6x86 PROCESSOR

Superscalar, Superpipelined,
Sixth-generation, x86 Compatible CPU

Cyrix® Advancing the Standards

Product Overview

1

1. ARCHITECTURE OVERVIEW

The Cyrix 6x86 CPU is a leader in the sixth generation of high performance, x86-compatible processors. Increased performance is accomplished by the use of superscalar and superpipelined design techniques.

The 6x86 CPU is superscalar in that it contains two separate pipelines that allow multiple instructions to be processed at the same time. The use of advanced processing technology and the increased number of pipeline stages (superpipelining) allow the 6x86 CPU to achieve clocks rates of 100 MHz and above.

Through the use of unique architectural features, the 6x86 processor eliminates many data dependencies and resource conflicts, resulting in optimal performance for both 16-bit and 32-bit x86 software.

The 6x86 CPU contains two caches: a 16-KByte dual-ported unified cache and a 256-byte instruction line cache. Since the unified cache can store instructions and data in any ratio, the unified cache offers a higher hit rate than separate data and instruction caches of equal size. An increase in overall cache-to-integer unit bandwidth is achieved by supplementing the unified cache with a small, high-speed, fully associative instruction line cache. The inclusion of the instruction line cache avoids excessive conflicts between code and data accesses in the unified cache.

The on-chip FPU allows floating point instructions to execute in parallel with integer instructions and features a 64-bit data interface. The FPU incorporates a four-deep instruction queue and a four-deep store queue to facilitate parallel execution.

The 6x86 CPU operates from a 3.3 volt power supply resulting in reasonable power consumption at all frequencies. In addition, the 6x86 CPU incorporates a low power suspend mode, stop clock capability, and system management mode (SMM) for power sensitive applications.

1.1 Major Functional Blocks

The 6x86 processor consists of five major functional blocks, as shown in the overall block diagram on the first page of this manual:

- Integer Unit
- Cache Unit
- Memory Management Unit
- Floating Point Unit
- Bus Interface Unit

Instructions are executed in the X and Y pipelines within the Integer Unit and also in the Floating Point Unit (FPU). The Cache Unit stores the most recently used data and instructions to allow fast access to the information by the Integer Unit and FPU.

Physical addresses are calculated by the Memory Management Unit and passed to the Cache Unit and the Bus Interface Unit (BIU). The BIU provides the interface between the external system board and the processor's internal execution units.

1.2 Integer Unit

The Integer Unit (Figure 1-1) provides parallel instruction execution using two seven-stage integer pipelines. Each of the two pipelines, X and Y, can process several instructions simultaneously.

