is the control of the 5-to-1 multiplexer shown in Figure 2.6 on page 115, the output of which is the shaded box "Instruction Buffer Index 0" in Figure 2.6 on page 115. If no successful decode of any type can be done, as was just hypothesized, then the control logic will select the current Decode PC, the second input to the multiplexer, to re-circulate its value. If instead a successful instruction decode can be done, then if at most one short decode can be done, then the third multiplexer input will be selected. If, in fact, two short decodes can be done, the fourth input is selected. If no short decodes can be done, but either a long decode or a vector decode can be done, the fifth input is selected. In the case of a branch type instruction being decoded, then the first multiplexer input is used in the following situations: PC relative jumps, calls, conditional branches (as decoded by either short decoder), and the RET instruction. In summary, the logic that controls the multiplexer must address the questions shown in Figure 2.6 on page 115, namely, "Can a successful instruction decode be done and, if so, how many instruction decodes and what types of instructions are they?"

The Decoders

Decoding of x86 instructions begins when sufficient instruction bytes have been fetched into the instruction buffer. A single stage in the K6 3D's six-stage pipeline is used to decode up to two x86 instructions per cycle. The K6 3D uses a combination of decoders to convert x86 instructions into RISC86 Ops. As shown in Figure 2.2 on page 69, this combination consists of three types of decoders—two short decoders (short decoder 1 and short decoder 2), one long decoder, and one vector decoder. However, as shown in more detail in Figure 2.6 on page 115, another decoder, the exception decoder, which we will learn about later, is also involved.

Historical Comment and Definition

Naming the *Vector* Decoder

The *vector* decoder uses "vector" to mean the "address of the location in a ROM where a sequence of Ops begins." We retain the name the K6 designers gave it for historical purposes. We will learn more about OpQuad Sequences in the section titled "OpQuad Sequences and the RISC86 Operation Set" beginning on page 137.

Comparative Analysis

Pentium II Micro-ops

Intel calls its internal Pentium II microarchitectural operations "micro-ops." The Pentium II has three x86 instruction decoders that can operate in parallel. Two of them (the second and third decoders) handle only "short instructions," each of which produce a single micro-op. As a result, the Pentium II can multiple-decode only very simple instructions. Other reasonably simple instructions such as "ADD reg,mem" can only be decoded in a more limited manner. While this can be a good compromise in the context of new and Pentium II-optimized code, this can be more problematic for decode performance on existing x86 code.

A different approach using two decoders and having each able to decode a larger subset of common instructions was taken within the K6 3D design. This taken as an improvement of an prototype silicon implementation of the processor with three moderately capable decoders able to decode a larger subset of x86 instructions than the two Pentium II simple decoders, but not as large a subset as the final K6 3D decoders. The K6 3D decoders also differ in being nearly symmetric. Most often, one-to-two Op instructions (principally with the exception of x87 floating-point instructions) can be decoded as either the first or the second of a pair of instruction decodes.

What the K6 3D designers found, after analysis of the prototype silicon design and after continuing analysis of a growing variety of instruction traces from PC applications and system software, was the following set of points:

1. a shorter cycle time (and thus higher frequency of operation) was possible by supporting decode of only two versus three x86 instruction decodes.
2. the performance loss due to this change (as judged by instruction traces run through a very detailed and accurate performance model of the design), was far less than the gain in frequency of operation.
3. it was possible to increase the subset of instructions that could be decoded by one of the K6 short decoders without significantly impacting the cycle time. This was because the Op generation by the decoders was not part of the most critical and thus cycle-determining timing paths through the decode logic.
4. significant architectural performance could be gained with just a moderate increase in the size of the short decode instruction subset.

The end result of these realizations was the changeover (in going from the prototype to the final K6 3D design), from having three moderately "simple" decoders to having two more-"sophisticated" decoders. Unlike what is more commonly the case, the frequency of operation and the (net) architectural performance of K6 3D were both significantly improved. Part of this, put differently, is that while the peak decode rate was decreased, the average instruction decode rate was improved!

Each decoder is specialized to handle a specific group of instructions. The use of specialized parallel decoders: (a) significantly improves decode efficiency thereby increasing the decode bandwidth, and (b) enables a shorter clock cycle and thus a higher frequency of operation. The K6 processor classifies short instructions as the most commonly used x86 instructions and those that can be implemented using one or two Ops. These instructions constitute the majority of the x86 instructions and allow the K6 to typically sustain two x86 instruction decodes per cycle. They are handled by the short decoders. Less commonly used instructions, and instructions implemented by three or four Ops, are handled by the long decoder. Finally, x86 instructions that are the least common, the most complex instructions, and instructions that require more than four Ops to be implemented, such as string moves, are decoded using the vector decoder.

The above discussion about the use of the decoders can be expanded in the context of the length of the x86 instructions that are decoded. Short decoder 1 and short decoder 2 decode the most commonly used x86 instructions (e.g., moves, shifts, branches, calls, ALU, MMX, 3D, and FPU instructions) into zero, one, or two Ops each. The short decoders only operate on x86 instructions that are less than or equal to seven bytes in length but can decode up to two such instructions per clock.

The commonly used x86 instructions that are greater than seven bytes in length as well as less commonly used x86 instructions are handled by the long decoder. The long decoder only performs one decode per clock but generates up to four Ops. Both the short decoders and the long decoder can decode instructions longer than seven or twelve bytes when the extra length is due to prefix bytes. The prefix bytes are accumulated and then factored into the one short, long, or vector decode that ultimately occurs.

OpQuad sequences

All x86 instructions that are not handled by either the two parallel short decoders or the long decoder are decoded by the vector decoder (e.g., complex instructions, serializing instructions, interrupts and exceptions, etc.) Short and long decodes are processed completely within their respective decoders. Vector decodes, on the other hand, are started by the vector decoder and then completed by RISC86 OpQuad Sequences fetched from an on-chip OpQuad ROM. The vector decoder logic provides:

1. the initial set of up to four Ops to set up or start execution of the decoded instruction (this initial OpQuad is sometimes referred to as the *vector* OpQuad).
2. a vector (or entry point address) to a sequence of additional Ops stored in the on-chip OpQuad ROM.

No *special* types of RISC86 Ops are stored in the OpQuad ROM; it contains the exact same types of Ops as those that are generated by the hardware decoders.

Articles on CD-ROM

Chapter 4 in the *AMD-K6 3D Processor Code Optimization* Application Note, gives a series of tables showing which decoder and Ops are use for various x86 instructions. This application note is on the CD-ROM.

Decoder Combinations

All short-decodable integer instructions, as well as the MMX and 3D instructions, can be decoded by either short decoder 1 or short decoder 2. As stated earlier, all of the common, and a few of the uncommon, floating-point x86 instructions are decoded by the short decoders. X86 floating-point instructions are also known as ESC instructions. Such decodes generate a RISC86 FpOp and, optionally, an associated RISC86 floating-point LdOp or StOp. Only the first short decoder, short decoder 1, can be used to decode x86 floating-point instructions. This restriction allows the generation of, at most, one FpOp operation per clock, which matches the single floating-point execution pipeline. Non-ESC x86 instructions can be decoded simultaneously by the second short decoder, short decoder 2, along with an ESC instruction decode in the first short decoder.

All of the x86 MMX and the AMD 3D extensions to the x86 instruction set are also decoded by the short decoders. MMX and 3D instruction decodes generate a RISC86 MMXOp and, optionally, an associated RISC86 MMX LdOp or StOp. Both short decoders can decode MMX and 3D instructions. There are no decoding pairing constraints between integer, MMX, and 3D instructions.

Design Note

Simultaneously Decoding ESC and MMX Instructions

Unlike the K6 3D, only the first short decoder, short decoder 1, can be used to decode x86 MMX instructions in the K6. This restriction allows the generation of, at most, one x86 MMXOp per clock and matches the single-pipeline MMX implementation on the K6.

Decoder and Scheduler OpQuads

A copy of the instruction buffer contents is sent to all of the decoders simultaneously. However, only one of the three types of decoders is used during any one decode clock. As mentioned earlier in this chapter, a group of four Ops called an OpQuad is always produced as the output of the short decoders, the long decoder, the vector decoder, and the OpQuad

ROM. When decodes cannot fill a group with four useful Ops, the empty locations of the group are filled with NoOps. For example, a long-decoded x86 instruction that converts to only three Ops is padded with a single NoOp operation and then passed to the scheduler's buffer. Up to six OpQuads, i.e., twenty-four Ops total, can be in the scheduler's buffer at any given time.

An OpQuad passes through the OpQuad expansion logic just before it is loaded into the scheduler as shown in the following Figure 2.8. MUX A and MUX B in this figure are the same as those that appear in Figure 2.6 on page 115.

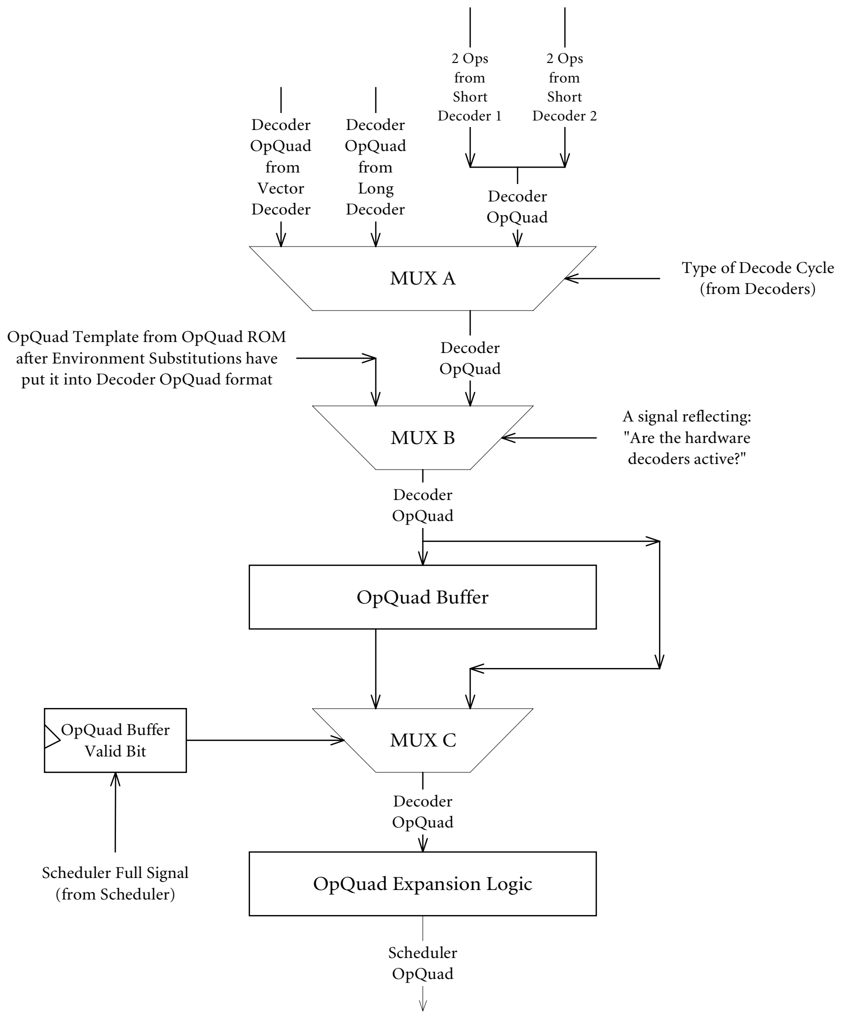

2 Ops from Short Decoder 1
2 Ops from Short Decoder 2
Decoder OpQuad from Vector Decoder
Decoder OpQuad from Long Decoder
Decoder OpQuad
MUX A
Type of Decode Cycle (from Decoders)
OpQuad Template from OpQuad ROM after Environment Substitutions have put it into Decoder OpQuad format
Decoder OpQuad
MUX B
A signal reflecting: "Are the hardware decoders active?"
Decoder OpQuad
OpQuad Buffer
OpQuad Buffer Valid Bit
MUX C
Decoder OpQuad
Scheduler Full Signal (from Scheduler)
OpQuad Expansion Logic
Scheduler OpQuad

Figure 2.8 DECODER OPQUADS AND SCHEDULER OPQUADS

To distinguish between the OpQuads produced by the decoders and the OpQuads produced by the OpQuad Expansion Logic, they will be referred to as *decoder* OpQuads and *scheduler* OpQuads respectively. A *decoder* OpQuad consists of four 38-bit Ops to form a 152-bit OpQuad. We will learn in Chapter 3 that a *scheduler* OpQuad consists of a variety of static and dynamic fields that total to 578 bits. OpQuads stored in the OpQuad ROM are essentially 152-bit decoder OpQuads with an additional 14-bit sequencing field associated with them for use with the OpQuad ROM fetch control logic. When they are stored in the OpQuad ROM, these ROM-based OpQuads have an internal 166-bit OpQuad format. However, only their decoder OpQuad component is ever sent to the scheduler. Thus, in all cases, as is shown in Figure 2.8 only 152-bit *decoder* OpQuads are sent to the OpQuad Expansion Logic to be expanded into *578 bit scheduler* OpQuads. The contents and formats of *decoder* OpQuads will be discussed later in this chapter as will all of the resources shown in Figure 2.8. *Scheduler* OpQuads, as noted above, are discussed in Chapter 3.

Before leaving this section, we call your attention to the control signal for the MUX B multiplexer in this diagram. There is some control logic and a single bit of state in the scheduler that is used to distinguish between the use of the hardware decoders and the use of OpQuad Sequences fetched from the OpQuad ROM. We will identify these two states respectively as the "hardware decoders are active" state and the "OpQuad ROM fetch is active" state. Whenever the system is in "OpQuad ROM fetch is active" state, it will remain in the state until the last OpQuad in the OpQuad Sequence, as identified by action in the sequencing field of a ROM-based OpQuad, has completed processing. When this occurs the state will change to the "hardware decoders are active" state. The system will remain in this state until a condition arises, such as a vector decode is encountered or an exception abort is encountered, that require OpQuad Sequences to be fetched from the OpQuad ROM. This will be discussed later in this chapter.

The Scheduler

As mentioned earlier, the scheduler is the heart of the K6 microarchitecture. It contains the logic necessary to manage out-of-order issue, speculative execution, data forwarding, register renaming, and the simultaneous issue, execution, and retirement of multiple Ops. Said differently, the scheduler, shown in Figure 2.9 on page 130, contains logic to track Ops throughout their lifetime, determine dependencies, schedule execution, and commit architecture state. Whenever possible, the scheduler can simultaneously issue Ops to appropriate execution units.

Comparative Analysis

Combined Reservation Station and Reorder Buffer

The K6's scheduler includes both centralized reservation station and reorder buffer functionality. This is in contrast to other processor designs, such as the Intel Pentium Pro, where these are separate structures.

One of the main components of the scheduler is its centralized buffer. The main advantage of the scheduler and its centralized buffer is the ability to examine the Ops associated with up to twelve x86 instructions at one time. The scheduler examines the contents of the buffer in parallel and performs dynamic scheduling of the Ops for optimized execution. Since the scheduler can issue the Ops out-of-order and speculatively, the corresponding x86 instructions are executed out of order and speculatively. However, the scheduler always retires the Ops in order; thus, the x86 instructions are always retired in order. In total, the scheduler can issue up to six Ops and retire up to four Ops per cycle.

The OCU (Op Commit Unit) is also shown in this figure. It is separated from the scheduler's resources by the dashed line to indicate that it is not part of the scheduler. It has been included in the figure because it will be discussed later in this section in terms of its interaction with the scheduler.

The scheduler's buffer is logically structured as a FIFO (first-in, first-out) queue. Younger Ops (*later in the program order*) enter at the top of the buffer. Older Ops (*earlier in the program order*) are at the bottom of the buffer. Ops enter the top of the buffer four at a time as a *scheduler* OpQuad and are retired up to four at a time (again as a *scheduler* OpQuad) from the bottom. For purposes of this discussion, the buffer will be thought of as either having six elements (each element being a *scheduler* OpQuad) or being six rows deep. The rows are numbered from 0 to 5, with 0 referring to the top row (youngest Ops) and 5 referring to the last row (oldest Ops). Each of the buffer's six rows has four entries, one for each of the four Ops. The entry for an Op is called the Op Entry associated with that Op. A row of four Op entries corresponds to a scheduler OpQuad—the four entries/row correspond to four Ops/row or one scheduler OpQuad/row.

The execution units do not have specialized reservation stations or queues that are blocked if the execution unit itself is stalled. Rather, the pending Ops are in the centralized buffer's Op entries. Many Ops are immediately eligible for execution when they are loaded into the top row of the buffer, but may, in fact, be issued to appropriate execution units from any point in the buffer.

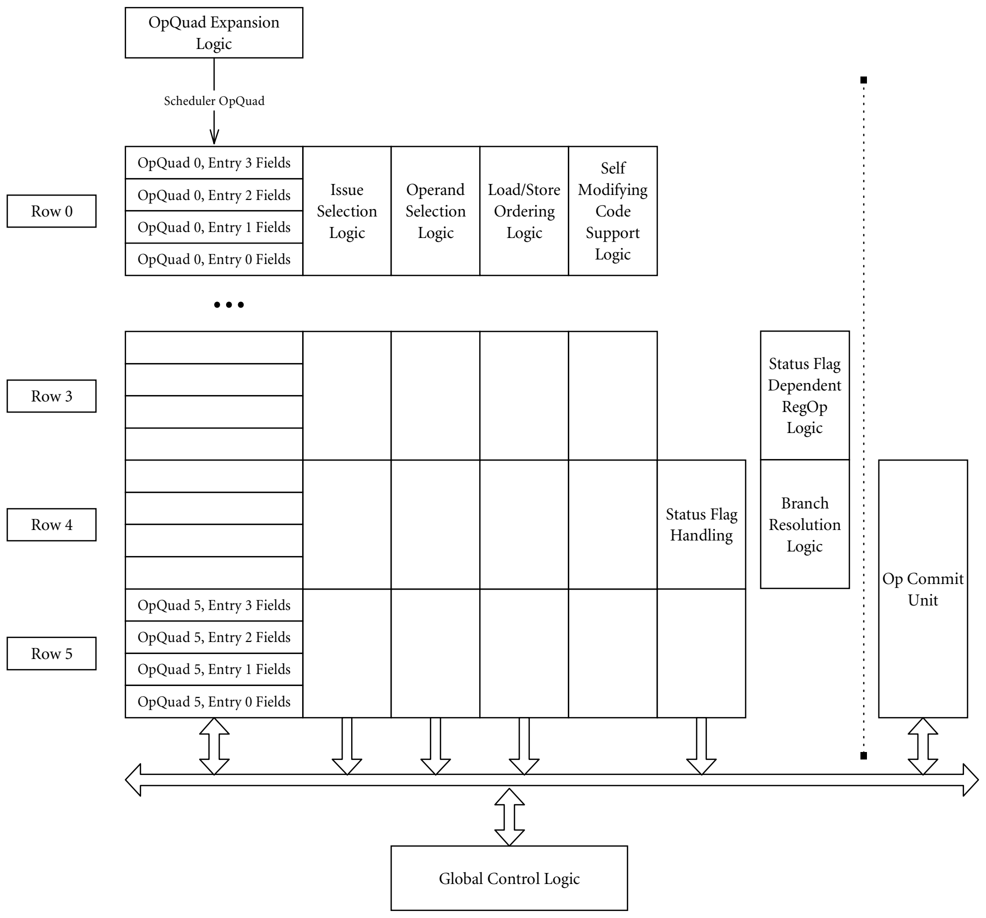


Figure 2.9 THE SCHEDULER AND ITS CENTRALIZED BUFFER

State field

Each Op entry has a State field that indicates whether the Op has been issued, is in a specific stage of an execution pipeline, or has been completed.

Video Clip on CD-ROM

	In this video clip, Amos Ben-Meir, Principal Designer of the K6 3D addresses the following question, "How does the K6 3D scheduler work?

Design Note

State and Position are Independent

The state of the Op is independent of its position in the buffer. However, the longer an Op is in the scheduler, the greater the chance that the Op will be issued and completed.

Each clock cycle:

1. new Ops can be loaded into the buffer.
2. existing Ops can be issued, advanced in their execution, and completed (with appropriate updating of their execution state).
3. old and completed Ops can be committed and removed from the buffer.

All LdOps and StOps and most RegOps can execute from any row in the buffer. However, some Ops (such as the evaluation of conditional branches and RegOps that depend on status flags) are executed when the Ops reach a particular row of the buffer. This simplifies and speeds up the hardware by eliminating a design requiring the ability to support execution of these "special" Ops in other rows. Scheduling delays are minimized by selecting the row for executing such operations to be where the necessary operands are likely to be available. For example, Ops that depend on status flags (such as ADD-with-carry) are handled lower in the scheduler at a point where older operations are likely to have completed modification of the status flag values required for the completion of the status flag- dependent operation.

Design Note

Execution of Status Flag-Dependent Ops

Additional circuitry that would allow execution of status flag-dependent Ops higher in the scheduler's buffer would provide minimal improvement in Op execution rate because the necessary flags are unlikely to be available when a status flag-dependent Op is in a higher row of the buffer.

The scheduler issues (i.e., dispatches) the Ops to the execution units. The Ops are selected and issued by the issue selection logic according to:

The terms dispatches *and* issues *are used interchangeably in this book.*

1. the type and availability of an execution unit—this means that different types of Ops can be executed out of order with respect to each.
2. sequential program order.

The physical position of an entry in the buffer indicates the program order of the corresponding Op. Each entry contains:

1. storage for the information required for the execution of the associated Op and storage for the result values produced by the execution of the Op.
2. logic for directing the information to the correct execution unit when required.
3. logic for detecting and handling register, status flag, and memory dependencies.

The result values stored in an entry provide the register and status values at the corresponding point in the program order. The buffer's entries are general in the sense that any entry can be used for any type of Op. There is no need for separate, specialized queues for Ops destined for different execution units—the entries in the scheduler are not specialized according to the type of Op that is to be executed.

The determination of dependencies between Ops and the generation of operand forwarding controls takes advantage of the physical ordering of the Ops within the buffer and also depends on the fact that all Ops reside in this central, general entry buffer. Furthermore, as will be seen later, the scheduling and execution of an Op is independent of the grouping of Ops into rows.

Design Note

Position in the Scheduler and Program Order

The physical position of an entry in the scheduler's buffer indicates the program order of the corresponding Op. The result values stored in an entry provide the register and status values, produced by that specific Op, at the corresponding point in the program order.

The scheduler retains result values until the OCU determines that no exception and no mispredicted branch precedes the associated operation. After the execution of an abortable operation, the results of the operation are kept in the associated scheduler entry and/or in a store queue. If the OCU determines that the oldest executed operations would be generated in a sequential execution of the program, the results are made permanent by writing them to a register file, a status register, cache, or memory, and the operation is retired. If the OCU determines that a result would not be generated in a sequential execution of the program due to an exception or mis-predicted branch at that point, the operation is retired without making permanent changes.

Abortable state changes are supported by the scheduler and the store queue through the general technique of temporarily storing (a) register and status results in the scheduler and (b) memory write data in store queue entries until the associated Ops are committed and retired. Permanent state changes are made during Op commitment when it is safe and definite for the changes to be made. While these new state values reside in the scheduler and the store queue, they are forwarded to dependent Ops as necessary. Nonabortable state changes, in contrast, occur immediately during execution of certain special RegOPs and the responsibility or burden is placed on the OpQuad Sequences containing these RegOPs to ensure sufficient synchronization with surrounding operations.

Later it will be shown that the scheduler incorporates the functions of a reorder buffer and implied register renaming. Explicit tags indicating the program order of operation results are not used. The physical positions of entries in the scheduler indicate the program order of the corresponding operations. The result values stored in an entry provide the register and status values of the "renamed" registers at the associated point in the program order. Explicit register renaming is not required. Exactly how renaming is accomplished will be discussed in Chapter 3. Briefly, the scheduler's use scan chains which, when directed in the proper physical direction across the scheduler Op entries, locate preceding or older Ops that affect desired register operands and status flags for subsequent operations.

Audio Clip on CD-ROM

This audio clip gives a summary of the K6 3D's scheduler's functions.

The following is a summary of the role each of the units identified in Figure 2.9 on page 130 play in the scheduler processes just described. Each unit is discussed in detail in Chapter 3.

Issue Selection Logic

The issue selection logic is involved with the selection of the next Ops to enter the LU, SU, RUX, and the RUY processing pipelines—i.e., four Op selections occur every clock. Each cycle, based on the updated state information in the scheduler as of the beginning of the cycle, the issue selection logic performs a selection process to determine the next LdOp, the next StOp, and the next two RegOps to be issued into the corresponding execution unit processing pipelines.

Historical Comment and Suggested Readings

Reorder Buffer

Processors that support speculative and out-of-order execution typically have operations completing execution before they are ready to be committed. The results of such operations are not committed (i.e., producing permanent state change) until it is safe to do so. The collection of storage elements that hold the results of the as-yet-uncommitted operations is often called a reorder buffer, for it is from this buffer that the instructions which have been executed out of order will be reordered and committed in an in-order fashion. A reorder buffer also supports the use and forwarding of results of completed operations as source operands for other dependent operations. The K6 is an example of a microprocessor in which its reorder buffer (included in the scheduler's centralized buffer functionality) also serves as an environment to support register renaming[a]. As we will learn in Chapter 3, the K6's renaming registers hold result register values in the DestVal field of the appropriate Op entries in the scheduler until they are committed. In the K6, a single unified structure, the scheduler's buffer, was designed to hold all information related to the processing of an Op throughout its lifetime. All Ops enter the scheduler after being decoded (or fetched from the OpQuad Sequence ROM) and remain there until the end of their life (i.e., until they are committed or removed). Most out-of-order execution designs utilize separate structures for the functions of a reorder buffer, reservation stations, and possibly other dispatch queues.

See, "Implementation of Precise Interrupts in Pipelined Processors," by J. E. Smith and A. R. Pleszkum, *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June 1985, pp. 34-44. You can find the full text version of this article on the CD-ROM.

[a] See the section titled "Register Renaming" beginning on page 300.

Operand Selection Logic

The operand selection logic is involved with determining:

1. the status of each value, i.e., whether a valid value is or is not available from the designated source.
2. where each of nine operand values actually needs to come from—i.e., from which specific scheduler Op entry, architectural register, or execution unit result bus.

Based on this information, the scheduler determines which Ops will be able to advance in their respective execution pipelines and actually start execution.

Load/Store Ordering Logic

Just as certain execution ordering must be maintained between Ops due to register dependencies, a certain degree of execution ordering must be maintained between LdOps and StOps due to memory dependencies. For example, LdOps cannot freely execute ahead of older StOps. There are two chunks of logic, one associated with the LU pipeline and one associated with the SU pipeline that deal with this ordering and determine when given LdOps and StOps are independent of each other and thus can safely be allowed to execute out of order with respect to one another. These two chunks of logic are collectively referred to as the load/store ordering logic in Figure 2.9 on page 130.

Status Flag Handling Logic

There is a chunk of scheduler logic, called the status flag handling logic, associated with the fetching and usage of status flag operand values. Two relatively independent areas are involved: the fetching of status flag values for status-dependent RegOps and the fetching of status flag values for the resolution of BRCOND Ops.

Status Flag-Dependent RegOp Logic

cc-dependent Ops
cc-dep Ops

All status-dependent RegOps, which are referred to as condition code dependent, "cc-dependent," or "cc-dep" Ops, are executed by the RUX or RUY execution units and require their status operand value with the same timing as their register operand values. The status flag-dependent RegOp logic is responsible for ensuring that this happens correctly.

Branch Resolution Logic

As is shown in Figure 2.21 on page 175, a BRCOND Ops (BrOp) does not require any actual execution processing. Instead, while a BRCOND Op is outstanding and before it reaches the bottom row of the scheduler's buffer, it must be resolved as to whether the associated conditional branch instruction was correctly predicted or not. This is done for each BRCOND Op, in order, at a rate of up to one per cycle. When the above status flag-handling logic obtains the appropriate status for the next unresolved BRCOND Op, the appropriate set of status flag values is used to determine if the condition code specified within the BRCOND Op is TRUE or FALSE. If valid values for the required status flags are not yet all available, then resolution of the Op is held up.

If the branch condition is FALSE, the BRCOND Op was incorrectly predicted and an appropriate restart signal is immediately asserted to restart the upper portion of the processor at the correct next program address (i.e., the instruction fetch and decode portion—see Figure 2.4 on page 87). The correct address is either the branch target address or next sequential address, whichever was not predicted.

If the branch was correctly predicted, then nothing happens other than BRCOND Op resolution processing advances on to the next

BRCOND Op. The branch resolution logic (also called the BRU earlier) is concerned with resolving these issues.

Self-Modifying Code Support Logic

The self-modifying code support logic is concerned with the detection and handling of self-modifying code. Logically, in the scheduler, a detection of self-modifying code is treated as a type of an instruction trap (see the section titled "Handling Faults, Traps, and Precise Interrupts" beginning on page 175). The store queue provides the physical address of the store it is preparing to commit. It supplies "linear address" bits which are logically also the "physical address" bits since these bits are only untranslated bits. The bits are compared against the instruction addresses of each following OpQuad. If any OpQuad addresses match, then there may be a write to an instruction which has already been fetched, decoded, and is now in the scheduler. This is a potential self-modifying code situation. Accordingly, the scheduler is then flushed of these following OpQuads and the upper portion is restarted to refetch and redecode the associated x86 instructions, starting with the instruction after the "modifying" instruction.

Design Note

False Self-Modifying Code Traps

Due to cycle-time constraints, the self-modifying code logic does not compare all of the address bits. Therefore, an "address" match does not necessarily mean that the code is self-modifying. It is possible for a *false self-modifying code trap* to occur. Since the lower address bits are compared, these false traps can only occur when the memory size exceeds 1Mbyte, and even then they are statistically rare. In this situation, the processor is flushed as with normal trap handling, resulting in a small and transient performance loss.

Global Control Logic

Basically, the global control logic coordinates the overall operation of the scheduler. For example, it issues the control signals to load the pipeline registers and it controls the source operand input multiplexers for each of the execution units.

OpQuad Expansion Logic

This chunk of logic, which is shown in Figure 2.8 on page 127, expands *decoder* OpQuads into *scheduler* OpQuads before they are loaded into the top row of the scheduler. A description of what occurs during this expansion process is given in Chapter 3.

Op Commit Unit

The OCU operates in conjunction with the scheduler and generally operates on the Ops within the bottom two rows of the scheduler. During each cycle the OCU examines each of the Ops within the bottom OpQuad and tries to commit the results of as many of these Ops as possible. It is possible for the state changes of all four Ops to be committed in one cycle or for this to take many cycles. If all the Ops of an OpQuad have been committed or are being successfully committed, then the OpQuad is retired from the scheduler at the end of the current cycle. Otherwise, as many changes as possible are committed during the current cycle and the process is repeated on successive cycles until all changes have been committed. In some cases, the OCU will also look ahead into the second from the bottom OpQuad to (a) start committing StOps while RegOps in the bottom OpQuad remain to be committed, or (b) commit, in certain potential deadlock situations, register results while the bottom OpQuad cannot yet be retired from the scheduler. Thus the OCU's principal function is to commit the results of Ops and then to retire them from the scheduler. It also handles mispredicted BRCOND Ops by initiating abort cycles for them.

OpQuad Sequences and the RISC86 Operation Set

The performance of a microprocessor that decodes CISC instructions into RISC operations for execution on a RISC microarchitecture depends greatly on the number of RISC operations produced from a single CISC instruction. The decoding of the x86 instructions involves, in part, a mapping of variations of similar instructions into one or more Ops, (e.g., the decoding of the many variations of an x86 ADD instruction into an ADD Op.) A large number of the x86 memory addressing forms of a particular instruction can be converted into load or store Ops (LdOps or StOps) accessing memory in combination with a register-to-register Op (a RegOp).

OpQuad Sequences

Instruction decoders in superscalar microprocessors such as the Pentium, Pentium Pro, and the K6 often include one or more decoding pathways in which x86 instructions are decoded by hardware logic and a separate decoding pathway which uses an on-chip ROM memory for fetching an OpQuad Sequence that corresponds to a complex or uncommon x86 instruction. OpQuad Sequences may also be initiated as a result of servicing an exception or a trap. There is an important issue centered around an OpQuad Sequence that realizes the semantics of an x86 instruction. X86 instructions are atomic entities in the x86 instruction set architecture. This means that once they begin executing, they complete execution without any interruption. Thus, the x86 instruction's OpQuad Sequence must also be atomic—i.e., it must finish execution without interruption. To achieve this, the hardware decoders are inactive when an OpQuad Sequence from the OpQuad ROM is in the process of executing (see a related comment in the section titled "Use of the Predecode Bits" beginning on page 121).

OpQuad ROM

One problem with using the *OpQuad ROM* for storing OpQuad Sequences is that the process of accessing a ROM is typically slower and less efficient than hardwired decoding of instructions.[19] Another problem is that if a substantial number of x86 instructions are implemented by OpQuad Sequences, a large OpQuad ROM is required for storing them. Obviously, if the OpQuad ROM is large, there is increased circuit complexity for deriving and applying the pointer (vector) into it to identify the appropriate OpQuad sequence. Increased circuit complexity directly relates to increased overhead, which reduces instruction decoding throughput. Furthermore, a large OpQuad ROM increases the size of the processor's circuitry which, in turn, can reduce the manufacturing yields and increase production costs.

Historical Comment and Suggested Readings

Environment Substitution

Designers of some early microprogrammable machines recognized that a number of microcode sequences used to emulate ISAs were quite similar, differing, for example, only in the specific registers manipulated. To take advantage of this situation, the specific registers used in the microcode sequence were specified in auxiliary registers whose values were "merged" with the microcode sequence as it was being executed. Such implementations can be viewed as precursors to the K6's OpQuad Template environment substitution and its use of dynamic fields (see, for example, the article by Gerald Jay Sussman, Jack Holloway, Guy Lewis Steel, Jr., and Alan Bell, "Scheme-79—Lisp on a Chip," in *Computer*, July, 1981, pp. 10-21). For a description of a microarchitecture which dealt with some of these issues and proposed a "snooper" facility for doing both microcode debugging and performance measurements see, B. D. Shriver, "A Description of the Mathilda System," Department of Computer Science, University of Aarhus, Denmark, April 1973.

The approach taken to resolve the issues surrounding the use of an OpQuad ROM and OpQuad Sequences in the K6 was to design a RISC86 Op Set that would:

1. facilitate the decoding of as many x86 instructions as possible into a small (i.e.,"minimum") number of OpQuad Sequences.
2. permit more common x86 instructions to be decoded using hardwired logic, versus being implemented using OpQuad Sequences stored in the OpQuad ROM.

[19] It is interesting to note that the K6 has the same Op fetch/generation bandwidth in either case, as will be shown later.

To achieve this, RISC86 Ops are relatively "powerful"—for example, LdOps and StOps support (base + scaled index + displacement) address computations within one Op and executing in one cycle. The K6's OpQuad ROM and its relationship to other resources (such as the instruction vector decoder and the exception vector decoder) is shown in Figure 2.6 on page 115. The relationship is shown in more detail in Figure 2.10.