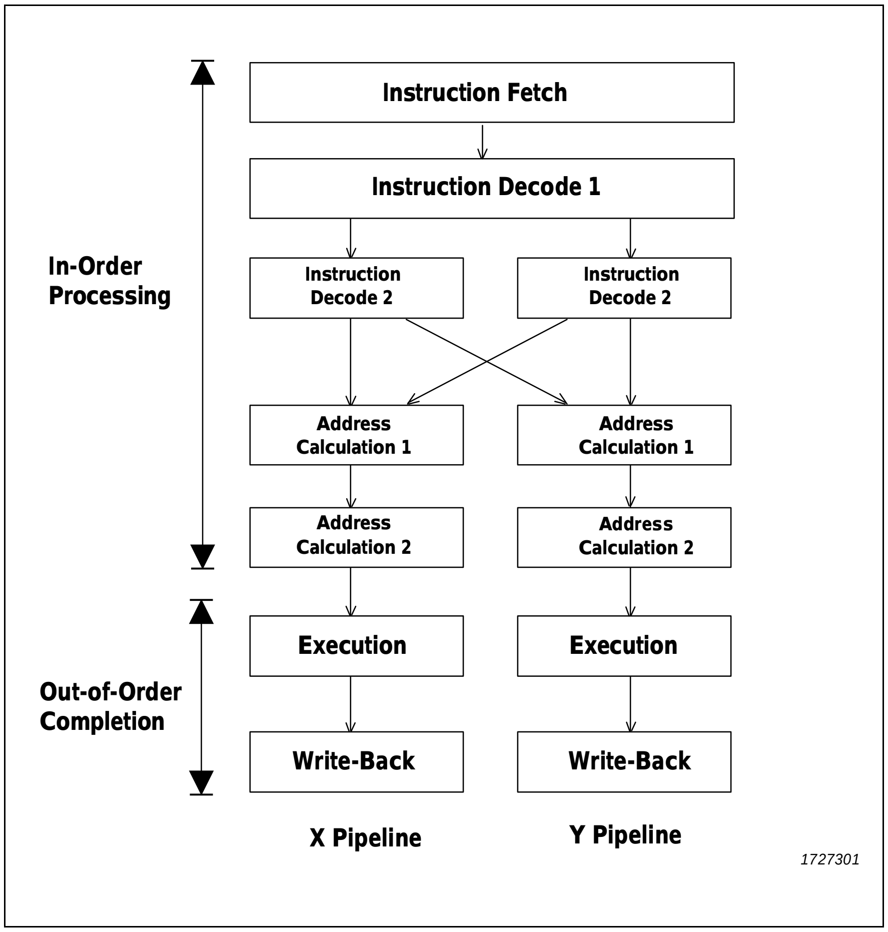

Figure 1-1. Integer Unit

The Integer Unit consists of the following pipeline stages:

- Instruction Fetch (IF)
- Instruction Decode 1 (ID1)
- Instruction Decode 2 (ID2)
- Address Calculation 1 (AC1)
- Address Calculation 2 (AC2)
- Execute (EX)
- Write-Back (WB)

The instruction decode and address calculation functions are both divided into superpipelined stages.

1.2.1 Pipeline Stages

The **Instruction Fetch** (IF) stage, shared by both the X and Y pipelines, fetches 16 bytes of code from the cache unit in a single clock cycle. Within this section, the code stream is checked for any branch instructions that could affect normal program sequencing.

If an unconditional or conditional branch is detected, branch prediction logic within the IF stage generates a predicted target address for the instruction. The IF stage then begins fetching instructions at the predicted address.

The superpipelined **Instruction Decode** function contains the ID1 and ID2 stages. ID1, shared by both pipelines, evaluates the code stream provided by the IF stage and determines the number of bytes in each instruction. Up to two instructions per clock are delivered to the ID2 stages, one in each pipeline.

The ID2 stages decode instructions and send the decoded instructions to either the X or Y pipeline for execution. The particular pipeline is chosen, based on which instructions are already in each pipeline and how fast they are expected to flow through the remaining pipeline stages.

The **Address Calculation** function contains two stages, AC1 and AC2. If the instruction refers to a memory operand, the AC1 calculates a linear memory address for the instruction.

The AC2 stage performs any required memory management functions, cache accesses, and register file accesses. If a floating point instruction is detected by AC2, the instruction is sent to the FPU for processing.

The **Execute** (EX) stage executes instructions using the operands provided by the address calculation stage.

The **Write-Back** (WB) stage is the last IU stage. The WB stage stores execution results either to a register file within the IU or to a write buffer in the cache control unit.

1.2.2 Out-of-Order Processing

If an instruction executes faster than the previous instruction in the other pipeline, the instructions may complete out of order. All instructions are processed in order, up to the EX stage. While in the EX and WB stages, instructions may be completed out of order.

If there is a data dependency between two instructions, the necessary hardware interlocks are enforced to ensure correct program execution. Even though instructions may complete out of order, exceptions and writes resulting from the instructions are always issued in program order.

1.2.3 Pipeline Selection

In most cases, instructions are processed in either pipeline and without pairing constraints on the instructions. However, certain instructions are processed only in the X pipeline:

- Branch instructions
- Floating point instructions
- Exclusive instructions

Branch and floating point instructions may be paired with a second instruction in the Y pipeline.