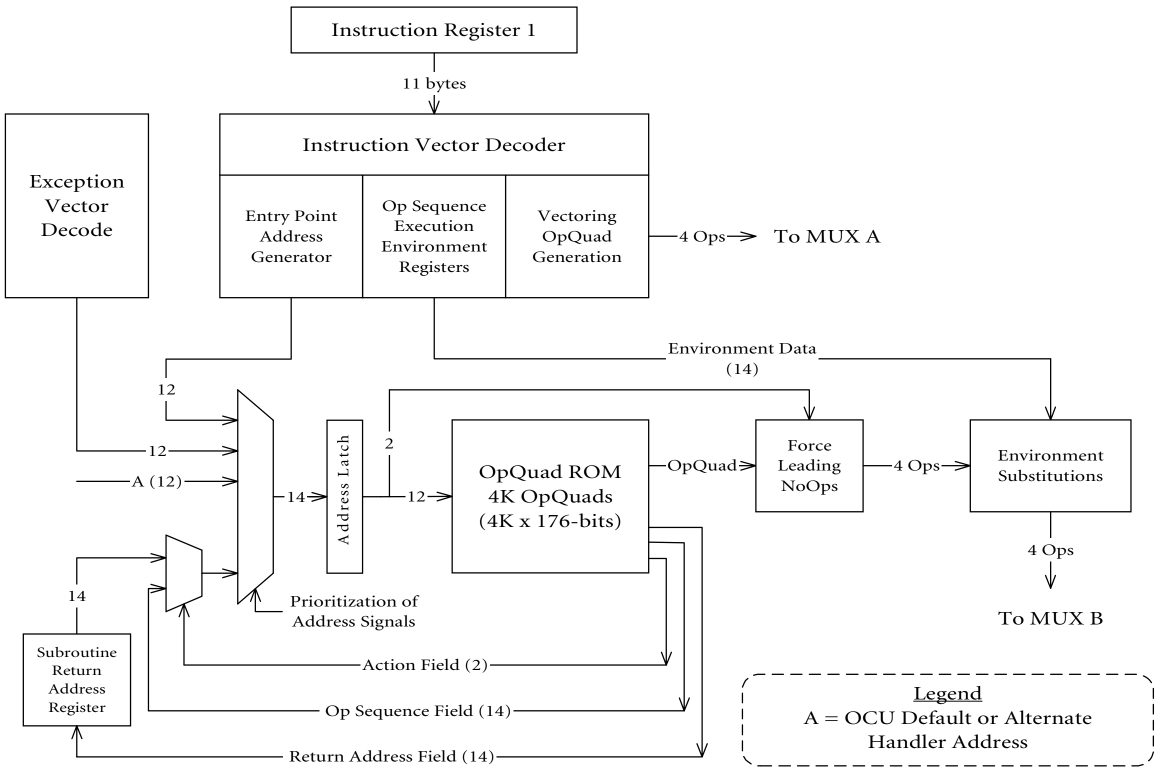

Instruction Register 1
11 bytes
Instruction Vector Decoder
Exception Vector Decode
Entry Point Address Generator
Op Sequence Execution Environment Registers
Vectoring OpQuad Generation
4 Ops
To MUX A
Environment Data (14)
12
12
A (12)
14
Address Latch
2
12
OpQuad ROM 4K OpQuads (4K x 176-bits)
OpQuad
Force Leading NoOps
4 Ops
Environment Substitutions
4 Ops
To MUX B
14
Prioritization of Address Signals
Subroutine Return Address Register
Action Field (2)
Op Sequence Field (14)
Return Address Field (14)
Legend
A = OCU Default or Alternate Handler Address

Figure 2.10 OpQuad ROM, Vector Decoder, and Exception Decoder

It is important to understand that *all* x86 instructions have a corresponding OpQuad sequence in the OpQuad ROM. The hardware decoders can be viewed as a performance enhancement over invoking every specific OpQuad sequence when needed. The presence of the hardware decoders also reduces the pressure to maximize the performance of the logic associated with implementing OpQuad sequences.

Design Note

Indirect Register Names

The "reg" and "regm" mnemonics in Table 2.4 on page 92 represent indirect register names for the x86's 32-bit integer registers (i.e., AX-DI or AL-BH). They are replaced, at Op decode time, by the current register number (i.e., 00xxx) from the corresponding EmReg or EmRegM execution environment variable. Similarly, the "MMreg" and "MMregm" mnemonics represent indirect register names for the x86's 64-bit MMX registers (i.e., MM0-MM7). They are replaced, at Op decode time, by the current register number (i.e., 11xxx) from the corresponding EmReg or EmRegM environment substitution variable. As was seen in Table 2.4, the indirect register names have multiple encoding. This is done to reduce decode logic.

The K6's RISC86 Op Set is highly regular with a fixed length and format for each Op type. By comparison, conventional x86 instructions are highly irregular, having greatly different instruction lengths and formats. OpQuad Sequences, when required, are implemented in a small OpQuad ROM and exploit the reuse of operation structures for variations that are common among x86 instructions.

OpQuad templates

environment substitutions

OpQuad eexecution environment registers

force leading NoOps logic

The short decoders and the long decoder produce decoder OpQuads. The instruction vector decoder produces an initial decoder OpQuad and an entry point address to an OpQuad Sequence in the OpQuad ROM. The OpQuad Sequence consists of a sequence of *OpQuad templates*. The *environment substitutions* logic, shown in Figure 2.6 on page 115 and Figure 2.10 on page 139, processes various fields in the OpQuad templates and outputs decoder OpQuads. An OpQuad Sequence environment is maintained in the *OpQuad Execution Environment Registers* in the instruction vector decoder and includes default address and data sizes for the code segment and the register numbers from the x86 instruction. The environment variables allow a section of an OpQuad Sequence to be re-used for different x86 instructions by proper replacement of field values with environmental variables appropriate for the code section and x86 instruction. Importantly, the OpQuad ROM contains OpQuad Sequences where not all of the Ops in the OpQuads are actually part of the implementation of one specific x86 instruction. Such Ops are changed to NoOps by the *Force Leading NoOps logic* as required for the x86 instruction being decoded. The use of the execution environment substitution achieves encoding of the x86 instruction set architecture functionality while substantially reducing the number and size of the code sequences in the OpQuad ROM. Correspondingly, this approach also reduces the size and cost of required circuitry.

The exception vector decoder and the OCU produce entry point addresses into the OpQuad ROM, but do not produce an initial vectoring OpQuad as the hardware instruction vector decoder does. Concurrent with an abort cycle, the OCU also vectors the machine to one of two possible OpQuad Sequence entry point addresses—either the "default" OCU fault handler address or an "alternate" OCU handler address (see "A" in Figure 2.10). The setting of these addresses is supported by the LDDHA Op (load *default handler address*) and the LDAHA Op (load *alternate handler address*). The default fault handler address is initialized by the processor reset OpQuad Sequence and the alternate handler address is specified within OpQuad sequences for some x86 instructions and for some cases of exception processing. Alternative handler addresses remain active from execution of the LDAHA Op (a) until the end of the instruction or exception processing OpQuad sequence, or (b) until another LDAHA Op executes. At this point, control for subsequent faults reverts to the default fault handler addresses.

DESIGN NOTE

Decoder OpQuads, *Scheduler* OpQuads, and OpQuad Templates

As was mentioned earlier and as shown in Figure 2.8 on page 127, the *decoder* OpQuads produced by the decoders pass through the OpQuad Expansion Logic to produce *scheduler* OpQuads prior to being loaded into the scheduler's buffer. When we use the term "RISC86 Ops," we are referring to the Ops in *decoder* OpQuads and the fields and formats of these Ops. The differences between the contents of *decoder* OpQuads and *scheduler* OpQuads is discussed in Chapter 3. The OpQuad ROM stores OpQuad Templates that are changed into *decoder* OpQuads by the environmental substitution logic.

There are a few additional resources in Figure 2.10 on page 139 that we should explain to give you a more complete understanding of the decoding process and OpQuad ROM work.

The Op Sequence and Action Fields

Every OpQuad *must* specify a branch, even if it is only a branch to the next sequential instruction in the ROM. Branches in OpQuad sequences are specified by a combination of the SpecOp Type field (discussed later in Table 2.39 on page 155), the *Op Sequence field* and the *action field*, both of which are shown in Figure 2.10. There is a single Op Sequence field and a single action field for each OpQuad in the OpQuad ROM. The action field specifies the type of branch. The branches may be conditional. Thus, in addition to the Op sequence and action fields, the OpQuad may also contain a BRCOND Op. If there is a BRCOND Op, then the OpQuad sequence branch is considered to be a conditional branch. If it does not, then the OpQuad sequence branch is considered to specify an unconditional branch. The exception to this is the OpQuad sequence subroutine branch (BSR), discussed below, which is always unconditional, yet always requires a BRCOND Op in the OpQuad to supply the return address in the *Return Address Field*. You can have four Ops plus a branch in a single OpQuad if the branch is unconditional, but only three Ops plus a branch in an OpQuad if the branch is conditional (or is a BSR branch).

OpQuad Sequence Subroutines

An OpQuad sequence can branch to a subroutine in the OpQuad ROM by executing the BSR branch described above. The Subroutine Return Address Register represents a one-deep OpQuad Sequence return address stack. This allows one level of subroutine nesting within OpQuad sequences. Being only one entry, this allows extremely simple control logic (for example, no actual top-of-stack pointer needs to be maintained, particularly in the face of mispredicted OpQuad sequence branch and exceptions). In practice this proves sufficient for most all OpQuad sequences across the entire x86 instruction set architecture functionality. OpQuad sequence subroutines need not return to the OpQuad following the caller.

Initial Vector OpQuad Generation Logic

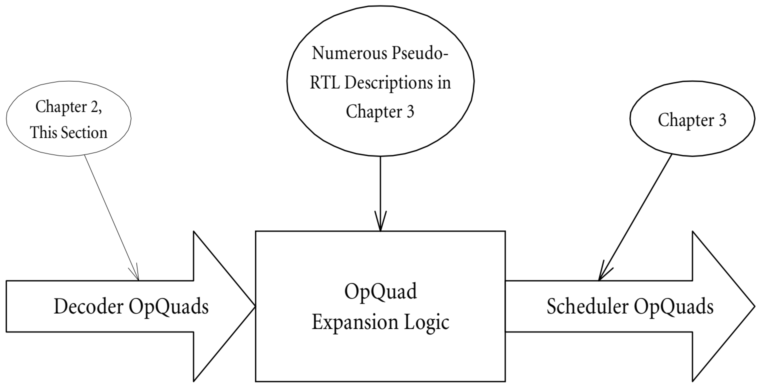
The Initial Vector OpQuad generation logic handles the generation of an initial four Ops during an instruction vector-decode cycle (in parallel with generation of a ROM entry point address of the appropriate instruction OpQuad Sequence). During successive clock cycles, additional OpQuads are supplied by the ROM. The initial vectoring OpQuad varies between instructions. There are special cases for certain specific instructions, but most are grouped into a small number of categories or cases. For each of these general cases (e.g., for the register and for the memory forms of x86 mod R/M instructions), the Ops in the vectoring OpQuad primarily serve to set up various immediate displacement, and/or effective memory address values in scratch integer registers (via LIMM Ops). The fourth of the four Ops is typically the address of the next sequential instruction.

Address Latch

And finally in Figure 2.10 on page 139 there is the *Address Latch*. This latch is simply a registered address input to the ROM. This holds the ROM fetch address from the end of one clock cycle to the start (and completion) of the fetch itself in the next cycle.

FORMATS FOR DECODER OPS

A *decoder* OpQuad consists of four 38-bit Ops. Decoder OpQuads can contain LdOps and StOps, RegOps, SpecOps, and LIMMOps. The formats for these Ops are shown in Table 2.22 on page 143, Table 2.30 on page 150, Table 2.38 on page 155, and Table 2.42 on page 158, respectively. Although some of the fields are similar, there are a number of differences among these Op formats. Having an understanding of these fields will help you understand the mappings between the decoder OpQuads and scheduler OpQuads that the OpQuad Expansion Logic implements. The following diagram indicates how we will cover these topics:



The *decoder* Op formats are presented in this section. The *scheduler* OpQuad formats are discussed in Chapter 3. Explanations of various chunks of the OpQuad Expansion Logic exist in numerous pseudo-RTL descriptions running throughout Chapter 3. We'll begin by looking at the format of LdOps and StOps.

LdOp and StOp Field Descriptions

LdOps and StOps perform memory accesses and related operations. They have the following format in a *decoder* OpQuad:

Table 2.22 Decoder OpQuad LdOp and StOp Format

37 36	35 32	31 30	29 26	25 24	23 22	21 17	16	15 12	11 4	3 0
0 1	Type	ISF	Seg	ASz	DSz	Data	LD	Base	Disp8	Index

We now describe each of the fields in this table: Type[3:0], ASz[1:0], DSz[1:0], Data[4:0], Seg[3:0], Base[3:0], Index[3:0], ISF[1:0], LD, and Disp8[7:0]. The Type[3:0] field specifies the specific type of LdOp or StOp to be performed according to the following table:

Historical Comment and Suggested Readings

Expanding Microinstructions

The technique of expanding a "smaller" microinstruction into one or more "larger" microinstructions has a long history of use in processor design. The Nanodata QM-1 is typical of two-level emulation systems. The machine had both a control store for holding microinstructions and a nanostore for holding nanoinstructions. The control store consisted of from 2K to 65K 16-bit words and the nanostore consisted of up to 1K 342-bit words. A microinstruction (usually) consisted of a 6-bit opcode field and two 5-bit address parts. Microprograms consisted of sequences of microinstructions. Microinstructions can be viewed as pointers in nanostore. A nanoinstruction consisted of a 38-bit constant field and four 76-bit T-fields. Execution progressed from the first T-field, then to the second T-field and so on and then returned to the first T-field. A nanoinstruction could specify a branch to another nanoword specified by a 10-bit subfield of the K-field. T-fields can be skipped. These features allowed a microinstruction to be realized by: (1) the execution of one-to-four T-field operations in a single nanoinstruction, (2) a nanoprogram consisting of the iterative execution of a single nanoinstruction, (3) a nanoprogram consisting of the successive execution of multiple nanoinstructions, (4) a nanoprogram combining nanoprograms of types 2 and 3. One can view the 16-bit microinstructions being expanded in place into a set of 342-bit nanoinstructions during the execution of a microprogram. Given this, there are two interesting features to mention in this historical comment: (a) there could exist nanowords that were common to more than one microinstruction and (b) one could essentially use some of the fields of a microinstruction as parameters to the nanoprogram. The QM-1, like other microprogrammable machines, made extensive use of residual data and control. This allowed microinstructions to set up an environment for execution by other microinstructions. In the case of the QM-1, this meant for nanoprograms as well. Of interest is the fact that (a) the mapping of microprogram-accessible registers' host resources could be statically established upon entry to an emulator or dynamically altered by reestablishing bus address mappings within microprograms, and (b) the addresses of interrupt-specific nanoprograms were specified in a set of registers so they could be dynamically changed by the nanoprogrammer (see, for example, the articles by Robert F. Rosin, Gideon Frieder, and Richard H. Eckhouse, Jr., "An Environment for Research in Microprogramming and Emulation," in *Communications of the ACM*, Vol. 15, No. 8, August 1972, pp. 748-760, and by Michael J. Flynn and Robert F. Rosin, "Microprogramming: an Introduction and a Viewpoint," *IEEE Transactions on Computers,* July, 1971, pp. 721-731).

For an interesting discussion of various vertical, diagonal, and horizontal single-level and two-level microinstruction formats for a variety of different processors see the book, *Foundations of Microprogramming,* by Ashok K. Agrawala and Tomlinson G. Rauscher, Academic Press, 1976. A number of early papers dealing with microcode verification can be found in *Firmware, Microprogramming and Restructurable Hardware,* by Gerhard Chroust and Jorg R. Muhlbacher, North-Holland Publishing Company, 1980.

Table 2.23 LdOp and StOp Type(3:0) Field

Type(3:0)	Op Symbol	Type of LdOp or StOp to be Performed
0000	LD, LDL	Load integer data
0001	LDF	Load floating-point data
0010	LDST	Load integer data with store check
0011	LDM	Load MMX or 3D data
0100	CDAF(X)	CDA (see below) plus flush cache line(s)
0101	LDPF	Load Prefetch (prefetches a block)
0110	LDSTL	Load integer data with store check, locked
0111	LDMSTL	Load MMX or 3D data with store check, locked
1000	ST	Store integer data
1001	STF	Store floating-point data
1010	STUPD	Store integer data and update base register
1011	STM	Store MMX or 3D data
1100	CDA	Check "data" effective address (segment and page protection)
1101	CIA	Check "instruction" effective address (segment protection only)
1110	TIA	TLB invalidate address (based on TLB index only)
1111	LEA	Load effective address

The LDL Op is a synonym for LD which is used as the first of a pair of LD Ops used to read an 8-byte segment descriptor. All LdStOps, except for ST, STF, and STM, produce a result that is stored in a general register. LdOps load a general register (the data register) with data from memory. For CDA, CIA, TIA, and LEA StOps, the data register is loaded with the calculated effective (logical) address. In both of these cases, the register modification size is based on the DSz field. For STUPD StOps, the base address register is loaded with the calculated effective (logical) address (i.e., "store and update"). The register modification size is also based on the ASz field instead of the DSz field.

The ASz[1:0] field specifies the address calculation size (in bytes) before and after the environment substitutions described above and according to the table on the following page:

Table 2.24 LdOp and StOp ASz(1:0) Field

ASz(1:0)	Address Calculation Size Before Environment Substitution	Address Calculation Size After Environment Substitution
00	Asize	2 Bytes
01	Ssize	—
10	4 Bytes	4 Bytes
11	Dsize	—

The DSz[1:0] field specifies the size of the data (in bytes) before and after the environment substitutions described above and according to the following two tables. The first table is for all LdOp and StOp operation Types other than LDF, STF, LDM, and CDAF(X):

Table 2.25 DSz(1:0) Field for LdStOps other than LDF STF, LDM, & CDAF

DSz(1:0)	Data Size Before Environment Substitution	Data Size After Environment Substitution
00	1 Byte	1 Byte
01	2 Bytes	2 Bytes
10	4 Bytes	4 Bytes
11	Dsize	—

The second table is only for the LdOp and StOp operation Types LDF, STF, LDM, and CDAF(X):

Table 2.26 DSz(1:0) Field for LdStOps LDF, STF, LDM & CDAF

DSz(1:0)	Data Size Before Environment Substitution	Data Size After Environment Substitution
00	FpDSize	—[a]
01	2 Bytes	2 Bytes
10	4 Bytes	4 Bytes
11	8 Bytes	8 Bytes

[a] LDF and STF only.

The Data[4:0] field specifies the general register to be used by an Op. The register field encodings can be found in Table 2.4 on page 92. If the Op is a LdOp, the specified register should be thought of as a “destination” register. If the Op is a StOp, it should be thought of as the “source” register of the data to be stored. The Seg[3:0] field specifies the segment register to be used by this Op for the “segment descriptor.”

Table 2.27 LdOp and StOp Seg(3:0) Field

Seg(3:0)	Register	Description
0000	ES	x86 architectural ES register
0001	CS	x86 architectural CS register
0010	SS	x86 architectural SS register
0011	DS	x86 architectural DS register
0100	FS	x86 architectural FS register
0101	GS	x86 architectural GS register
0110	HS	microarchitectural temporary segment register
0111	—	reserved
100x	TS	“descriptor table segment register” for accessing the x86 architectural global and local descriptor tables (GDT and LDT respectively)
1010	LS	“linear segment register” (i.e., the segment base = 0)
1011	MS	“special memory” memory segment register (the special memory contains the “scratchpad” memory, as well as other special address spaces—I/O, cache flush, and special bus cycles)
11xx	OS	effective x86 architectural operand segment register

The Base[3:0] field specifies the general register containing the base address operand for this Op. The Index[3:0] field specifies the general register containing the index address operand for this Op. The Base and Index register fields only refer to the low half of the general register set, i.e., to AX-DI and t0-t7. The mapping between the set of x86 architectural registers and the set of microarchitectural physical registers is discussed in detail in Chapter 3.

DESIGN NOTE

Expanded Segment Register Address Space

The segment register address space has been expanded to four bits, allowing additional special microarchitectural segment registers to be supported. The segment OS register is replaced when the environment substitution occurs by 0xxx, where xxx is the register number from the OpQuad Sequence environment.

Note the use of the scaled index in Figure 2.17 on page 166. The ISF[1:0] field specifies the index register scale factor according to the following table:

Table 2.28 LDOP AND STOP ISF(1:0) FIELD

ISF(1:0)	Index Register Scale Factor
00	1 X
01	2 X
10	4 X
11	8 X

The LD (large displacement) field specifies what field is to be used as the displacement according to the following table:

Table 2.29 LDOP AND STOP LD (LARGE DISPLACEMENT) FIELD

LD	Field to Use as the Displacement
0	Use the Disp8 field of this Op.
1	Use the 32-bit displacement from the appropriate decoder displacement bus

In order to understand what the choice means when LD = 1, we need to revisit Figure 2.8 on page 127. A modified version of it appears as Figure 2.11, below.

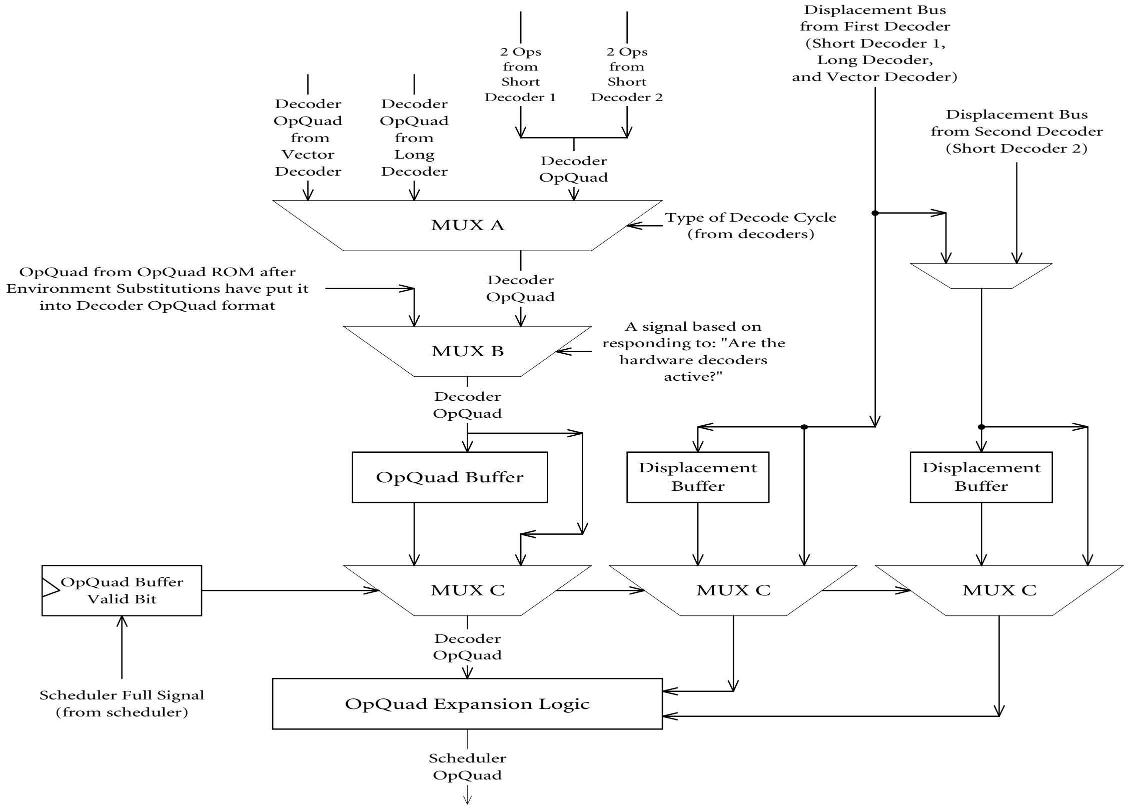

Decoder OpQuad from Vector Decoder
Decoder OpQuad from Long Decoder
2 Ops from Short Decoder 1
2 Ops from Short Decoder 2
Decoder OpQuad
Displacement Bus from First Decoder (Short Decoder 1, Long Decoder, and Vector Decoder)
Displacement Bus from Second Decoder (Short Decoder 2)
MUX A
Type of Decode Cycle (from decoders)
OpQuad from OpQuad ROM after Environment Substitutions have put it into Decoder OpQuad format
Decoder OpQuad
MUX B
A signal based on responding to: "Are the hardware decoders active?"
Decoder OpQuad
OpQuad Buffer
Displacement Buffer
Displacement Buffer
OpQuad Buffer Valid Bit
MUX C
MUX C
MUX C
Decoder OpQuad
Scheduler Full Signal (from scheduler)
OpQuad Expansion Logic
Scheduler OpQuad

Figure 2.11 DISPLACEMENT BUSES FROM DECODER

Whenever an instruction that specifies a long displacement is decoded by either the short decoders or the long decoder, that 32-bit (or sign-extended 16-bit) displacement is passed along with the decoder OpQuad through a multiplexer similar to MUX A. In order to keep Figure 2.11 simple, this multiplexer is not shown. If one of the short decoders or the long decoder is selected as the output of MUX A and if the decoder OpQuad is selected as the output of MUX B, then the 32-bit displacement is either:

1. loaded into the displacement buffer if the corresponding decoder OpQuad is loaded into the OpQuad buffer or
2. immediately sent to the OpQuad Expansion Logic via the displacement register if the corresponding decoder OpQuad is sent directly to the OpQuad Expansion Logic.

If LD = 1, this 32-bit displacement is then used in the OpQuad expansion and is stored in the DestVal field of the corresponding Op entry. The Disp8[7:0] field specifies an 8-bit value which is sign-extended to 32-bits when used as the displacement, otherwise it is ignored. The LD field specifies whether to use the Disp8 field or not.

RegOp Field Descriptions

RegOps perform register operations. They have the following format in a *decoder* OpQuad.

Table 2.30 Decoder OpQuad RegOp Format

37 36	35 30	29 26	25	24 20	21 17	16 12	11 10	9	8	7 0
0 0	Type	Ext	RX	DSz	Dest	Src1	—	SS	I	Imm8/Src2

We now describe each of the fields in this table: Type, Ext, RX, DSz, Dest, Src1, SS, I, and Imm8/Src2. The Type field specified the type of RegOp operation to be performed according to the encodings in the following four tables. Table 2.31 is for ALU (arithmetic logic unit) type operations:

Table 2.31 RegOp Type(5:1) Field General Arithmetic Operations

Type(5:1)	DSz Other Than 1 Byte	DSz = 1 Byte	cc-dep	RUX Only
00000	ADD, INC, CADD	ADD, INC	—	—
00001	MOV, OR, EOR	OR, EOR	—	O
00010	ADC	ADC	X	X
00011	SBB	SBB	X	X
00100	AND, EAND, BAND	AND, EAND	—	—
00101	SUB, ESUB, DEC, CSUB	SUB, DEC, ESUB	—	—
00110	XOR, EXOR	XOR, EXOR	—	—
00111	CMP	CMP	—	—

In the above table, Type[0] = 0 for all of the RegOp except for BAND for which Type[0] = 1.

DESIGN NOTE

Interpretation of RegOp Type Field

As is seen in Table 2.31, the RegOp Type field is interpreted differently based on the DSz field of the RegOp. One set of operations is implemented by hardware for byte-size operations, and a different set for 2-byte and 4-byte (i.e., 16-bit and 32-bit) operations. This distinction is made at Op execution time, after the DSz field has been resolved into an absolute Op size specification.

The "cc-dep" column means that the RegOp is dependent on a condition code as described in Chapter 3. All RegOps with Type = xx01x are treated by the hardware as being condition-code-dependent and are synchronized by the status operand forwarding logic discussed in Chapter 3. The "RUX Only" column means that the operation can only be executed in the RUX pipeline; see Figure 2.3 on page 81. Finally, the BAND ALU Op differs from AND Op in the generation of the CF status result.

The other RegOps sharing the same Type encoding only differ in the value put into the EXT field as the status modification bits as described later.

Table 2.32 is for Shift and MMX Ops. All MMX register operations share one RegOp Type. The individual MMX operations are distinguished by a separate MMX opcode (or operation type) that is passed to and decoded by the MMX execution units at Op execution time.

Table 2.32 REGOP TYPE(5:1) FIELD SHIFT AND MMX OPERATIONS

Type(5:1)	DSz Other Than 1 Byte	DSz = 1 Byte	cc-dep	RUX Only
01000	SLL, BLL	SLL	—	X
01001	SRL	SRL	—	X
01010	SLC, RLC	—	X	X
01011	SRC, RRC	—	X	X
01100	SLA	SLA	—	X
01101	SRA	SRA	—	X
01110	SLD, RLD	RLS	—	X
01111	SRD, RRD, MMX	RRS	—	—

In the above table, Type[0] = 0 for all of the Shift and MMX Ops except for BLL and MMX for which Type[0] = 1.

Design Note

Difference in BLL and SLL Op

The BLL Shift Op differs from the SLL Op in that the masking of the shift amount is DSz-dependent.

The third table is for other arithmetic RegOps:

Table 2.33 RegOp Type(5:1) Field for Other Arithmetic RegOps

Type(5:1)	DSz Other Than 1 Byte Type(0)= 0	DSz Other Than 1 Byte Type(0)=1	DSz = 1 Byte Type(0)= 0	DSz = 1 Byte Type(0)=1	cc-dep	RUX Only
10000	ZEXT8	SEXT8	—		—	X
10001	ZEXT16	SEXT16	—		—	X
10010	RDFLGS		DAA	DAS	X	X
10011	MOVcc	MOVcc	—		X	X
10100	MUL1S	MUL1U	—		—	X
10101	MULEH	MULEL	—		—	X
10110	DIV1	DIV2	—		—	X
10111	DIVER	DIVEQ	—		—	X

Design Note

ZEXT8, SEXT8, ZEXT16, and SEXT16 RegOp

For the ZEXT8, SEXT8, ZEXT16, and SEXT16 RegOps, only the low one or two bytes of the source operand is required to be valid, even if DSz is two or four bytes. Further, for the ZEXT8 and SEXT8 RegOps, the source operand register number is interpreted as specifying a byte register. Note, the other source operand is not used by these Ops.

The fourth and last RegOp Type table is for special RegOps:

Table 2.34 REGOP TYPE(5:1) FIELD FOR SPECIAL REGOPS

Type(5:1)	DSz Other Than 1 Byte Type(0)= 0	DSz Other Than 1 Byte Type(0)=1	DSz = 1 Byte Type(0)= 0 Type(0)=1	cc-dep	RUX Only
11000	RDxxx	RDxxx	—	—	X
11001	RDFLG	BSWAP	—	—	X
11010	RDSEG	RDSEG	—	X	X
11011	—		—	—	—
11100	WRDR	WRDL	—	—	X
11101	WRxxx	WRxxx	—	—	X
11110	WRIP	WRDLIP	—	—	X
11111	CHKS	WRDH	—	—	X

DESIGN NOTE

Valid Source Operand Bytes for CHKS and WRDR

Only the low two bytes of the source operands are required to be valid for the CHKS and WRDR RegOps, even if DSz is equal to four bytes.

The Ext[3:0] (Extension) field combines with or extends the meaning of the Type field as follows:

Table 2.35 REGOP EXT(3:0) FIELD

Type of Ops	Field Combination	Used to Specify
MOVcc	Type[0] Ext[3:0]	a 5-bit condition code
RDxxx and WRxxx	Type[0] Ext[3:0]	a 5-bit special register number
RDSEG	Type[0] Ext[3:0]	a 5-bit segment descriptor register number
(other) Ops with SS = 1	Ext[3:0]	four status flag modification bits stored in the scheduler
(other) Ops with SS = 0	Ext[3:0]	not used

For the WRFLG WRxxx Op, the special register number specifies the x86 EFLAGS register. The low four bits of this register number is also identical to the desired four status modification bits for loading the status flag bits of EFLAGS, when SS = 1.

The DSz[2:0] field specifies the size of the data (in bytes) for the Op according to the following table:

Table 2.36 RegOp DSz(2:0) Field

DSz(2:0)	Data Size Before Environment Substitution	Data Size After Environment Substitution
000	1 Byte	1 Byte
001	2 Bytes	2 Bytes
010	4 Bytes	4 Bytes
011	Dsize	—
100	Asize	—
101	Ssize	—
110	—	—
111	—	—

The Dest[4:0] field specifies which general register the Op will use to store its results in (i.e., the destination register). The Src1[4:0] field specifies which general register the Op will use to obtain its first operand from (i.e., the source register Src1). The SS field (Set Status field) specifies whether the Op affects status flags. If it does, then the status modification bits in the Ext field indicate which groups of flags are affected. The I field (Immediate field) indicates if the second operand of the Op is an immediate value or is obtained from a general register. It is interpreted in conjunction with the Imm8/Src2 field according to the following table:

Table 2.37 RegOp I (Immediate) Field

I Field	Imm8/Src2	Source for Second Operand
0	Imm8/Src2[4:0]	a general register to be the source register Src2
1	Imm8/Src2[7:0]	an 8-bit immediate value that is extended to the DSz size

Lastly, bits [11:10] of a RegOp are unused and should be set to "00".

SpecOp Field Descriptions

SpecOps perform a variety of special operations. They have the following format in a decoder OpQuad:.

Table 2.38 Decoder OpQuad SpecOp Format

37 35	34 31	30 26	25 24	23 22	21 17	16 0
1 0 1	Type	CC	—	DSz	Dest	Imm17

We now describe each of the fields in this table: Type, CC, DSz, Dest, and Imm17. The Type field specifies the type of SpecOp to be performed according to the following table:

Table 2.39 SpecOp Type(3:0) Field

Type(3:0)	Op Symbol	Type of SpecOp
00xx	BRCOND	Branch condition
0100	LDDHA	Load default handler address
0101	LDDHAB	Load binary default handler address
0110	LDAHA	Load alternate handler address
0111	LDAHAB	Load binary alternate handler address
1000	LDK	Load constant
1001	FPOP	Floating-point Op
1010	LDKD	Load DSz-modified constant
1011	FPOPE	Floating-point Op from the OPQuad Sequence Environment
11xx	FAULT	Unconditional fault

As discussed in the section titled "The Op Sequence and Action Fields" beginning on page 141, every OpQuad *must* specify a branch, even if it's only a branch to the next sequential instruction in the ROM. It was also noted that branches in OpQuad sequences are specified by a combination of the SpecOp Type field (discussed above) and the Op Sequence and Action Fields shown in Figure 2.10 on page 139.

Further, the OpQuad may or may not also contain a BRCOND Op. If there is a BRCOND Op, then the OpQuad sequence branch is considered to specify a conditional branch, otherwise it is considered an unconditional branch. The exception to this that was mentioned is the subroutine branch (BSR), which is always unconditional, yet always requires a BRCOND Op in the OpQuad to supply the return address (remember that OpQuad sequence subroutines need not return to the OpQuad following the caller).