Exclusive Instructions cannot be paired with instructions in the Y pipeline. These instructions typically require multiple memory accesses. Although exclusive instructions may not be paired, hardware from both pipelines is used to accelerate instruction completion. Listed below are the 6x86 CPU exclusive instruction types:

- Protected mode segment loads
- Special register accesses (Control, Debug, and Test Registers)
- String instructions
- Multiply and divide
- I/O port accesses
- Push all (PUSHA) and pop all (POPA)
- Intersegment jumps, calls, and returns

1.2.4 Data Dependency Solutions

When two instructions that are executing in parallel require access to the same data or register, one of the following types of data dependencies may occur:

- Read-After-Write (RAW)
- Write-After-Read (WAR)
- Write-After-Write (WAW)

Data dependencies typically force serialized execution of instructions. However, the 6x86 CPU implements three mechanisms that allow parallel execution of instructions containing data dependencies:

- Register Renaming
- Data Forwarding
- Data Bypassing

The following sections provide detailed examples of these mechanisms.

1.2.4.1 Register Renaming

The 6x86 CPU contains 32 physical general purpose registers. Each of the 32 registers in the register file can be temporarily assigned as one of the general purpose registers defined by the x86 architecture (EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP). For each register write operation a new physical register is selected to allow previous data to be retained temporarily. Register renaming effectively removes all WAW and WAR dependencies. The programmer does not have to consider register renaming; it is completely transparent to both the operating system and application software.

Example #1 - Register Renaming Eliminates Write-After-Read (WAR) Dependency

A WAR dependency exists when the first in a pair of instructions reads a logical register, and the second instruction writes to the same logical register. This type of dependency is illustrated by the pair of instructions shown below:

X PIPE	Y PIPE
(1) MOV BX, AX	(2) ADD AX, CX
BX ← AX	AX ← AX + CX

Note: In this and the following examples the original instruction order is shown in parentheses.

In the absence of register renaming, the ADD instruction in the Y pipe would have to be stalled to allow the MOV instruction in the X pipe to read the AX register.

The 6x86 CPU, however, avoids the Y pipe stall (Table 1-1). As each instruction executes, the results are placed in new physical registers to avoid the possibility of overwriting a logical register value and to allow the two instructions to complete in parallel (or out of order) rather than in sequence.

Table 1-1. Register Renaming with WAR Dependency

Instruction	Physical Register Contents					Action	
	Reg0	Reg1	Reg2	Reg3	Reg4	Pipe	
(Initial)	AX	BX	CX				
MOV BX, AX	AX		CX	BX		X	Reg3 ← Reg0
ADD AX, CX			CX	BX	AX	Y	Reg4 ← Reg0 + Reg2

Note: The representation of the MOV and ADD instructions in the final column of Table 1-1 are completely independent.

Example #2 - Register Renaming Eliminates Write-After-Write (WAW) Dependency

A WAW dependency occurs when two consecutive instructions perform writes to the same logical register. This type of dependency is illustrated by the pair of instructions shown below:

X PIPE	**Y PIPE**
(1) ADD AX, BX	(2) MOV AX, [mem]
AX ←AX + BX	AX ← [mem]

Without register renaming, the MOV instruction in the Y pipe would have to be stalled to guarantee that the ADD instruction in the X pipe would write its results to the AX register first.

The 6x86 CPU uses register renaming and avoids the Y pipe stall. The contents of the AX and BX registers are placed in physical registers (Table 1-2). As each instruction executes, the results are placed in new physical registers to avoid the possibility of overwriting a logical register value and to allow the two instructions to complete in parallel (or out of order) rather than in sequence.

Table 1-2. Register Renaming with WAW Dependency

Instruction	Physical Register Contents				Action	
	Reg0	Reg1	Reg2	Reg3	Pipe	
(Initial)	AX	BX				
ADD AX, BX		BX	AX		X	Reg2 ← Reg0 + Reg1
MOV AX, [mem]		BX		AX	Y	Reg3 ← [mem]

Note: All subsequent reads of the logical register AX will refer to Reg 3, the result of the MOV instruction.