You can have four Ops plus a branch in a single OpQuad if the branch is unconditional, but only three Ops plus a branch in an OpQuad if the branch is conditional (or is a BSR branch). In tabular form, this discussion can be summarized as follows:

More Detailed Analysis of OpQuad Sequence Branch Types

Action Field	BRCOND in OpQuad	Branch Type
00	No	BR
00	Yes	BRcc
01	No	*Illegal*
01	Yes	BSR
10	No	ERET
10	Yes	ERETcc
11	No	SRET
11	Yes	SRETcc

The CC[4:0] field specifies the particular condition to be tested for in the case of BRCOND SpecOps. CC[4:1] specifies the condition to be tested according to the following table: CC[0] specifies whether the condition or its complement is to be tested. CC[0] = 1 complements the condition.

Table 2.40 SpecOp CC(4:1) Field

CC(4:1)	Mnemonic	Condition to be Tested	Usage
0000	True	1	Always TRUE
0001	ECF	ECF	OpQuad Sequence Carry Flag
0010	EZF	EZF	OpQuad Sequence Zero Flag
0011	SZnZF	EZF | ~ZF	Early termination of string instructions due to debug trap or hardware interrupt
0100	MSTRZ	~EZF & ~IP & ~ (DTF | SSTF | MDD)	String instruction exit condition
0101	STRZ	~EZF & ~IP & ~ (DTF | SSTF | MDD)	String instruction exit condition
0110	MSTRC	~EZF & ~IP & ~ (DTF | SSTF | MDD)	String instruction exit condition
0111	STRZnZF	~EZF & ~IP & ~ (DTF | SSTF | MDD) & ZF	String instruction exit condition

Table 2.40 SpecOp CC(4:1) Field

CC(4:1)	Mnemonic	Condition to be Tested	Usage
1000	OF	OF	Overflow Flag
1001	CF	CF	Carry Flag
1010	ZF	ZF	Zero Flag
1011	CvZF	CF | ZF	Used for "below or equal", "not below or equal", "above", "not above" conditions
1100	SF	SF	Sign Flag
1101	PF	PF	Parity Flag
1110	SxOF	SF ^ 0F	Used for "less", "not less", "greater or equal", "not greater or equal" conditions
1111	SxOvZF	(SF ^ 0F) | ZF	Used for "greater", "not greater", "less or equal", "not less or equal" conditions

The DSz[1:0] field specifies the size of the data (in bytes) for the SpecOp according to the following table:

Table 2.41 SpecOp DSz(1:0) Field

DSz(1:0)	Data Size Before Environment Substitution	Data Size After Environment Substitution
00	1 Byte	1 Byte
01	2 Bytes	2 Bytes
10	4 Bytes	4 Bytes
11	Dsize	—

The Dest[4:0] field specifies the general register the SpecOp will use to store its results (i.e., the destination register).

The Imm17[16:0] field is used as either (a) a 17-bit signed constant (for LDK or LDKD SpecOps), (b) a 14-bit Op address (for BRCOND or LDxHAx SpecOps), or (c) to specify a specific type of FpOp for FPOP SpecOps.[20] In the latter two cases, only bits [13:0] of this field are used. Lastly, bits [25:24] of a SpecOp are unused and should be set to "00".

[20] Imm7 is not used for FPOPE SpecOps as the OpQuad sequence environment specifies the specific type of FpOp.

LIMM Op Field Descriptions

LIMM Ops perform a 32-bit load immediate operation. They have the following format in a decoder OpQuad.

Table 2.42 Decoder OpQuad LIMM Op Format

37 36	35 21	20 17	16 0
1 1	ImmHi	Dest	ImmLo

Bits [37:36] indicate a LIMM Op; there is no further Type field. ImmHi[15:0] and ImmLo[15:0] together specify a 32-bit immediate value. Dest[3:0] is a 4-bit destination register specifier. Like LdStOp base and index registers, this can only specify the low half of the full 5-bit register space.

Design Note

Definition of a NoOp

Register t0 serves as a "bit bucket" for writes as it always returns "0" for reads. A NoOp is defined as a LIMM Op into t0.

Execution Pipelines

The LU, SU, RUX, and RUY execution units are all implemented as multistage pipelines. The LU execution unit is a six-stage pipeline. The SU execution unit is a seven-stage pipeline. The RUX and RUY execution units are either six- or seven-stage pipelines, depending on the type of Op being executed. Each stage in the pipelines nominally requires one processor clock cycle. An Op can be held up in one of the stages for stage-specific reasons or because an Op is held up in the next consecutive pipeline stage. For most Ops, the Op's position in the scheduler is independent of the Op's stage of execution in one of the pipelines.

An overview of the design and implementation of the RUX and RUY pipelines is given as an example of various microarchitectural characteristics of the K6 3D that come into play in the design and implementation of the execution unit pipelines. Both RUX and RUY have identical pipelines, which we will call the RUX/RUY pipeline" or the "RegOp pipeline." Figure 2.12 is used to describe the RUX/RUY pipeline for integer RegOps and single-cycle MMX Ops.

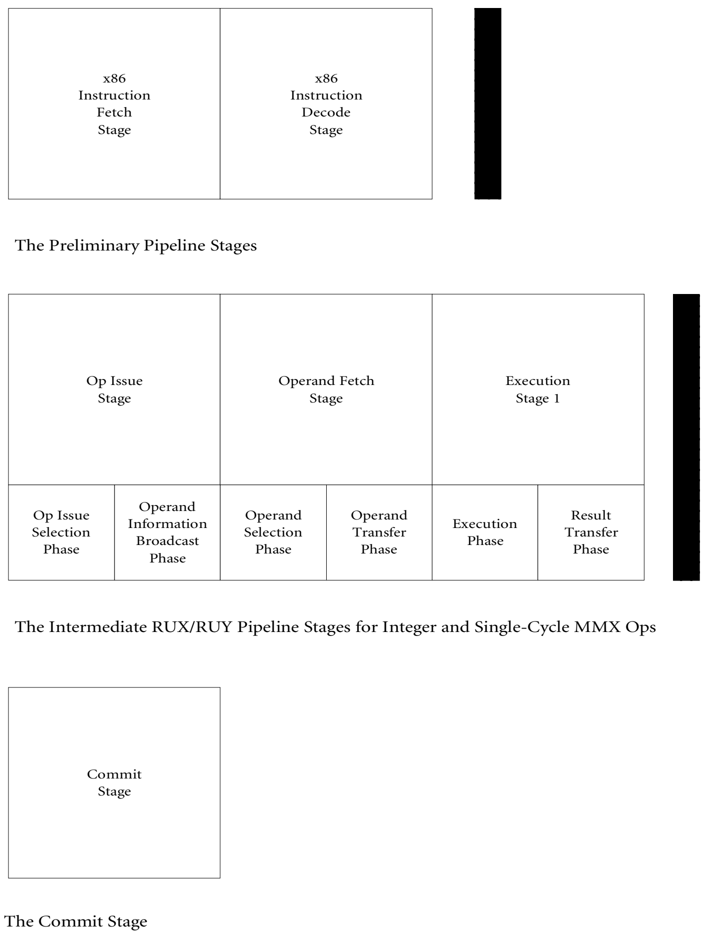

x86 Instruction Fetch Stage
x86 Instruction Decode Stage
The Preliminary Pipeline Stages
Op Issue Stage
Operand Fetch Stage
Execution Stage 1
Op Issue Selection Phase
Operand Information Broadcast Phase
Operand Selection Phase
Operand Transfer Phase
Execution Phase
Result Transfer Phase
The Intermediate RUX/RUY Pipeline Stages for Integer and Single-Cycle MMX Ops
Commit Stage
The Commit Stage

Figure 2.12 INTEGER AND SINGLE-CYCLE MMX RUX/RUY PIPELINE STAGES

A pipeline consists of one or more pipeline stages. Figure 2.12 shows the RegOp pipeline having six pipeline stages: two preliminary pipeline stages, three intermediate or *execution* stages, and a *results* commit stage. Moreover, there is the normal buffering that one would find in between the stages of a pipeline implementation. Results from one stage are captured in pipeline registers or latches for use in the next stage. A typical connection between showing the pipeline register explicitly could have been represented by the following diagram.

RegOp pipeline
preliminary pipeline stages
intermediate stages
commit stage
pipeline registers
pipeline latches

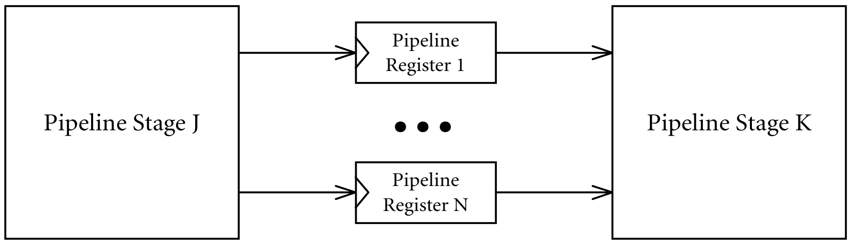


Figure 2.13 GENERIC PIPELINE STAGES

Explicit representation of these registers, as well as other inputs and outputs to various pipeline stages, would have made Figure 2.12 (and the other pipeline figures in this section) overly complex for the current discussion and therefore we did not show them. Each boundary between adjacent stages represents as many pipeline registers as are needed to hold and transfer information to the next stage. Some of these registers are explicitly shown in more detailed figures for the LU and SU pipelines that appear later in this section. The global control logic, shown in Figure 2.9 on page 130, is directly involved with controlling the loading of the pipeline registers.

We emphasize the buffering between the two preliminary pipeline stages and the three intermediate stages and the buffering between three intermediate stages and the commit stage by the dark vertical bars in the Figure 2.12. We do this to draw your attention to the fact that the buffering in between the preliminary pipeline stages and the intermediate pipeline stages is part of the support for the *decoupled decode/execution structure* of the microarchitecture. Similarly, the buffering in between the preliminary pipeline stages and the commit stage is part of the support for the *decoupled execution/commitment structure* of the microarchitecture. Both of these types of decoupling were discussed earlier in this chapter, see "Decoupled Decode and Execution Decoupled Execution and Commitment" on page 77.

instruction fetch stage
x86 instruction decode stage
commit stage

The x86 Instruction Fetch Stage, the x86 Instruction Decode Stage, and the Commit Stage are common to all execution unit pipelines. During the x86 Instruction Fetch Stage, up to 16 bytes of x86 instructions and associated predecode bits are fetched from the L1 I-Cache into the instruction buffer. During the x86 Instruction Decode Stage the decoders decode up to two x86 instructions out of the instruction buffer and form an OpQuad that is loaded into the top row of the scheduler's buffer.

Historical Comment and Suggested Readings

Design and Implementation of High-Performance Pipelines

There is a rich history of the design and implementation of very high-performance pipelines. Indeed, the use of deep pipelines (often termed superpipelines) is high on the list of design alternatives of both architects and microarchitects. The basic elements in pipeline design are the chunks of logic to do the computation within each stage, the mechanism to capture the outputs of one stage for use in the next stage, and the approach used to control the pipeline. Clock signals are typically used to synchronize the inputs to the stages and inhibit signals are used to halt the flow within a pipeline.

The partitioning of the functions to be performed within the pipeline into specific stages, the subsequent control of the stages, and the pipeline's interface to the memory and caching hierarchy are the most important parts of pipeline design at the microarchitectural level. For excellent treatments of these and related issues see Harold Stone's book, *High Performance Computer Architecture*, 3rd Edition, Addison Wesley, 1993, pp. 143-148 and pp. 169-192. Harvey Cragon's book, *Memory Systems and Pipeline Processors*, Jones and Bartlett Publishers, *1996*, pp. 294-309 and Michael Flynn's *Computer Architecture,* Chapters 4 and 7, Jones and Bartlett Publishers, *1995*. Issues such as how to deal with pipeline hazards (stalls), exceptions, and traps and how to support operand forwarding must factor into the design of the pipeline. The overall objective is to achieve as simple a design as possible while meeting the cost/performance goals.

The analysis to determine the number of pipeline stages to employ runs along the following line. The performance of the pipeline basically rests on the designer's ability to break up the functions to be performed into stages of equal duration. The greater the number of stages in the pipeline means less work per stage. Less work per stage means fewer levels of logic per stage. Fewer levels of logic means a faster clock. A faster clock means faster program execution. For deep pipelines (i.e., those with many stages) this analysis must be tempered by the facts that: (a) a long delay to both flush and fill the pipeline occurs, impacting the overall performance and (b) the memory and cache hierarchy interface must match the higher performance of the pipeline. For short pipelines (i.e., those with few stages) the analysis must be tempered by the facts that: (a) the stages are more complex and require more levels of logic to implement, particularly during mispredicted branches and (b) the clock will be slower. Importantly, regardless of whether a short, intermediate, or deep approach is taken, the memory and cache interface needs to support the total number of operand and results accesses required by the pipeline performance in a non conflicting way and at the required rate and data bandwidth. It should not come as too much of a surprise that increasing the number of stages does not always increase the overall performance of the processor. What this means is that the pipelines and the memory and cache hierarchies must be designed together. However, this is not the end of the story. The actual performance of the implementation of a pipeline can depend heavily on variations in logic circuit delays and in clock skew. Thus, attention to the physical circuit interconnections, the electrical characteristics of the circuits, the effect of loading on the components, and the techniques employed to distribute the clock signal must often be taken into consideration,. The implementation solutions employed may very well impact the design of the pipeline at the microarchitectural level, (e.g., such implementation considerations might dictate whether or not pipeline latches are actually used at specific stage boundaries or for all of the outputs at a given boundary). This is another example of where the microarchitects and the logic designers must work closely together.

Additional readings: Peter M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill Book Company, 1981 and Harvey Cragon, cited above, both give brief but interesting histories of the use of pipelines in processors. The LARC, co-designed by IBM and Univac and delivered to Lawrence Livermore Laboratory in 1959, had a four-stage pipeline. Descriptions of a number of pipelined machines can be found in C. Gordon Bell and Allen Newell, *Computer Structures: Readings and Examples*, McGraw-Hill Book Company, 1971, and Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, 1982.

Op issue stage
operand fetch stage

The scheduler then controls the Op Issue Stage and the Operand Fetch Stage for all LdOps, StOps, RegOps other than branch operations (BrOps) and floating-point operations (FpOPs). During the Op Issue Stage, which consists of an Op Issue Selection Phase and an Operand Information Broadcast Phase, the scheduler scans the entries in its buffer (instruction window) and issues up to six Ops to appropriate execution units if an unissued Op for the type of execution unit is available.

Operands for the Ops issued during the Op Issue Stage are forwarded to the execution units in Operand Fetch Stage which consists of an Operand Selection Phase and an Operand Transfer Phase. For many types of RegOp, the operation completes in the one clock cycle identified as Execution Stage 1, which, as seen in Figure 2.12 on page 159, consists of:

1. an Execution Phase in which the integer, MMX, or 3D execution sub-units within the register execution processes the source operands of the RegOp (according to the type of RegOp being executed).
2. a Result Transfer Phase in which result values and status flag values from one of the register execution sub-units are stored back in the scheduler entry corresponding to the RegOp being executed.

Register and status flags result values, stored in the Op's scheduler entry, are subsequently committed to the architectural register file and the architectural status flag register if and when it is safe to do so. Internal microarchitectural state is also changed as appropriate. After an Op completes and there are no preceding exceptions or mispredicted branches the following actions occur during the Commit Stage:

1. the Op's results can be committed.
2. the Op can be retired by moving the OpQuad containing the Op out of the scheduler's buffer, once all of the Ops within the OpQuad are ready to be retired.

The register and status flag result values from an Op can be used by the scheduler as operands for execution of dependent Ops between the execution completion and results commitment times of that Op.

LdOp pipeline
StOp pipeline
BrOp pipeline

This brief introduction to the K6 3D pipelines and their operation will be continued later in this and the next chapter. A diagram for 2-Cycle MMX and 3D RegOps is given in Figure 2.14.

Diagrams for the LU pipeline (also called the LdOp pipeline), the SU pipeline (also called the StOp pipeline), and the BRU pipeline (also called the BrOp pipeline) are shown as Figure 2.15 through Figure 2.21 on page 175 respectively.

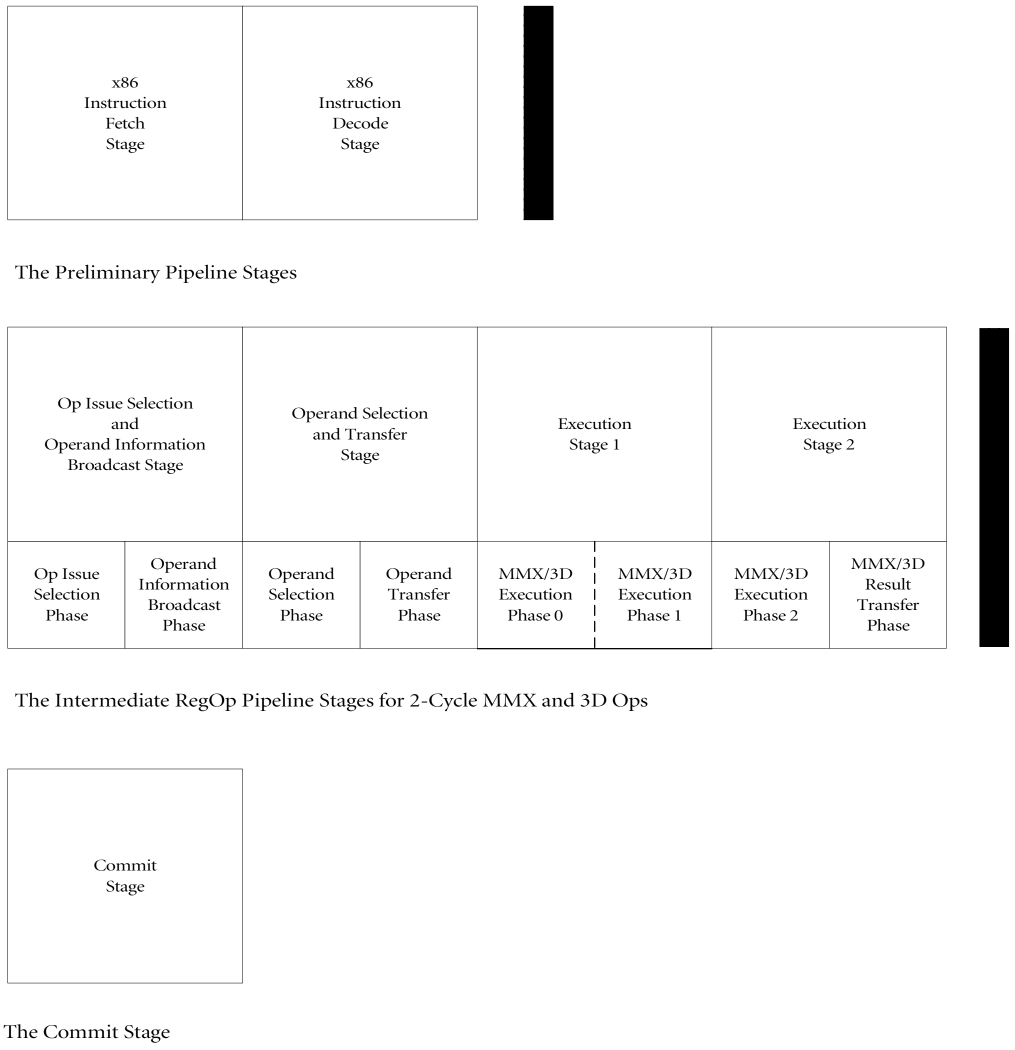

x86 Instruction Fetch Stage
x86 Instruction Decode Stage
The Preliminary Pipeline Stages
Op Issue Selection and Operand Information Broadcast Stage
Operand Selection and Transfer Stage
Execution Stage 1
Execution Stage 2
Op Issue Selection Phase
Operand Information Broadcast Phase
Operand Selection Phase
Operand Transfer Phase
MMX/3D Execution Phase 0
MMX/3D Execution Phase 1
MMX/3D Execution Phase 2
MMX/3D Result Transfer Phase
The Intermediate RegOp Pipeline Stages for 2-Cycle MMX and 3D Ops
Commit Stage
The Commit Stage

Figure 2.14 2-Cycle MMX and 3D RUX/RUY Pipeline Stages

They are included here, with some brief remarks, to let you see some of the differences in the various pipelines. Both the LU pipeline and SU pipeline are shown in more detail immediately after their high-level pipeline diagram. Further explanations of these figures are in Chapter 3.

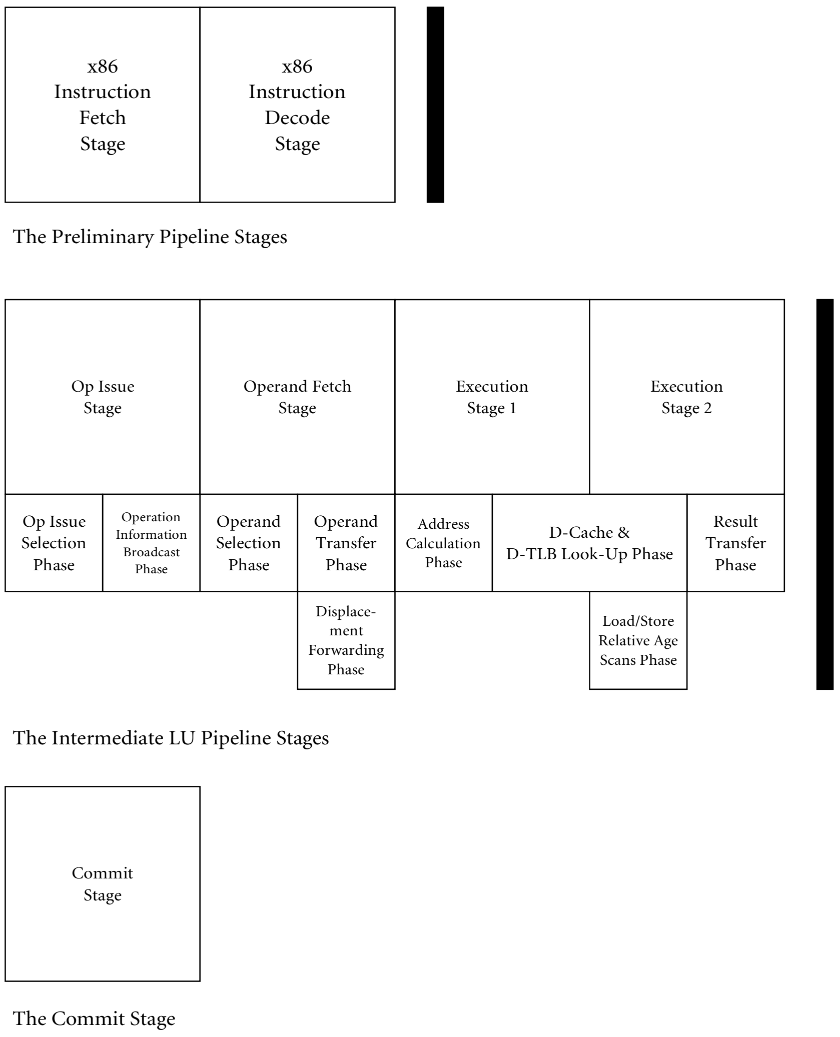

x86 Instruction Fetch Stage
x86 Instruction Decode Stage
The Preliminary Pipeline Stages
Op Issue Stage
Operand Fetch Stage
Execution Stage 1
Execution Stage 2
Op Issue Selection Phase
Operation Information Broadcast Phase
Operand Selection Phase
Operand Transfer Phase
Address Calculation Phase
D-Cache & D-TLB Look-Up Phase
Result Transfer Phase
Displace-ment Forwarding Phase
Load/Store Relative Age Scans Phase
The Intermediate LU Pipeline Stages
Commit Stage
The Commit Stage

Figure 2.15 LU Pipeline Stages

Both Figure 2.16 on page 165 and Figure 2.19 on page 168 contain a shaded box which is labeled “Sources of Segment Base, Base, Index, and Displacement.” The logic contained in this box is the same for both of these figures and is shown separately in Figure 2.17. The logic identified as “SLC” in Figure 2.16 refers to the Segment Limit Violation Check logic which is discussed later in this section.

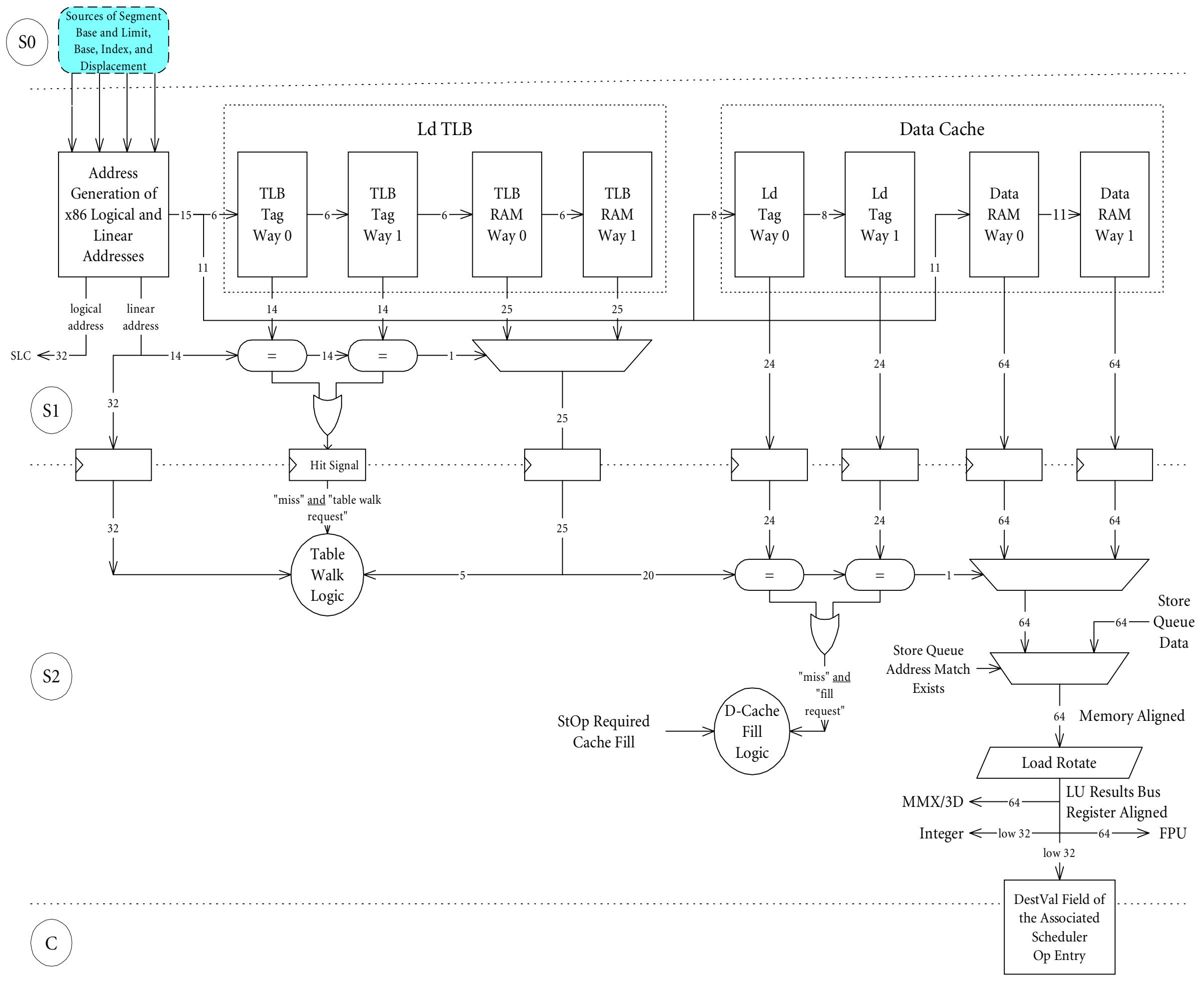


Figure 2.16 The LU Intermediate or Execution Pipeline Stages

We have introduced some textual abbreviations in Figure 2.16, Figure 2.19 on page 168, and Figure 2.20 on page 169 to reduce the visual clutter in these three already crowded figures. Using Si to stand for stage and Ci to stand for commit, we introduce the notation Si and Ci at the left-hand side of these figures.

S = stage
C = commit

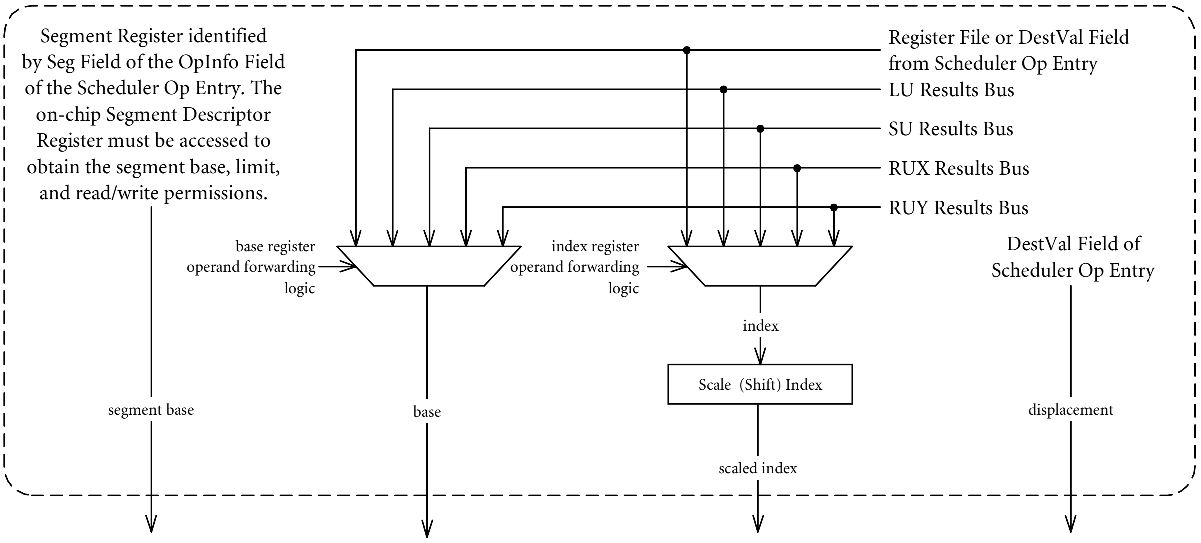

Segment Register identified by Seg Field of the OpInfo Field of the Scheduler Op Entry. The on-chip Segment Descriptor Register must be accessed to obtain the segment base, limit, and read/write permissions.
Register File or DestVal Field from Scheduler Op Entry
LU Results Bus
SU Results Bus
RUX Results Bus
RUY Results Bus
base register operand forwarding logic
index register operand forwarding logic
DestVal Field of Scheduler Op Entry
index
Scale (Shift) Index
segment base
base
displacement
scaled index

Figure 2.17 Sources of Segment Base and Limit, Scaled Index, and Displacement Values

The notations correspond in the following fairly obvious way:

Table 2.43 Pipeline Notational Correspondence

Figure 2.16 and Figure 2.19	Figure 2.12, Figure 2.14 and Figure 2.18
S0	Operand Fetch Stage
S1	Execution Stage 1
S2	Execution Stage 2
C	Commit Stage

The use of C1, C2, and C3 in Figure 2.20 on page 169 reflects the fact that the overall Commit Stage for StOps is composed of several "stages." We will find this useful when discussing the operation of the store queue commit L1 D-Cache Access logic. The registers shown in Figure 2.16, Figure 2.19, and Figure 2.20 in between the pipeline stages (i.e., S0, S1, S2, C, C1, and C2) are the pipeline registers discussed earlier in this section.

DESIGN NOTE

Dual Ported TLB

The D-TLB shown in Figure 2.2 on page 69 and Figure 2.22 on page 180 is actually designed to be “dual ported,” i.e., to translate both a load address and a store address per cycle. Thus it is shown in Figure 2.16 on page 165 and Figure 2.19 on page 168 as logically consisting of two TLBs, the Ld-TLB and the St-TLB, that are copies of each other.

The three SU pipeline figures corresponding to the three LU pipeline figures follow:

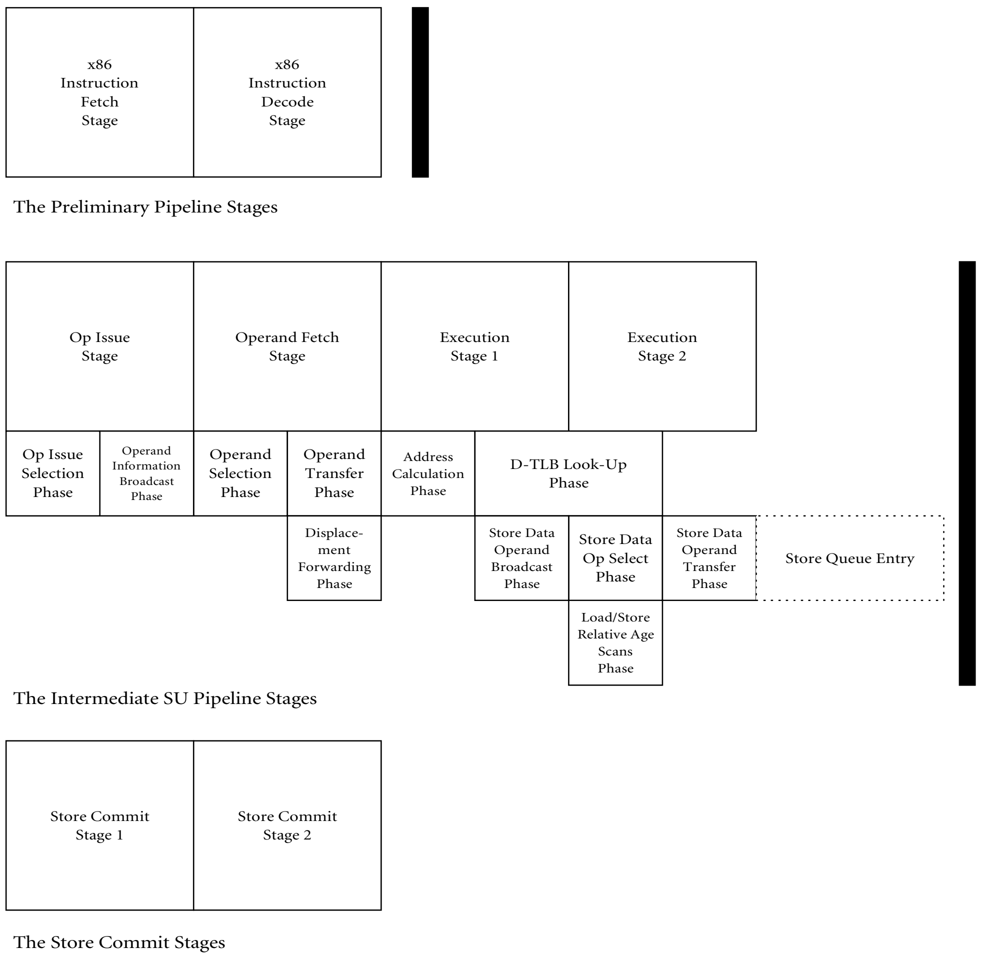

x86 Instruction Fetch Stage
x86 Instruction Decode Stage
The Preliminary Pipeline Stages
Op Issue Stage
Operand Fetch Stage
Execution Stage 1
Execution Stage 2
Op Issue Selection Phase
Operand Information Broadcast Phase
Operand Selection Phase
Operand Transfer Phase
Address Calculation Phase
D-TLB Look-Up Phase
Displace-ment Forwarding Phase
Store Data Operand Broadcast Phase
Store Data Op Select Phase
Store Data Operand Transfer Phase
Store Queue Entry
Load/Store Relative Age Scans Phase
The Intermediate SU Pipeline Stages
Store Commit Stage 1
Store Commit Stage 2
The Store Commit Stages