1.2.4.2 Data Forwarding

Register renaming alone cannot remove RAW dependencies. The 6x86 CPU uses two types of data forwarding in conjunction with register renaming to eliminate RAW dependencies:

- Operand Forwarding
- Result Forwarding

Operand forwarding takes place when the first in a pair of instructions performs a move from register or memory, and the data that is read by the first instruction is required by the second instruction. The 6x86 CPU performs the read operation and makes the data read available to both instructions simultaneously.

Result forwarding takes place when the first in a pair of instructions performs an operation (such as an ADD) and the result is required by the second instruction to perform a move to a register or memory. The 6x86 CPU performs the required operation and stores the results of the operation to the destination of both instructions simultaneously.

Example #3 - Operand Forwarding Eliminates Read-After-Write (RAW) Dependency

A RAW dependency occurs when the first in a pair of instructions performs a write, and the second instruction reads the same register. This type of dependency is illustrated by the pair of instructions shown below in the X and Y pipelines:

X PIPE	Y PIPE
(1) MOV AX, [mem]	(2) ADD BX, AX
AX ← [mem]	BX ← AX + BX

The 6x86 CPU uses operand forwarding and avoids a Y pipe stall (Table 1-3). Operand forwarding allows simultaneous execution of both instructions by first reading memory and then making the results available to both pipelines in parallel.

Table 1-3. Example of Operand Forwarding

Instruction	Physical Register Contents				Action	
	Reg0	Reg1	Reg2	Reg3	Pipe	
(Initial)	AX	BX				
MOV AX, [mem]		BX	AX		X	Reg2 ← [mem]
ADD BX, AX			AX	BX	Y	Reg3 ← [mem] + Reg1

Operand forwarding can only occur if the first instruction does not modify its source data. In other words, the instruction is a move type instruction (for example, MOV, POP, LEA). Operand forwarding occurs for both register and memory operands. The size of the first instruction destination and the second instruction source must match.

Example #4 - Result Forwarding Eliminates Read-After-Write (RAW) Dependency

In this example, a RAW dependency occurs when the first in a pair of instructions performs a write, and the second instruction reads the same register. This dependency is illustrated by the pair of instructions in the X and Y pipelines, as shown below:

X PIPE	Y PIPE
(1) ADD AX, BX	(2) MOV [mem], AX
AX ←AX + BX	[mem] ← AX

The 6x86 CPU uses result forwarding and avoids a Y pipe stall (Table 1-4). Instead of transferring the contents of the AX register to memory, the result of the previous ADD instruction (Reg0 + Reg1) is written directly to memory, thereby saving a clock cycle.

Table 1-4. Result Forwarding Example

Instruction	Physical Register Contents			Action	
	Reg0	Reg1	Reg2	Pipe	
(Initial)	AX	BX			
ADD AX, BX		BX	AX	X	Reg2 ←Reg0 + Reg1
MOV [mem], AX		BX	AX	Y	[mem] ← Reg0 +Reg1

The second instruction must be a move instruction and the destination of the second instruction may be either a register or memory.

1.2.4.3 Data Bypassing

In addition to register renaming and data forwarding, the 6x86 CPU implements a third data dependency-resolution technique called data bypassing. Data bypassing reduces the performance penalty of those memory data RAW dependencies that cannot be eliminated by data forwarding.

Data bypassing is implemented when the first in a pair of instructions writes to memory and the second instruction reads the same data from memory. The 6x86 CPU retains the data from the first instruction and passes it to the second instruction, thereby eliminating a memory read cycle. Data bypassing only occurs for cacheable memory locations.

Example #1- Data Bypassing with Read-After-Write (RAW) Dependency

In this example, a RAW dependency occurs when the first in a pair of instructions performs a write to memory and the second instruction reads the same memory location. This dependency is illustrated by the pair of instructions in the X and Y pipelines as shown below:

X PIPE	Y PIPE
(1) ADD [mem], AX	(2) SUB BX, [mem]
[mem] ←[mem] + AX	BX ← BX - [mem]

The 6x86 CPU uses data bypassing and stalls the Y pipe for only one clock by eliminating the Y pipe's memory read cycle (Table 1-5). Instead of reading memory in the Y pipe, the result of the previous instruction ([mem] + Reg0) is used to subtract from Reg1, thereby saving a memory access cycle.

Table 1-5. Example of Data Bypassing

Instruction	Physical Register Contents			Action	
	Reg0	Reg1	Reg2	Pipe	
(Initial)	AX	BX			
ADD [mem], AX	AX	BX		X	[mem] ← [mem] + Reg0
SUB BX, [mem]	AX		BX	Y	Reg2 ← Reg1 - {[mem] + Reg0}

1.2.5 Branch Control

Branch instructions occur on average every four to six instructions in x86-compatible programs. When the normal sequential flow of a program changes due to a branch instruction, the pipeline stages may stall while waiting for the CPU to calculate, retrieve, and decode the new instruction stream. The 6x86 CPU minimizes the performance degradation and latency of branch instructions through the use of branch prediction and speculative execution.

1.2.5.1 Branch Prediction

The 6x86 CPU uses a 256-entry, 4-way set associative Branch Target Buffer (BTB) to store branch target addresses and branch prediction information. During the fetch stage, the instruction stream is checked for the presence of branch instructions. If an unconditional branch instruction is encountered, the 6x86 CPU accesses the BTB to check for the branch instruction's target address. If the branch instruction's target address is found in the BTB, the 6x86 CPU begins fetching at the target address specified by the BTB.

In case of conditional branches, the BTB also provides history information to indicate whether the branch is more likely to be taken or not taken. If the conditional branch instruction is found in the BTB, the 6x86 CPU begins fetching instructions at the predicted target address. If the conditional branch misses in the BTB, the 6x86 CPU predicts that the branch will not be taken, and instruction fetching continues with the next sequential instruction. The decision to fetch the taken or not taken target address is based on a four-state branch prediction algorithm.

Once fetched, a conditional branch instruction is first decoded and then dispatched to the X pipeline only. The conditional branch instruction proceeds through the X pipeline and is then resolved in either the EX stage or the WB stage. The conditional branch is resolved in the EX stage, if the instruction responsible for setting the condition codes is completed prior to the execution of the branch. If the instruction that sets the condition codes is executed in parallel with the branch, the conditional branch instruction is resolved in the WB stage.

Correctly predicted branch instructions execute in a single core clock. If resolution of a branch indicates that a misprediction has occurred, the 6x86 CPU flushes the pipeline and starts fetching from the correct target address. The 6x86 CPU prefetches both the predicted and the non-predicted path for each conditional branch, thereby eliminating the cache access cycle on a misprediction. If the branch is resolved in the EX stage, the resulting misprediction latency is four cycles. If the branch is resolved in the WB stage, the latency is five cycles.

Since the target address of return (RET) instructions is dynamic rather than static, the 6x86 CPU caches target addresses for RET instructions in an eight-entry return stack rather than in the BTB. The return address is pushed on the return stack during a CALL instruction and popped during the corresponding RET instruction.

1.2.5.2 Speculative Execution

The 6x86 CPU is capable of speculative execution following a floating point instruction or predicted branch. Speculative execution allows the pipelines to continuously execute instructions following a branch without stalling the pipelines waiting for branch resolution. The same mechanism is used to execute floating point instructions (see Section 1.5) in parallel with integer instructions.

The 6x86 CPU is capable of up to four levels of speculation (i.e., combinations of four conditional branches and floating point operations). After generating the fetch address using branch prediction, the CPU checkpoints the machine state (registers, flags, and processor environment), increments the speculation level counter, and begins operating on the predicted instruction stream.

Once the branch instruction is resolved, the CPU decreases the speculation level. For a correctly predicted branch, the status of the checkpointed resources is cleared. For a branch misprediction, the 6x86 processor generates the correct fetch address and uses the checkpointed values to restore the machine state in a single clock.

In order to maintain compatibility, writes that result from speculatively executed instructions are not permitted to update the cache or external memory until the appropriate branch is resolved. Speculative execution continues until one of the following conditions occurs:

1) A branch or floating point operation is decoded and the speculation level is already at four.