Figure 2.18 SU PIPELINE STAGES

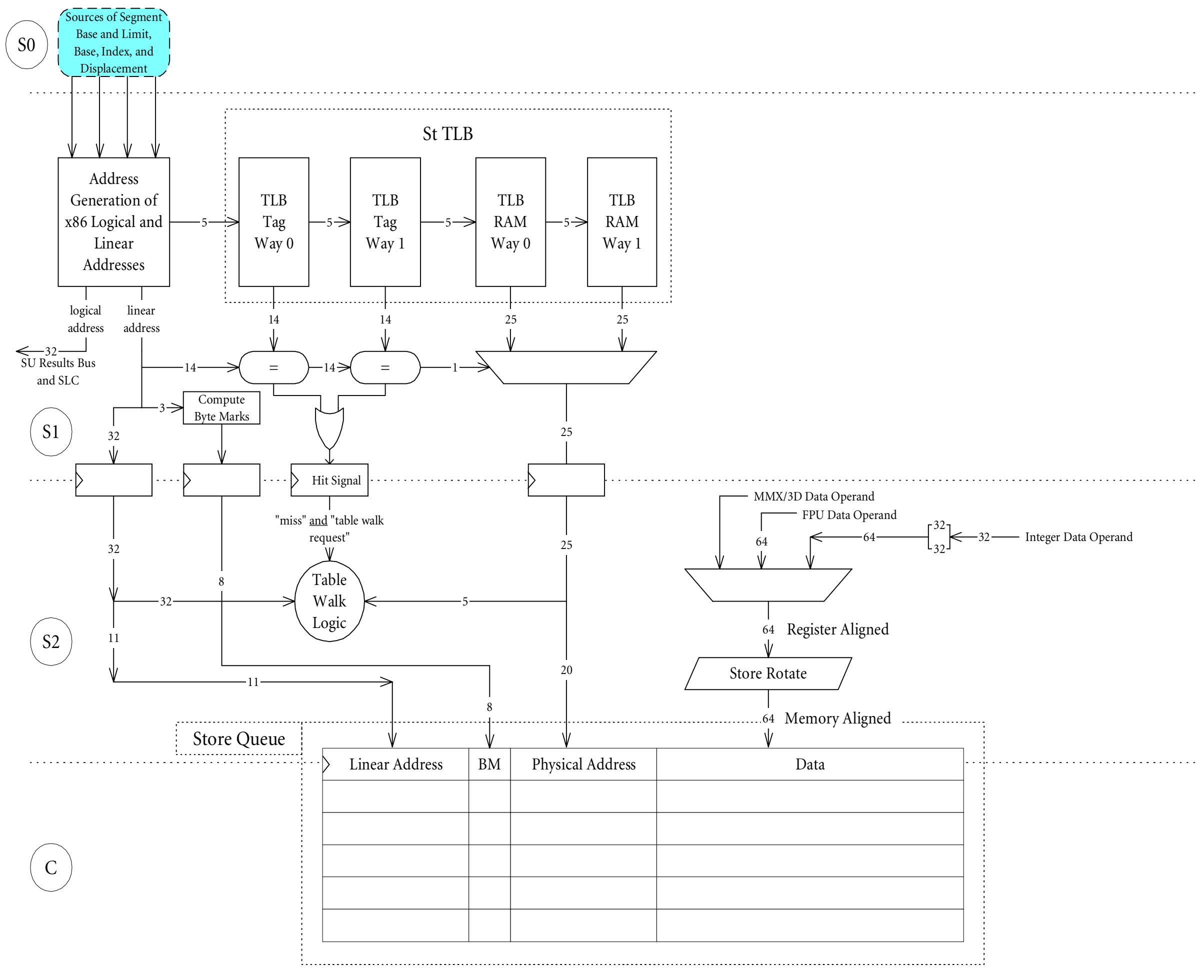


Figure 2.19 THE INTERMEDIATE OR EXECUTION SU PIPELINE STAGES AND STORE QUEUE ACCESS

FAULTS, TRAPS, ABORT CYCLES, AND THE PIPELINES

When the execution units were introduced, we discussed how the K6 deals with exceptions, traps, and abort cycles. The mechanisms employed are extremely important in the design of the pipeline. In fact, there are those who believe that these issues are among the difficult ones to resolve and are central to "what makes pipelining hard to implement," see Patterson and Hennessy, *Computer Architecture: A Quantitative Approach*, 2nd Edition, Morgan Kaufmann Publishers, Inc. 1996, pp. 178-187. You may want to review the section titled "Status Flags, Faults, Traps, Interrupts, and Abort Cycles" beginning on page 83 before proceeding. We will expand on our explanations there by taking a look in more detail at how the K6 deals with these issues by looking at how it handles a misaligned access while executing a LdOp or a StOp.

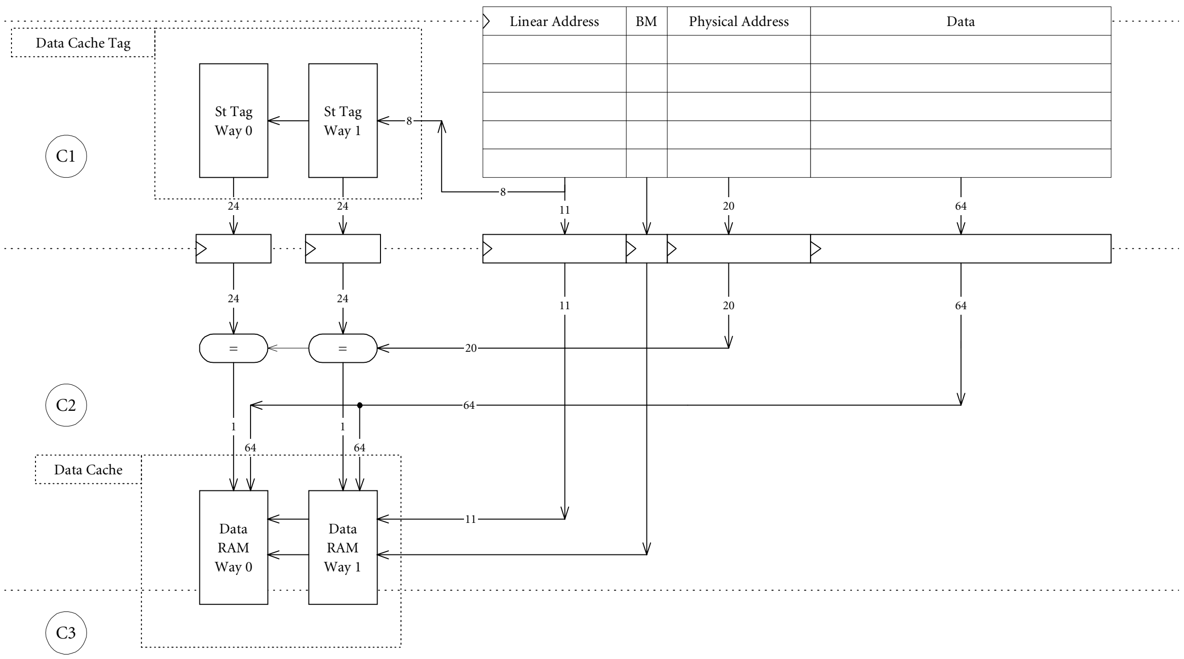


Figure 2.20 Store Commit Pipeline Stages

In the x86 instruction set architecture, a misaligned access occurs either when an 8-byte (a quadword) access is made to an address that is not on an 8-byte boundary, or when a 2-byte (a word) or a 4-byte (a double word) access is made to an address that is not on a 4-byte boundary. We'll first look at how traps are handled and then examine the misaligned access. The x86 instruction set architecture has an alignment check bit which, if set, enables the generation of a misaligned access fault when a misaligned access occurs. This feature is generally disabled as it has limited, if any, use under contemporary operating systems.

Fault and Trap Handling

When a fault (such as a segment violation or page fault exception) occurs during the execution of a LdOp or a StOp, the OCU is immediately notified and an abort cycle typically is eventually initiated for the associated Op. Later, we will discuss faults that do not cause an abort cycle. The abort cycle results in a fault handler getting invoked which determines what caused the fault and then initiates an appropriate response. Traps, in contrast to the treatment of faults, are handled differently.

When a trap occurs, trap information is loaded into the scheduler in the entry associated with the Op that caused the trap. Later, when the OCU analyzes if it can commit that Op, it recognizes that the Op caused a trap and it sets a pending trap flip-flop. In effect, traps are accumulated as pending traps until the end of an instruction is reached. Recall that instructions decoded by either of the short decoders or the long decoder produce at most one OpQuad and instructions that are decoded by the vector decoder produce OpQuad sequences that end with an "ERET" Action Field value within the sequencing field of its last OpQuad.

The OCU can recognize that it is retiring the last of all of the Ops that are associated with a given instruction. If there are any pending traps at the end of the commitment of an instruction, the OCU will initiate an exception (i.e., a fault) at the beginning of the next instruction—i.e., an abort cycle will occur at the beginning of the next instruction whenever the pending trap flip-flop is set. This discussion points out that the abort cycle is central to the handling of both faults and traps. So, let's take a closer look at it.

LdOp Abort Cycles

Typically a LdOp, like all Ops, finishes execution successfully. But there are a number of faults that can arise that cause the LdOp to be "held up" in the last execution stage of the LU pipeline. Whenever one of these situations occurs, the LU sends a signal to the OCU indicating that there is a LdOp held up in Stage2 of its pipeline because the Op has a violation associated with it.

Eventually, when the OCU examines the Ops in the bottom OpQuad of the scheduler, it attempts to commit the LdOp for which the LU has sent the violation signal. The OCU then recognizes that the LdOp is not going to complete. It will retire any older Ops in the OpQuad and initiate an abort cycle which will flush both the upper and lower portions of the machine and vector the processor to a fault handler address in the OpQuad ROM. Simultaneously, a snapshot of information in the LU is written into the OCU's Fault ID Register. This information is readable from SR1[2:0] by the OCU, i.e., bits [2:0] of Special Register 1 (see Table 2.6 on page 98). The fault handler OpQuad Sequence examines the fault information in the SR1 through the use of a special RegOp that allows the Special Registers to be read by an OpQuad sequence to determine what caused the fault.[21] Based on this, the fault handler branches to an appropriate OpQuad sequence that will process the specific type of fault encountered.

Fault ID register

[21] For some interrupts and "exceptions," an x86 error code is constructed and ultimately placed on the architectural interrupt stack (see, for example, Chapter 5 in Intel's publication, *The Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*).

Design Note

Default Handler Address and Alternate Handler Address

As explained in Chapter 3, concurrent with the abort cycle, the OCU vectors the machine to one of two possible OpQuad sequence entry point addresses—either the OCU default handler address or an OCU alternate handler address; see Figure 2.10 on page 139. The setting of these addresses is supported by the LDDHA Op (Load Default Handler Address) and the LDAHA Op (Load Alternate Handler Address). The default fault handler address is initialized by the Reset OpQuad Sequence and the alternate handler address is specified within some instruction and exception processing OpQuad sequences.

Now that we have explained how faults and traps are handled and what occurs during an abort cycle, we can examine how the misaligned access is handled.

LdOp Misaligned Accesses

Since the size of the access is known when the LdOp begins execution, it can be determined if an access is misaligned when the computed address is available at the end of the address calculation phase of Stage1 of the LU pipeline.

If the access is misaligned, the hardware in LU pipeline Stage1 splits the LdOp into a cloned pair of LdOps. One of the cloned LdOps handles the first half of the access (the lower byte addresses) while the other cloned LdOp handles the upper half of the access (the high byte addresses). Thus, the two cloned LdOps have different addresses and access sizes. The LdOp accessing the lower half proceeds into Stage2 while the LdOp accessing the upper half is left immediately behind it in Stage1. The two cloned LdOps now progress down the LU pipeline back-to-back and get processed individually—(e.g., each LdOp will get separately translated by the Ld-TLB, and each will get separately looked up in the D-Cache). The two LdOps are guaranteed to access data within neighboring aligned octets of memory. The assembly of the output of the back-to-back accesses into the originally required access is done in the assembly buffer register on the output of the 2:1 multiplexer, just before the rotator, shown in Figure 2.16 on page 165. When the first LdOp completes, its data is loaded into the assembly buffer. When the second LdOp completes, its data are combined with the data in the assembly buffer and the correctly combined group of bytes is input to the rotator.

Design Note

Transparency of Cloned LdOps

The forming of the cloned LdOps and their progression through the remaining LU pipeline stages are essentially transparent to the scheduler, which only has a single Op entry for the original LdOp that caused the misaligned access. The exception to this statement is a signal generated within the LU pipeline and used in the scheduler's LdOp and StOp relative age determination process. This situation is discussed in Chapter 3.

Design Note

LdOp Worst Case Misaligned Access

The worst case misaligned access occurs when both of the cloned LdOps each require an Ld-TLB fill and a D-Cache line fill. The processor will basically do a Ld-TLB fill followed by a D-Cache fill followed by the second Ld-TLB fill followed by the second D-Cache fill. The two cloned LdOps are effectively executed by the majority of the LU pipeline just the same as two unrelated LdOps would be.

The approach taken in the K6 to handling misaligned access effectively results in a one clock penalty for such accesses. We will now examine how a misaligned access is handled when executing a StOp since there are some important similarities and differences from the above discussion for LdOps.

StOp Abort Cycles

Both the logical and linear address are calculated in the address calculation phase of Stage1 of the SU pipeline, using a) the base, scaled index, and displacement, b) the segment base, and c) taking into account the requirement of producing either a 16-bit style or a 32-bit style address. The lower twenty bits of the linear address is sent to the St-TLB, while the 32-bit logical address is sent to the Segment Limit and Access Check logic, along with the segment limit and access right bits. A page-related access check is also done, assuming a St-TLB hit. If there is a miss in the St-TLB then a table walk request to the table walk logic in the system interface unit initiates a table walk to retrieve the appropriate page translation and load it into the St-TLB and Ld-TLB (see Figure 2.22 on page 180).

If there is a fault in the processing of a StOp in the SU pipeline, the StOp will "stick" in Stage2 of this pipeline, just as was the case in the processing of a LdOp in the LU pipeline. The SU will then signal that along with the information about the fault to the OCU. Again, just as with the signal from the LU, when the OCU examines Ops in the bottom OpQuad of the scheduler and eventually attempts to commit the StOp for which the SU has sent the violation signal, the OCU recognizes that the StOp is not going to complete. It will retire any older Ops in the OpQuad and initiate an abort cycle which will flush both the upper and lower portions of the machine and vector the processor to a fault handler address in the OpQuad ROM. Just as was the case with a LdOp abort cycle, a snapshot of information in the SU is written into the OCU's Fault ID Register. As before, this information is readable from the OCU as SR1[2:0]. The fault handler OpQuad sequence examines the fault information and branches to the appropriate OpQuad sequence to handle this particular type of fault.

StOp Misaligned Accesses

The issues with misaligned StOp accesses are quite similar to those for misaligned LdOp accesses. Since the size of the access is known when the StOp begins execution, it can be determined if an access is misaligned when the computed address is available at the end of the address calculation phase of Stage1 of the SU pipeline.

If the access is misaligned, the reference is split by the SU into two memory writes and two associated store queue entries. The hardware in SU pipeline Stage1 splits the StOp into a cloned pair of StOps. One of the cloned StOps handles the first half of the access (the lower byte addresses) while the other cloned StOp handles the upper half of the access (the high byte addresses), and access sizes. Thus, the two cloned StOps have different addresses. The StOp accessing the lower half proceeds into Stage2 while the StOp accessing the upper half is left immediately behind it in Stage1. The two cloned StOps now progress down the SU pipeline back-to-back and get processed individually—(e.g., each StOp will get separately translated by St-TLB, and each will get separately looked up in the D-Cache). The two StOps are guaranteed to access data within neighboring aligned octets of memory. Unlike misaligned LdOps, each of these StOps creates a separate store queue entry which ultimately results in two separate writes into the D-Cache and/or out onto the system bus to main memory.

Design Note

Transparency of Cloned StOps

The forming of the cloned StOps and their progression through the remaining SU pipeline stages is transparent to the scheduler, which only has a single Op entry for the original StOp that caused the misaligned access. Further, the OCU, when committing the original StOp from its scheduler Op entry, recognizes that it has two associated store queue entries and commits both entries before viewing the original StOp as having been committed. The OCU is able to do this since the first store queue entry of a related pair of entries is so marked within the store queue entry itself. If the StOp has a fault then it must be aborted without retirement of either store queue entry. This issue is discussed in some detail in Chapter 3.

Design Note

StOp Worst Case Misaligned Access

The worst case misaligned access occurs when both of the cloned StOps require both a St-TLB fill and a D-Cache line fill. The processor will basically do a St-TLB fill followed by a D-Cache fill followed by the second St-TLB fill followed by the second D-Cache fill. The two cloned StOps are effectively executed by the majority of the SU pipeline just the same as two unrelated StOps would be.