2) An exception or a fault occurs.

3) The write buffers are full.

4) An attempt is made to modify a non-checkpointed resource (i.e., segment registers, system flags).

1.3 Cache Units

The 6x86 CPU employs two caches, the Unified Cache and the Instruction Line Cache (Figure 1-2).

1.3.1 Unified Cache

The 16-KByte unified write-back cache functions as the primary data cache and as the secondary instruction cache. Configured as a four-way set-associative cache, the cache stores up to 16 KBytes of code and data in 512 lines. The cache is dual-ported and allows any two of the following operations to occur in parallel:

- Code fetch
- Data read (X pipe, Y pipeline or FPU)
- Data write (X pipe, Y pipeline or FPU)

The unified cache uses a pseudo-LRU replacement algorithm and can be configured to allocate new lines on read misses only or on read and write misses. More information concerning the unified cache can be found in Section 2.7.1 (Page 2-52).

1.3.2 Instruction Line Cache

The fully associative 256-byte instruction line cache serves as the primary instruction cache. The instruction line cache is filled from the unified cache through the data bus. Fetches from the integer unit that hit in the instruction line cache do not access the unified cache. If an instruction line cache miss occurs, the instruction line data from the unified cache is transferred to the instruction line cache and the integer unit, simultaneously.

The instruction line cache uses a pseudo-LRU replacement algorithm. To ensure proper operation in the case of self-modifying code, any writes to the unified cache are checked against the contents of the instruction line cache. If a hit occurs in the instruction line cache, the appropriate line is invalidated.

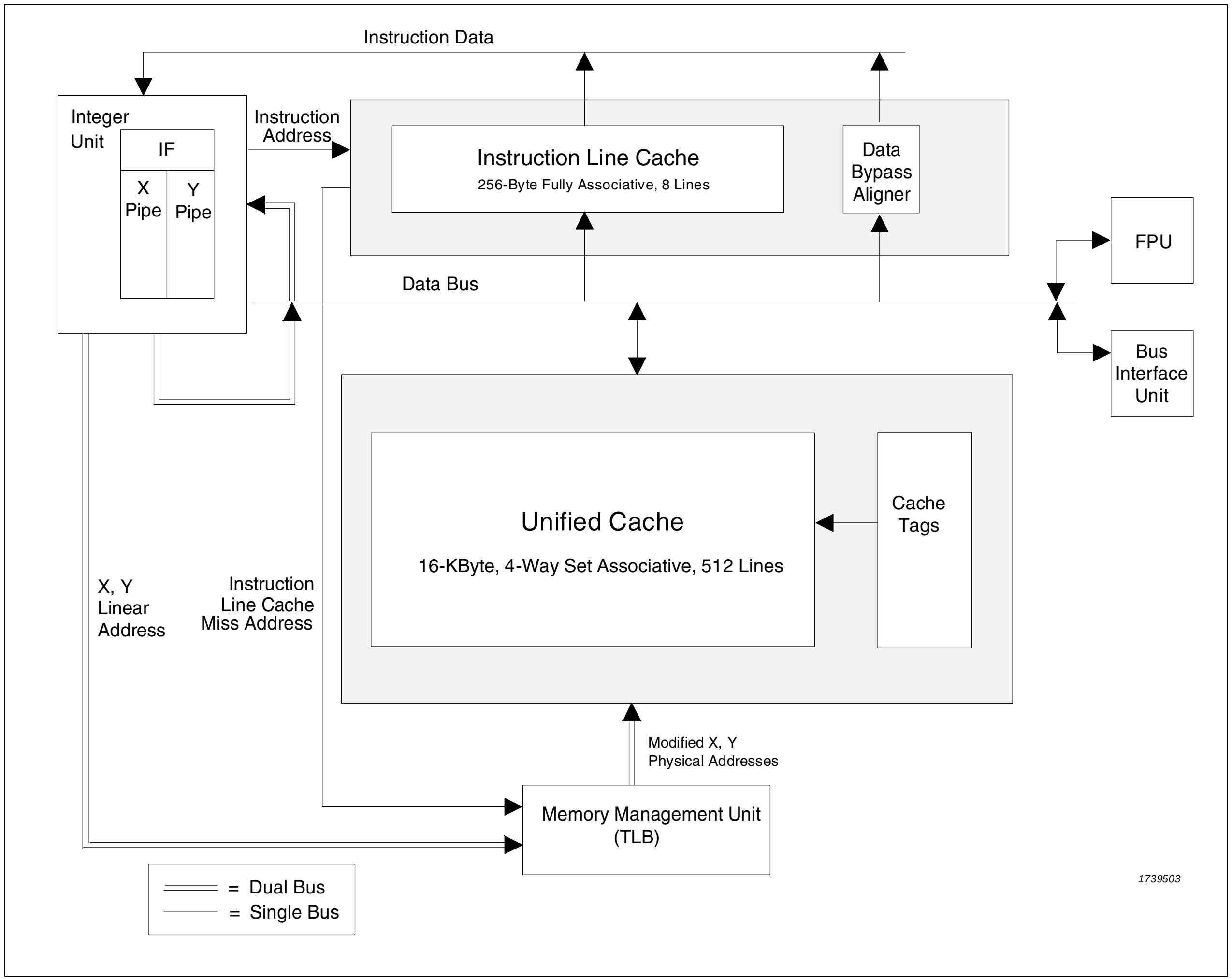

Figure 1-2. Cache Unit Operations

1.4 Memory Management Unit

The Memory Management Unit (MMU), shown in Figure 1-3, translates the linear address supplied by the IU into a physical address to be used by the unified cache and the bus interface. Memory management procedures are x86 compatible, adhering to standard paging mechanisms.

The 6x86 MMU includes two paging mechanisms (Figure 1-3), a traditional paging mechanism, and a 6x86 variable-size paging mechanism.

1.4.1 Variable-Size Paging Mechanism

The Cyrix variable-size paging mechanism allows software to map pages between 4 KBytes and 4 GBytes in size. The large contiguous memories provided by this mechanism help avoid TLB (Translation Lookaside Buffer) thrashing [see Section 2.6.4 (Page 2-45)] associated with some operating systems and applications. For example, use of a single large page instead of a series of small 4-KByte pages can greatly improve performance in an application using a large video memory buffer.