BRU Pipeline

Before leaving this section, we present the pipeline diagram for a BrOp. This pipeline is described in some detail in Chapter 3 and is presented in this section with the RUX/RUY, LU, and SU pipelines for completeness.

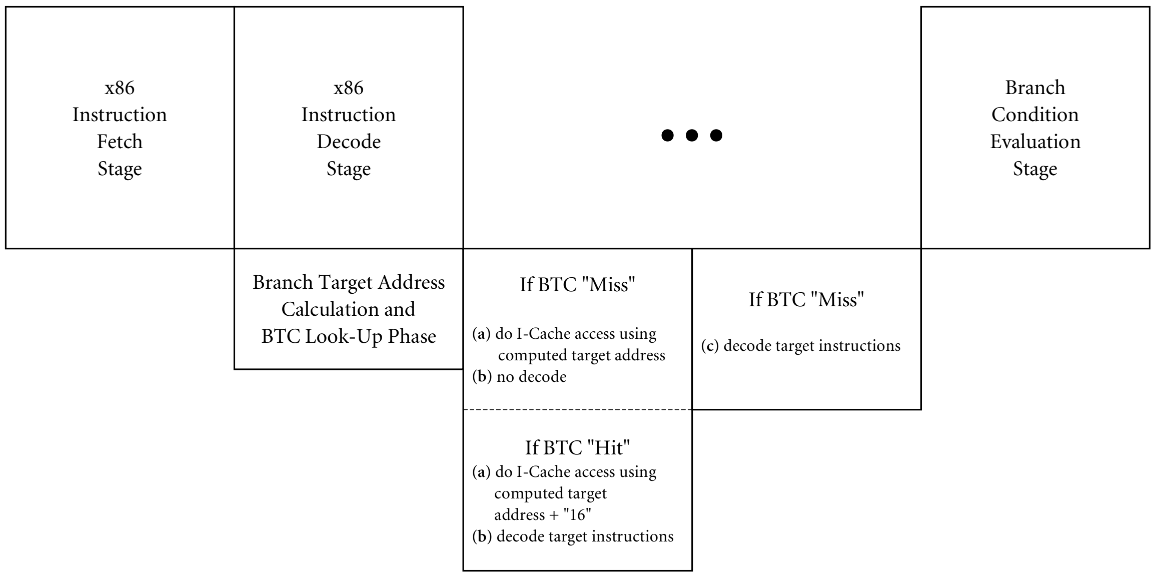

x86 Instruction Fetch Stage
x86 Instruction Decode Stage
• • •
Branch Condition Evaluation Stage
Branch Target Address Calculation and BTC Look-Up Phase
If BTC "Miss"
(a) do I-Cache access using computed target address
(b) no decode
If BTC "Miss"
(c) decode target instructions
If BTC "Hit"
(a) do I-Cache access using computed target address + "16"
(b) decode target instructions

Figure 2.21 The BRU Pipeline Stages

Handling Faults, Traps, and Precise Interrupts

We discussed the handling of faults and traps in two earlier segments of this chapter—the section titled "Status Flags, Faults, Traps, Interrupts, and Abort Cycles" beginning on page 83 and the section titled "Faults, Traps, Abort Cycles, and the Pipelines" beginning on page 168. You may want to review these sections before proceeding. As we pointed out in these sections, handling faults and traps in speculative, superscalar, pipelined processors can be difficult. There are several problems to contend with: (1) faults and traps can occur at various stages within a pipeline, (2) the results of operations that were issued and executed speculatively may no longer be valid after the fault or trap occurs, and (3) appropriate machine state must be saved in the event of faults or traps that abort a series of operations. There must be a relatively efficient mechanism to restore the state and restart execution at that point. The execution pipelines, various queues, and other processor resources may have to be flushed. Depending on the microarchitecture, this can be done either sequentially or in an overlapped fashion. The microarchitect basically attempts to minimize the latency incurred in restating execution while reducing the overall microarchitectural complexity in achieving this goal. Some simple examples of faults are a segment limit violation and a page fault. A simple example of a trap is the data breakpoint trap.

The K6 uses a uniform mechanism to handle faults, traps, and interrupts. In the K6, there are some faults that are detected at decode time by the hardware decoders, some that are detected in the scheduler by the OCU, and some that are detected during the execution of an OpQuad sequence by "manual" checks (and corresponding conditional branches) within the OpQuad sequence.

As just indicated, some faults are detected at decode time and some are detected at Op commit time. An "instruction length greater than sixteen bytes" fault is an example of a fault detected at decode time while a "memory-related" fault is a common example of a fault recognized at Op commit time (although initially detected at Op execution time). During each decode cycle, the decoders check for several x86 instruction set architecture-specific exception conditions, including a code segment overrun, an instruction fetch page fault, an instruction length greater than sixteen bytes, a nonlockable instruction with a "lock" prefix, and a floating-point not available condition. At the same time, the decoders also check for the assertion of any pending hardware interrupts, including INTR, NMI, SMI, STPCLK, INIT, and FLUSH. Some conditions are evaluated only during a successful decode cycle; other conditions, including all hardware interrupts, are decoded irrespective of any other possible decoding actions during the cycle.

When an active fault condition is detected, all short, long, and vector instruction decode cycles are inhibited and an exception vector decode cycle occurs in the following decode cycle. During this exception vector decode cycle, a special fault OpQuad sequence vector address is generated in place of a normal instruction vector address. The fault vector address is a fixed value except for low-order bits that are used to identify the particular fault condition that has been recognized and needs to be handled. Other than a different vector address, this behaves just like any instruction vector decode. There is no special synchronization within the scheduler with respect to the OpQuads already in the scheduler. The handling of this decoder-detected exception is naturally processed in the normal and proper program order. When multiple fault conditions are simultaneously detected, the faults are prioritized and the highest priority fault is recognized. In-order decode ensures precise fault handling for decoder-detected faults.

Faults that are detected during the execution of an OpQuad sequence do not insert an OpQuad into the top of the scheduler. A branch abort occurs due to a mispredicted BRCOND Op based on status flag values that were set by a preceding RegOp which performed some type of "check." The abort cycle results in a redirection to an alternate OpQuad sequence.

Re-examining The Abort Cycle

When discussing the abort cycle in the earlier sections of this chapter cited above, we introduced the notion of flushing and restarting the upper and lower portions of the processor; the former is part of the BRCOND

Op resolution cycle and the latter is part of the abort cycle in the case of branch aborts.

When a "flush" occurs, all Ops in the bottom portion of the machine are basically discarded, all abortable state is discarded, and the "valid bits" of all Ops in the scheduler are set to invalid. In the case of a mispredicted branch, the upper portion of the machine is flushed and then restarted to begin fetching from the mispredicted path. This is done as part of the resolution of the branch versus as part of the abort cycle. The restarting of the upper portion of the processor consists of reloading the PCs (program counters) in the decoders and sending the mispredicted fetch address to the I-Cache.

"Exception" aborts, discussed above, are different in that there is no "resolution" stage, only an abort cycle. During the abort cycle, the bottom portion of the machine is also flushed similar to the case of a mispredicted branch and the upper portion is flushed and restarted. In this case the scheduler vectors to the start of a "fault handler" OpQuad sequence in the OpQuad ROM.

Precise Interrupts and Precise Exceptions

As discussed earlier, interrupts are asynchronous events that occur independently of the synchronous activity within the microprocessor. The x86 instruction set architecture employs a precise interrupt model. This model holds that an interrupted process can resume correct execution after the interrupt has been serviced. When the microprocessor detects an interrupt, it:

1. halts the execution of the current instruction stream (i.e., the current process).
2. saves enough of the state of the machine so processing can resume at the point the interrupt was detected.
3. activates an interrupt handling routine to service the interrupt.
4. resumes processing after the interrupt has been serviced and after the saved state has been restored.

A precise exception model is implemented for program-related exceptions; see the section titled "Status Flags, Faults, Traps, Interrupts, and Abort Cycles" beginning on page 83. The issues that apply to precise interrupts can be equally challenging to resolve as those dealing with precise program exceptions. The following discussion is couched in the context of precise interrupts and can be extended to precise exceptions.

Let us use $\textit{state}_{\text{resume}}$ to be the machine state that is required to resume processing if an interrupt occurs. And, let us use $\textit{state}_{\text{no interrupt}}$ to be the state that is required to continue processing if no interrupt occurred. What must be done to support precise interrupts is to guarantee that $state_{\text{resume}} = state_{\text{no interrupt}}$ which essentially means that the saved machine state that exists after the interrupt has been serviced is that same saved state that would have existed if the interrupt had not occurred at all. This is reasonably straightforward to do in conventional, nonpipelined, in-order processors. It's a bit more difficult in pipelined processors, particularly those that support out-of-order execution.

Suggested Readings

Precise Interrupts

We have included three articles on the CD-ROM that deal either wholly or in part with the issue of providing for precise interrupts in pipelined processors. The article by Smith and Pleszkum, is considered by many as a classic paper in this area. The second article, by Walker and Cragon, develops a taxonomy of design strategies for providing precise interrupts based on a detailed examination of fifteen processors that support concurrent instruction execution. The third article, by Moudgill and Vassiliadis, gives an interesting treatment of the topic of precise interrupts.

1. James E. Smith and Andrew R. Pleszkum, "Implementation of Precise Interrupts on Pipelined Processors," in *Proceedings of the 12th Annual International Symposium on Computer Architecture,* June 1985, pp. 34-44.
2. Wade Walker and Harvey G. Cragon, "Interrupt Processing in Concurrent Processors," *Computer,* Vol. 28, No. 6, June 1995, pp. 36-46.
3. Mayan Moudgill and Stamatis Vassiliadis, "Precise Interrupts," *IEEE Micro*, Vol. 16, No. 1, February 1996, pp.58-67.

We also suggest a paper by Hwu and Patt that presents an interesting hardware checkpointing scheme to support precise interrupts, W. M. Hwu and Y N. Patt, "Checkpoint Repair for Out-of-Order Execution Machines," *Proceedings of the 14th Annual International Symposium on Computer Architecture,* 1987. Similar checkpointing schemes have been widely used in software systems.

Smith and Pleszkum identify three conditions that must hold to support precise interrupts:

1. all instructions that issued prior to the instruction that was executing when the interrupt was detected have completed and have modified the process state correctly.
2. all instructions after the one indicated by the saved program counter execute only after control is returned to the interrupted process.
3. if the interrupt were caused by an instruction (versus some activity external to the processor), then the saved program counter must point to that instruction.

The K6 supports the x86 instruction set architecture precise interrupt model. An interrupt, when detected, is directed to the decoders. As described in the preceding pages, hardware interrupts are just cases of exception vector decodes. Once the appropriate interrupt handling OpQuad sequence is initiated, the K6 goes through the details of storing the appropriate program state as defined by the x86 instruction set architecture. This state is restored through execution of an IRET instruction at the end of the x86 interrupt handler before returning control to the interrupted instruction stream.

System Interface

All transactions involving the system bus are mediated by the system interface unit shown in Figure 2.2 on page 69. However, transactions may involve the L2-Cache only, the system bus only, or both. System bus transactions can proceed concurrently with L2-only transactions. L2-only transactions may complete out of order with respect to system bus-only transactions.

Requests are presented to the system interface over six separate interfaces that are shown in Figure 2.22:

1. L1 I-Cache read interface, IC.
2. L1 D-Cache read interface, DC.
3. L1 D-Cache write back interface.
4. L2-Cache write back interface.
5. Table Walk Unit read/write interface, TW.
6. Write Pipeline/Merge Unit interface, PM.

Each interface shown in this figure consists of:

1. request and handshake signals.
2. an address bus.
3. a data bus.
4. various attribute and status signals.

Data transfers may involve 1 or 4 octets, where an octet is eight bytes or sixty-four bits. As mentioned earlier, standard Socket 7 pinouts and protocols are implemented and are discussed in more detail in Chapter 4. Also, operation of the system bus at 100-MHz is supported. The bus operates at a whole or half-integer divisor of the core frequency (e.g., 3.5X for a 350-MHz processor and a 100-MHz system bus). Because the L2-Cache is on-chip, it runs at the core speed of the microprocessor itself. The L2-Cache has separate read and write ports, allowing one read and one write per cycle.

Neither inclusion nor exclusion is enforced between the L1- and L2-Caches. L2-Cache line misses that result in an L2-Cache line fill cause simultaneous L1- and L2-fill operations. A given line is never marked modified in both the L1- and L2-Caches.

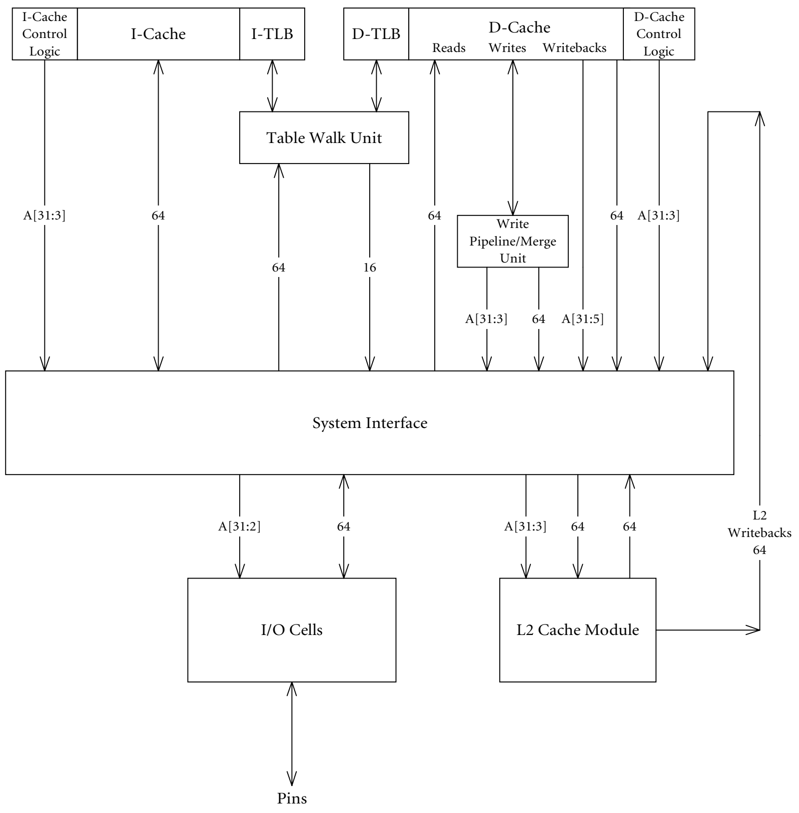


Figure 2.22 System Interface Unit

table walk unit

The table walk unit undertakes x86-based page table walks in response to requests from either the I-Cache TLB or the D-Cache TLB. It contains an 8-entry PDE cache (page directory entry) to reduce the average table walk time from a 2-level table walk to just a single table read. Request for such a table walk are shown in the LdOp Pipeline and L1 D-Cache Access diagram shown in Figure 2.16 on page 165 and in Figure 2.19 on page 168 for the StOp pipeline and store queue access.

The write pipeline/merge unit handles all single-octet writes from the D-Cache, merging and pipelining them onto the system bus wherever possible. The system bus supports 2-deep bus transaction pipelining.

The critical importance of how the microprocessor is integrated into the other elements that make up a typical system (i.e., platform) is one of the central themes of this book and is dealt with in a number of the following chapters.

Chapter Summary

A general overview of the K6 3D microarchitecture has been given. In particular, we examined a number of the design choices that were made which were part of the decisions that ultimately led to the K6 3D's microarchitecture. Some of these choices were centered around

1. the predecode scheme which is done (a) at cache fill time, outside of the fetch/decode/execute pipeline and (b) without knowledge of the actual instruction boundaries.
2. the number and types of decoders used.
3. the unification of most major control requirements (e.g., reservation stations, reorder buffer, and register renaming) into one structure—the scheduler and its centralized buffer.
4. the focus on short pipelines and short latencies, (e.g., short branch prediction, fetch redirection, execution latencies, short branch misprediction penalty, and short misaligned memory access penalty).
5. the use of an internal RISC-like microarchitecture in conjunction with the translation of x86 instructions into corresponding short or long sequences of RSIC-like Ops (OpQuads and OpQuad sequences).

In fact, as pointed out in Chapter 1, this case study provides a detailed and coherent context for studying the wealth of design problems encountered in processor design that are covered in conventional textbooks in computer architecture. Hopefully, the rich set of design issues that result from the above set of topics and other design choices discussed in this chapter have helped you understand the interrelationships and dependencies that exist among them.

Suggested Readings

Instruction Level Parallelism

Many of the approaches discussed in this chapter are discussed in the literature under the general phrase "instruction level parallelism (ILP)". Two articles we recommend, in addition to those already cited, are given here. The article, "The 16-Fold Way: A Microparallel Taxonomy," by Barton J. Sano and Alvin M. Despain, *Proceedings of the 26th Annual International Symposium on Microarchitecture,* 1993, presents an interesting taxonomy for processors that have multiple-instruction processing capabilities, including some of those discussed here. Additionally, the article, by Roger Espasa and Mateo Valero, "Exploiting Instruction- And Data-Level Parallelism," *IEEE Micro*, 1997, describes a design approach which combines ILP design approaches with data-level parallelism techniques (i.e., vectorization techniques). Both of these articles are on the CD-ROM.

Additionally, two presentations by Bruce Shriver related to these issues are on the CD-ROM. Their titles are, "*Instruction Level Parallelism*," and "*The Evolution of High-Performance ISAs*."

Now that we have given an overview of the K6 3D's microarchitecture, we will, in the next chapter, examine three of the major aspects of the it microarchitecture in considerably more depth to aid you in understanding some of the implementation issues associated with specific design approaches.