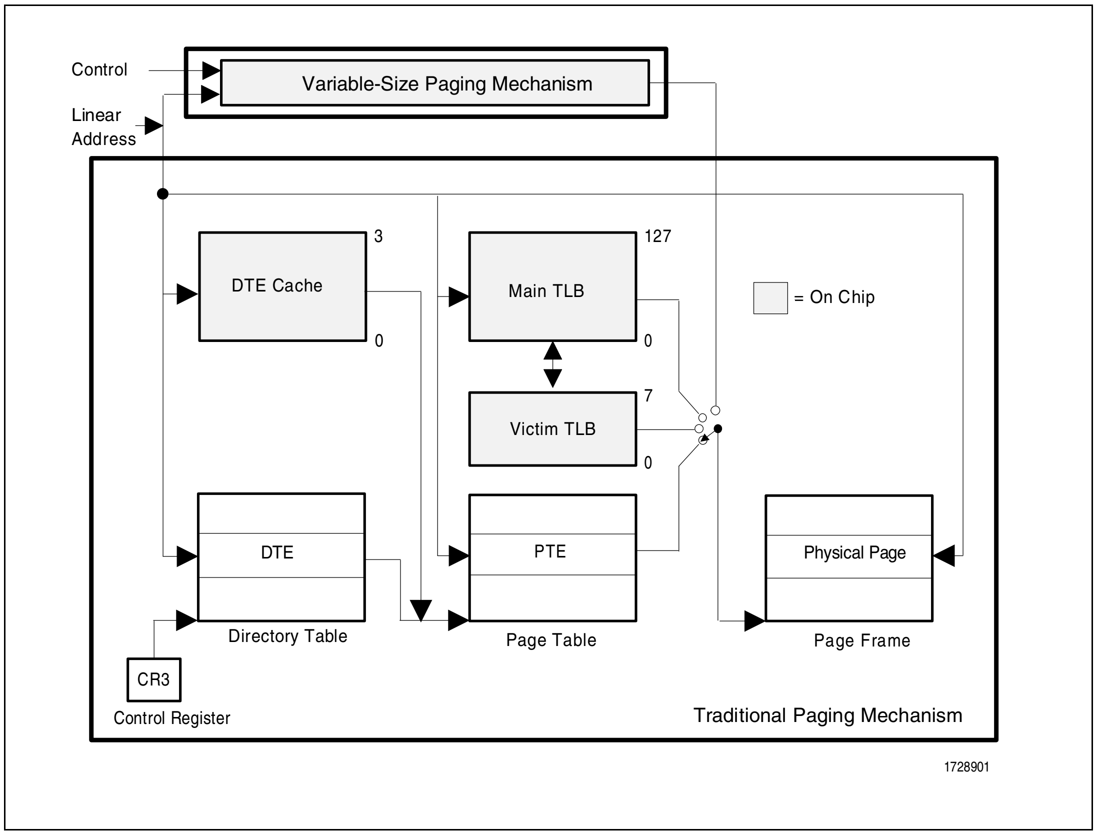

Figure 1-3. Paging Mechanism within the Memory Management Unit

1.4.2 Traditional Paging Mechanism

The traditional paging mechanism has been enhanced on the 6x86 CPU with the addition of the Directory Table Entry (DTE) cache and the Victim TLB. The main TLB (Translation Lookaside Buffer) is a direct-mapped 128-entry cache for page table entries.

The four-entry fully associative DTE cache stores the most recent DTE accesses. If a Page Table Entry (PTE) miss occurs followed by a DTE hit, only a single memory access to the PTE table is required.

The Victim TLB stores PTEs which have been displaced from the main TLB due to a TLB miss. If a PTE access occurs while the PTE is stored in the victim TLB, the PTE in the victim TLB is swapped with a PTE in the main TLB. This has the effect of selectively increasing TLB associativity. The 6x86 CPU updates the eight-entry fully associative victim TLB on an oldest entry replacement basis.

1.5 Floating Point Unit

The 6x86 Floating Point Unit (FPU) interfaces to the integer unit and the cache unit through a 64-bit bus. The 6x86 FPU is x87 instruction set compatible and adheres to the IEEE-754 standard. Since most applications contain FPU instructions mixed with integer instructions, the 6x86 FPU achieves high performance by completing integer and FPU operations in parallel.

FPU Parallel Execution

The 6x86 CPU executes integer instructions in parallel with FPU instructions. Integer instructions may complete out of order with respect to the FPU instructions. The 6x86 CPU maintains x86 compatibility by signaling exceptions and issuing write cycles in program order.

As previously discussed, FPU instructions are always dispatched to the integer unit's X pipeline. The address calculation stage of the X pipeline checks for memory management exceptions and accesses memory operands used by the FPU. If no exceptions are detected, the 6x86 CPU checkpoints the state of the CPU and, during AC2, dispatches the floating point instruction to the FPU instruction queue. The 6x86 CPU can then complete any subsequent integer instructions speculatively and out of order relative to the FPU instruction and relative to any potential FPU exceptions which may occur.

As additional FPU instructions enter the pipeline, the 6x86 CPU dispatches up to four FPU instructions to the FPU instruction queue. The 6x86 CPU continues executing speculatively and out of order, relative to the FPU queue, until the 6x86 CPU encounters one of the conditions that causes speculative execution to halt. As the FPU completes instructions, the speculation level decreases and the check-pointed resources are available for reuse in subsequent operations. The 6x86 FPU also uses a set of four write buffers to prevent stalls due to speculative writes.

1.6 Bus Interface Unit

The Bus Interface Unit (BIU) provides the signals and timing required by external circuitry. The signal descriptions and bus interface timing information is provided in Chapters 3 and 4 of this manual.